3
Volume 24 Number X June 2000 ISSN 0350-5596

Informatica

An International Journal of Computing and Informatics

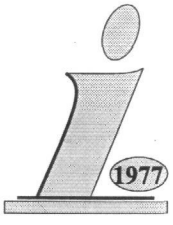
Special Issue:

Attribute Grammars and Their Applications

Guest Editors:

Marjan Mernik, Didier Parigot

Informatica **24** (2000) Number 3, pp. 285–428



The Slovene Society Informatika, Ljubljana, Slovenia

Informatica

An International Journal of Computing and Informatics

Archive of abstracts may be accessed at USA: http://, Europe: http://ai.ijs.si/informatica, Asia: http://www.comp.nus.edu.sg/ liuh/Informatica/index.html.

Subscription Information Informatica (ISSN 0350-5596) is published four times a year in Spring, Summer, Autumn, and Winter (4 issues per year) by the Slovene Society Informatika, Vožarski pot 12, 1000 Ljubljana, Slovenia.
The subscription rate for 2000 (Volume 24) is
– DEM 100 (US$ 70) for institutions,
– DEM 50 (US$ 34) for individuals, and
– DEM 20 (US$ 14) for students
plus the mail charge DEM 10 (US$ 7).
Claims for missing issues will be honored free of charge within six months after the publication date of the issue.

LaTeX Tech. Support: Borut Žnidar, Kranj, Slovenia.
Lectorship: Fergus F. Smith, AMIDAS d.o.o., Cankarjevo nabrežje 11, Ljubljana, Slovenia.
Printed by Biro M, d.o.o., Žibertova 1, 1000 Ljubljana, Slovenia.

Orders for subscription may be placed by telephone or fax using any major credit card. Please call Mr. R. Murn, Jožef Stefan Institute: Tel (+386) 1 4773 900, Fax (+386) 1 219 385, or send checks or VISA card number or use the bank account number 900–27620–5159/4 Nova Ljubljanska Banka d.d. Slovenia (LB 50101-678-51841 for domestic subscribers only).

Informatica is published in cooperation with the following societies (and contact persons):
Robotics Society of Slovenia (Jadran Lenarčič)
Slovene Society for Pattern Recognition (Franjo Pernuš)
Slovenian Artificial Intelligence Society; Cognitive Science Society (Matjaž Gams)
Slovenian Society of Mathematicians, Physicists and Astronomers (Bojan Mohar)
Automatic Control Society of Slovenia (Borut Zupančič)
Slovenian Association of Technical and Natural Sciences / Engineering Academy of Slovenia (Janez Peklenik)

Informatica is surveyed by: AI and Robotic Abstracts, AI References, ACM Computing Surveys, ACM Digital Library, Applied Science & Techn. Index, COMPENDEX*PLUS, Computer ASAP, Computer Literature Index, Cur. Cont. & Comp. & Math. Sear., Current Mathematical Publications, Engineering Index, INSPEC, Mathematical Reviews, MathSci, Sociological Abstracts, Uncover, Zentralblatt für Mathematik, Linguistics and Language Behaviour Abstracts, Cybernetica Newsletter

The issuing of the Informatica journal is financially supported by the Ministry for Science and Technology, Slovenska 50, 1000 Ljubljana, Slovenia.

Post tax payed at post 1102 Ljubljana. Slovenia taxe Percue.

Introduction: Attribute Grammars and Their Applications

Attribute grammars are formalism for specifying the syntax and the static semantics of programming languages, as well as for implementing syntax-directed editors, compilers/interpreters, debuggers and compiler/interpreter generators. Attribute grammars have become one of the most fundamental formalism of modern Computer Science. Since 1968, when Knuth [5] introduced the basic concepts, more than 1100 references [2] on theoretical aspects, applications and systems have appeared, proving the intensive research and importance of the area [3, 6, 7]. Research on attribute grammars in the first 15 years, when theoretical concepts (S-attributed grammars, L-attributed grammars, absolutely noncircular attribute grammars, ordered attribute grammars) and basic implementations (FOLDS, GAG, LINGUIST-86, HLP-84) were developed, moved to more pragmatic issues in recent years. Recently, there has been a lot of research work on augmenting ordinary attribute grammars with extensions to overcome the deficiencies of attribute grammars, such as lack of modularity, extensibility and reusability. Several concepts, such as remote attribute access, object-orientation, templates, rule models, symbol computations, high order features etc., have been implemented in various attribute grammar specification languages [3]. The implementation of programming languages is the original and the most widely recognized area of attribute grammars, but there are many other areas where they are used: software engineering, static analysis of programs, natural language processing, graphical user interfaces, communication protocols, databases, pattern recognition, hardware design, rapid prototyping domain-specific languages, web computing, e-commerce, etc.

The aim of workshops WAGA'99 and WAGA'00 was to bring together researchers from academia and industry interested in the field of attribute grammars. Workshops covered all aspects of attribute grammars, with special emphasis on new applications of attribute grammars and comparisons to other formalisms and to programming languages. WAGA'99 was held on March 26th, 1999 in Amsterdam, The Netherlands, as a satellite event of ETAPS'99, European Joint Conferences on Theory and Practice of Software. WAGA'00 was held on July 7th, 2000 in Ponte de Lima, Portugal, as a satellite event of MPC'2000, the 5th International Conference on Mathematics of Program Construction.

This special issue on *Attribute Grammars and their Applications* contains 6 papers, which have been selected from 37 submissions, of which 21 were accepted for presentation at WAGA'99 [7] and WAGA'00 [8]. These papers are extensively revised versions of original presentations published in WAGA proceedings [7, 8].

The first paper presents a new structure-oriented denotational semantics of attribute grammars where the attributed tree is presented by nested records. Katsuhiko Gondow and Takuya Katayama, in *Attribute Grammars as Record Calculus - A Structure-Oriented Denotational Semantics of Attribute Grammars by Using Cardelli's Record Calculus*, describe the theoretical framework for modeling attribute grammar extensions, such as higher-order attribute grammars, recursive attribute grammars and object-oriented attribute grammars. The new formalism is implemented using SML/NJ. Next two papers describe an object-oriented extension to canonical attribute grammars. Gorel Hedin, in *Reference Attributed Grammars*, introduces reference semantics to attribute grammars where attributes are allowed to be references to nodes in the syntax tree. Important practical problems, such as name and type analysis, inheritance, qualified use, and assignment compatibility in the presence of subtyping, can be expressed in a concise and modular manner in these grammars. The formalism and efficient algorithm have been implemented in APPLAB, an interactive language development tool. The next paper is focused on incremental language design. Marjan Mernik, Mitja Lenič, Enis Avdičaušević and Viljem Žumer, in *Multiple Attribute Grammar Inheritance*, introduce a new object-oriented attribute grammar specification language where specifications can be developed incrementally with multiple attribute grammar inheritance. Multiple attribute grammar inheritance is a structural organization of attribute grammars where the attribute grammar inherits the specifications from ancestor attribute grammars, may add new specifications or may override some specifications from ancestor specifications. The approach is successfully implemented in the compiler/interpreter generator tool LISA ver. 2.0. Modular descriptions of attribute grammar specification languages is a topic of the next paper. Oege de Moor, Kevin Backhouse and Doaitse Swierstra in *First-class Attribute Grammars*, presented a semantic view of attribute grammars, embedded as first-class values in the lazy functional programming language Haskell. In the next paper the importance of attribute grammars to functional programming is presented. Loïc Correnson, in *Equational Semantics*, continues his work on symbolic composition where the deforestation method provides a better deforestation method than other existing functional techniques. The equational program is a set of properties that rely on attributes and are especially dedicated to program transformations, such as partial evaluation, reduction, specialization, deforestation and elimination of identity. One drawback of attribute grammars is also that non-linear algorithms can not be expressed. However, this is not true for Equational Semantics, a formalism largely inspired by attribute grammars, but where non-linear algorithms can be encoded. In some sense, Equational Semantics is a kind of lambda-calculus dedicated to program transformations. In the final paper *Two-dimensional Approximation Coverage*, Jörg Harm and Ralf Lämmel present fundamentals for attribute grammar testing. Developing, extending and tuning real-world attribute grammar specifications are non-

trivial tasks. Automatic generation and application of test cases are then of great help to the language developer. The proposed approach is also applicable to first-order declarative programs, such as logic programs and constructive algebraic specifications.

In conclusion, we hope the papers in this special issue will provide readers with the glimpse of current research trends in attribute grammars. Also, we wish to sincerely thank the Program Committee for their assistance in the reviewing process.

References

[1] Knuth D. E. Semantics of contex-free languages. Math. Syst. Theory, Vol. 2, No. 2, pp. 127 - 145, 1968.

[2] Attribute Grammars Home Page. http://www-sop.inria.fr/oasis/Didier.Parigot/www/fnc2/attribute-grammar.html

[3] Deransart P., Jourdan M., Lorho B. Attribute Grammars: Definitions, Systems and Bibliography. Lecture Notes in Computer Science Vol. 323, Springer - Verlag, 1988.

[4] Deransart P., Jourdan M. (Eds.) Attribute Grammars and their Applications. Proceedings of 1st WAGA, Lecture Notes in Computer Science Vol. 461, Springer - Verlag, 1990.

[5] Alblas H., Melichar B. (Eds.) Attribute Grammars, Applications, and Systems. Proceedings of SAGA, Lecture Notes in Computer Science Vol. 545, Springer - Verlag. 1991.

[6] Paakki J. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. ACM Computing Surveys, Vol. 27, No. 2, pp. 196 - 255, 1995.

[7] D. Parigot, M. Mernik (Eds.) Attribute Grammars and their Applications. Proceedings of 2nd WAGA, INRIA, 1999.

[8] D. Parigot, M. Mernik (Eds.) Attribute Grammars and their Applications. Proceedings of 3rd WAGA, INRIA, 2000.

Marjan Mernik, Didier Parigot

Attribute Grammars as Record Calculus — A Structure-Oriented Denotational Semantics of Attribute Grammars by Using Cardelli's Record Calculus

Katsuhiko Gondow
Dept. of Information Science, Japan Advanced Inst. of Science and Technology,
1-1 Asahidai Tatsunokuchi Nomi Ishikawa, 923-1292, Japan
Phone: +81 761 51 1261, Fax: +81 761 51 1360
E-mail: gondow@jaist.ac.jp
AND
Takuya Katayama
Dept. of Information Science, Japan Advanced Inst. of Science and Technology,
1-1 Asahidai Tatsunokuchi Nomi Ishikawa, 923-1292, Japan
Phone: +81 761 51 1250, Fax: +81 761 51 1360
E-mail: katayama@jaist.ac.jp

Keywords: attribute grammars, Cardelli's record calculus, structure-oriented denotational semantics

Edited by: Marjan Mernik and Didier Parigot
Received: July 27, 2000 **Revised:** August 28, 2000 **Accepted:** September 11, 2000

1 Introduction

In this paper, we present a new denotational semantics of attribute grammars (AGs) [15] [16] [6] based on Cardelli's record calculus [5][1]. This semantics is structure-oriented as well as natural and simple. Unlike previous works, an attributed tree is represented as a nested record to preserve its structural information.

AGs[15][16][6] are a formal system for specifying semantics of programming languages, and many compiler generators are studied and implemented [7][14]. Since the latter half of 1980s, however, syntax-directed editors based on AGs have been considered useful in describing and generating interactive programming environments [21]. Declarative structures, separation of semantics and syntax definition, local description resulting in high readability and high maintainability, and clear description caused by functional computation of attributes are the positive characteristics of AGs.

Using AGs, interactive programming environments are often described as attributed trees with several AG extensions, e.g., higher-order AGs (HAGs)[26][23], subtree replacement in the Synthesizer Generator[10] and in object-oriented AGs(OOAG)[22][9], recursive AGs(RAGs)[8], and remote access[10][16][12]. Unfortunately, it was not easy to compare various definitions for these extensions in a formal way. One of the reasons is that previous studies(e.g., [24][13]) for AG semantics are not structure-oriented, that is, they are based on attribute valuation, not an attributed tree itself. For example, AG semantics based on attribute valuation can not deal directly with program transformation such as $a \times (b+c) \Rightarrow a \times b + a \times c$, since it focuses only on attribute values, not on the structure of an attributed tree.

In [24], Takeda and Katayama defined a semantics of AGs as a sequence of all attribute values in an attributed tree. In [13], Johnsson defined it as a collection of functions to compute values of synthesized attributes. These semantics are essentially based on *attribute valuation*, not *an attributed tree* itself. Thus, these formal semantics lack the structural information in AGs, so they do not suit to formalize structure-oriented aspects of OOAG, HAGs, and so on.

Fig.1 shows the overview of the new semantics. A derivation tree is represented as a term like $\mathsf{p}_1(\mathsf{p}_2, \mathsf{p}_3)$, and an attributed tree is represented as a nested record like $\langle \mathrm{a} = 1, \mathrm{X} = \langle \mathrm{a} = 1 \rangle \rangle$ where, for example, a denotes an attribute and X denotes a nonterminal. A production rule p and its semantic rules $R(p)$ are translated into a function $p_{\mathcal{E}}$. In other words, an AG is represented as a set of $p_{\mathcal{E}}$. The semantic function $\mathcal{E}$ is the key of our AG semantics; the semantic function $\mathcal{E}$ corresponds to an attribute evaluator. The definition of $\mathcal{E}$ is defined as simple recursion on tree structures in Def.3.6 as follows.

$$\mathcal{E}[\![\mathsf{p}(t_1, \ldots, t_n)]\!] = \mu \mathrm{self}.p_{\mathcal{E}}(\mathcal{E}[\![t_1]\!], \ldots, \mathcal{E}[\![t_n]\!], \mathrm{self})$$

where $t_1, \ldots, t_n$ are subtrees of the derivation tree, and μ is a fixed-point operator.

We think the semantics is a good theoretical groundwork for modeling AG extensions (especially structure-oriented

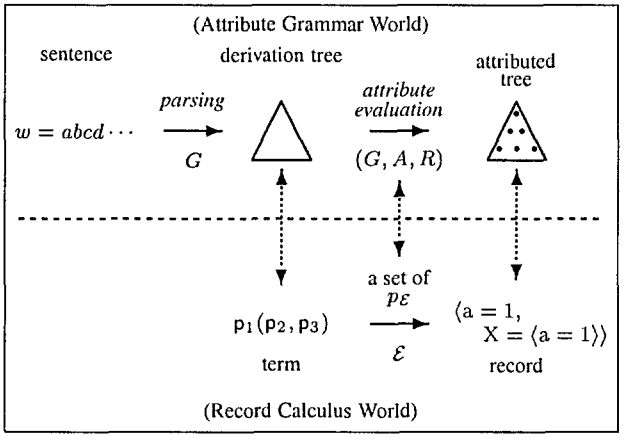


Figure 1: Relations among Attribute Grammars, Attribute Evaluator, Semantic Function $\mathcal{E}$, and Function $p_{\mathcal{E}}$

ones). To show it, we also represent HAGs, RAGs and OOAG as record calculus.

The rest of the paper is organized as follows. In the next section, Section 2, we present a brief background of AGs and Cardelli's record calculus. Section 3 provides the new AG semantics by using record calculus. In Section 4, we represent AG extensions as record calculus. Section 5 gives a simple implementation using SML/NJ[25]. Section 6 summarizes this paper and future works are presented in Section 7.

2 Backgrounds

2.1 Definition of Attribute Grammars

This section provides a tuple-style definition of AGs and some terminology.

An AG is defined by a 3-tuple $AG = (G, A, R)$, where G is an underlying context free grammar, A is a finite set of *attributes* and R is a finite set of *semantic rules*. A context free grammar is defined by a 4-tuple $G = (N, T, S, P)$, where N is a finite set of *nonterminals*, T is a finite set of *terminals*, $S \in N$ is a start symbol, and P is a finite set of production rules.

Each nonterminal is associated with two disjoint finite set $Inh(X)$ and $Syn(X)$, where $Inh(X) \cap Syn(X) = \phi, A = \bigcup_{X \in N}(Inh(X) \cup Syn(X))$. An element of $Inh(X)$ is called an *inherited attribute*, and that of $Syn(X)$ is called a *synthesized attribute*.

For a production rule $p : X_0 \to X_1 \cdots X_n$, we call an attribute a of X_i occurring in the semantic rules in $R(p)$ an *attribute occurrence*, which is written as $X_i \cdot a$, where $0 \le i \le n$ and $a \in Inh(X_i) \cup Syn(X_i)$[1]. The following $Occur(p)$ is a set of attribute occurrences that may occur in p.

$$Occur(p) = \{X_i \cdot a | (0 \le i \le n) \wedge (a \in Inh(X_i) \cup Syn(X_i))\}$$

A set $R(p)$ of semantic rules associated with a production rule $p : X_0 \to X_1 \cdots X_n$ is defined:

$$R(p) = \{X_i \cdot a = e | (i = 0 \wedge a \in Syn(X_0)) \vee (1 \le i \le n \wedge a \in Inh(X_i)), e \in Exp(Occur(p))\}$$

where $R = \bigcup_{p \in P} R(p)$, and $Exp(Occur(p))$ is a set of terms constructed by attribute occurrences $Occur(p)$ and function symbols.

For a given derivation tree on an AG, the denotational semantics of the AG is the attributed tree where all attribute values on the derivation tree satisfy their associated semantics rules. In other words, the semantic function of an AG is a mapping from any derivation tree to the attributed tree where all values of attributes are consistent.

A condition "attribute dependencies on any derivation tree is cycle free" is a sufficient condition (not a necessary one) to be able to compute all attribute values on the derivation tree. AGs that hold this condition are called *non-circular* AGs. This paper does not suppose non-circular AGs. A wider class of recursive AGs introduced by Farrow[8] is represented as records in Def.4.2.

2.1.1 An Example AG

Example 2.1 (AG_1: **n-radix numerals**)
$AG_1 = (G, A, R)$ is defined as follows.

$$\begin{aligned} N &= \{\mathrm{N}, \mathrm{I}, \mathrm{D}\} \\ T &= \{0, 1, \ldots, 9\} \\ S &= \mathrm{N} \\ P &= \{p_{\mathrm{N}} : \mathrm{N} \to \mathrm{I}, p_{\mathrm{I1}} : \mathrm{I} \to \mathrm{I\ D}, p_{\mathrm{I2}} : \mathrm{I} \to \mathrm{D}, \\ &\quad p_{\mathrm{D0}} : \mathrm{D} \to 0, \ \ldots\ , p_{\mathrm{D9}} : \mathrm{D} \to 9\} \end{aligned}$$

$$\begin{aligned} Inh(\mathrm{N}) &= \{\mathrm{radix}\} \\ Inh(\mathrm{I}) &= Inh(\mathrm{D}) = \{\mathrm{scale}, \mathrm{radix}\} \\ Syn(\mathrm{N}) &= Syn(\mathrm{I}) = Syn(\mathrm{D}) = \{\mathrm{val}\} \\ R(p_{\mathrm{N}}) &= \{\mathrm{I} \cdot \mathrm{scale} = 0, \\ &\quad \mathrm{I} \cdot \mathrm{radix} = \mathrm{N} \cdot \mathrm{radix}, \\ &\quad \mathrm{N} \cdot \mathrm{val} = \mathrm{I} \cdot \mathrm{val}\ \} \\ R(p_{\mathrm{I1}}) &= \{\mathrm{I}_2 \cdot \mathrm{scale} = \mathrm{I}_1 \cdot \mathrm{scale} + 1, \\ &\quad \mathrm{D} \cdot \mathrm{scale} = \mathrm{I}_1 \cdot \mathrm{scale}, \\ &\quad \mathrm{I}_2 \cdot \mathrm{radix} = \mathrm{I}_1 \cdot \mathrm{radix}, \\ &\quad \mathrm{D} \cdot \mathrm{radix} = \mathrm{I}_1 \cdot \mathrm{radix}, \\ &\quad \mathrm{I}_1 \cdot \mathrm{val} = \mathrm{I}_2 \cdot \mathrm{val} + \mathrm{D} \cdot \mathrm{val}\} \\ R(p_{\mathrm{I2}}) &= \{\mathrm{D} \cdot \mathrm{scale} = \mathrm{I} \cdot \mathrm{scale}, \\ &\quad \mathrm{D} \cdot \mathrm{radix} = \mathrm{I} \cdot \mathrm{radix}, \\ &\quad \mathrm{I} \cdot \mathrm{val} = \mathrm{D} \cdot \mathrm{val}\} \\ R(p_{\mathrm{D}i}) &= \{\mathrm{D} \cdot \mathrm{val} = \\ &\quad i \times \mathrm{D} \cdot \mathrm{radix} \uparrow \mathrm{D} \cdot \mathrm{scale}\} \end{aligned}$$

We assume that the value of inherited attribute N · radix is given a priori. Symbols $+$, $\times$, and $\uparrow$ are infix operators

[1] We use ' · ' to distinguish an attribute occurrence from record selecting operator ' . ' given in Section 2.2.

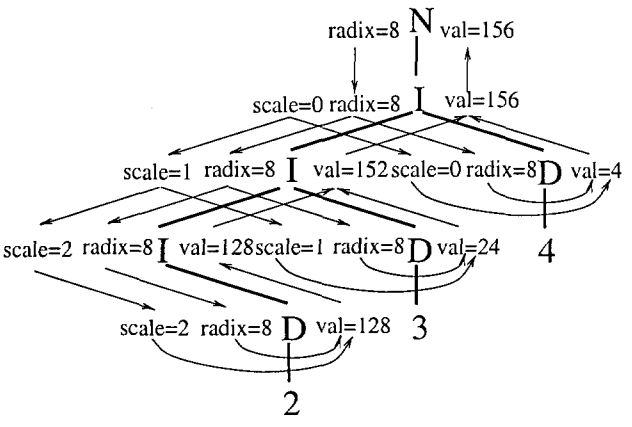


Figure 2: An Attributed Tree where $w = 234$, N.radix $= 8$

for addition, multiplication, and power respectively. To distinguish between different occurrences of the same nonterminal I in a production, indexing is used like I_1 and I_2. $R(p_{\mathrm{D}i}) = \{\cdots\}$ is an abbreviation for each definition of $R(p_{\mathrm{D}0}) \cdots R(p_{\mathrm{D}9})$.

□

Readers can understand AG_1 intuitively as follows. Let the value of N·radix be n, then the value of N·val indicates the value of $w \in L(G)$ as n-radix numeral. Fig.2 shows attributed trees where $w = 234$ and N · radix $= 8$. Arrows in Fig.2 indicate dependencies between attribute instances ($a \rightarrow b$ represents "b depends on a").

Any traditional semantics is enough when you need only the value of N · val, but not when you want to handle attribute trees as databases (or interactive programming environments). For example, you may want to search leaves whose values are 3 and change them to 0. The traditional semantics can not deal with such cases. Therefore we need a new semantics that handle not only attribute values but also attributed trees.

2.2 Definition of Cardelli's Record Calculus

In this section, we briefly explain a record and its calculus introduced by Cardelli[5][1]. We do not explain a record type, since it is beyond the scope of this paper to consider the aspects of record types. A Cardelli's record is a finite mapping from labels to values. A record is written[2]:

$$\langle l_1 = v_1, \ldots, l_n = v_n \rangle$$

where $l_1, \ldots, l_n$ are labels, and $v_1, \ldots, v_n$ are associated values, respectively. Each $l_i = v_i$ $(1 \leq i \leq n)$ is called a field. The following is an example record.

$$\langle a = 1, b = \text{false} \rangle$$

[2]In [5][1], symbols '{' and '}' are used as record constructors. But, to distinguish between a record and a set explicitly, we use '⟨' and '⟩' for record instead of '{' and '}'.

Selecting the value associated with a label l of a record r is given by $r.l$. Therefore, the value of the following expression is 1.

$$\langle a = 1, b = \text{false} \rangle.a$$

We assume the selecting operator is left associative, so we abbreviate $(\cdots((r.l_1).l_2)\cdots.l_n)$ to $r.l_1.l_2.\cdots.l_n$, where r is a record and $l_1, l_2, \ldots, l_n$ are labels. For example, the value of following expression is false.

$$\langle a = \langle a = 1, b = \text{false} \rangle, b = \text{true} \rangle.a.b$$

A record calculus used in this paper is λ-calculus with the following reduction rule $(1 \leq i \leq n)$.

$$\langle l_1 = v_1, \ldots, l_n = v_n \rangle.l_i \Rightarrow v_i$$

2.3 Objects and Classes as Records

Many studies have been done to formalize and discuss the various concepts in object-oriented programming such as objects, classes, encapsulation and inheritance by using record calculus (e.g., [1][4][2][3]). This section gives a brief review for objects and classes represented as records.

- *Objects* are represented as records whose fields are methods and instance variables. For example, an object *point* has two instance variables x and y and one method *dist*.

$$point = \langle x = 10, y = 20, dist = sqrt(x^2 + y^2) \rangle$$

Note that the method *dist* refers to labels x and y in *point*. Thus, the methods of an object may refer to each other. To eliminate references to labels x and y, a binding variable *self* and a fixed-point operator μ are traditionally used; the variable *self* corresponds to pseudo-variable self in object-oriented programming languages.

$$\begin{aligned} & point \\ = \; & \langle x = 10, y = 20, \\ & \quad dist = sqrt(point.x^2 + point.y^2) \rangle \\ = \; & (\lambda self.\langle x = 10, y = 20, \\ & \quad dist = sqrt(self.x^2 + self.y^2) \rangle)(point) \\ = \; & \mu self.\langle x = 10, y = 20, \\ & \quad dist = sqrt(self.x^2 + self.y^2) \rangle \end{aligned}$$

where μ is a fixed-point operator. A fixed-point of f (that is, x that satisfies $x = f(x)$) is represented as $\mu x.f(x)$. The fixed-point operator has the unrolling rule:

$$\mu x.f(x) = f(\mu x.f(x))$$

For readability, we also use a fixed-point combinator $\mathbf{Y}$.

$$\mu x.f(x) = \mathbf{Y}(f)$$

The combinator $\mathbf{Y}$ has the following reduction rule.

$$\mathbf{Y}(f) \Rightarrow f(\mathbf{Y}(f))$$

- *Classes* are parameterized objects (i.e. functions) that return object records.

$$\begin{aligned} &pointclass \\ =\ &\lambda i_x.\lambda i_y.\mu self.\langle x = i_x, y = i_y, \\ &dist = sqrt(self.x^2 + self.y^2)\rangle \end{aligned}$$

The methods of an object may refer to any class to create new instances, so a class may be (mutual) recursively defined. In the following example, we use *myclass* and μ to eliminate a recursive occurrence of *pointclass*.

$$\begin{aligned} &pointclass \\ =\ &\lambda i_x.\lambda i_y.\mu self.\langle x = i_x, y = i_y, \\ &dist = sqrt(self.x^2 + self.y^2), \\ &move = \lambda dx.\lambda dy.pointclass \\ &\qquad (self.x + dx)(self.y + dy)\rangle \\ =\ &\mu myclass.\lambda i_x.\lambda i_y.\mu self.\langle x = i_x, y = i_y, \\ &dist = sqrt(self.x^2 + self.y^2), \\ &move = \lambda dx.\lambda dy.myclass \\ &\qquad (self.x + dx)(self.y + dy)\rangle \end{aligned}$$

3 Formalizing AGs by Using Record Calculus

In this section, we define a new denotational semantics of AGs by using record calculus.

First, an attributed tree is represented as a nested record (Section 3.1). In Section 3.2, we introduce a function $p_{\mathcal{E}}$, which corresponds to a production rule p and a set of semantic rules $R(p)$. Here we use a term as a linear notation to express a derivation tree. In Section 3.3, we define the semantic function $\mathcal{E}$. We give an example to show a process of computing attributed trees from derivation trees (Section 3.4).

3.1 Record Representation for Attributed trees

In this section, we show how to represent an attributed tree as records.

Each node in an attributed tree has:

- attributes and their values
- subtrees

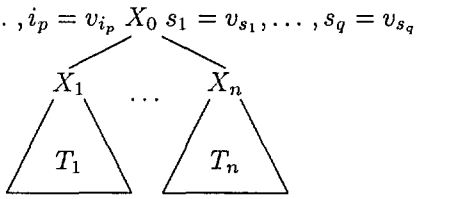


Figure 3: An Attributed Tree: T_0

Therefore, it is natural to regard an attributed tree as a record which has fields for both attributes and subtrees.

Definition 3.1 (Record Representation for an Attributed Tree)

Let the top production rule of an attributed tree be $p : X_0 \rightarrow X_1 \cdots X_n$. The record representation for an attributed tree is defined by the record that has the following fields:

1. fields whose labels are X_i $(1 \leq i \leq n)$, and whose values are records that represents attributed subtrees rooted in X_i $(1 \leq i \leq n)$, respectively
2. fields whose labels are attributes $a \in Inh(X_0) \cup Syn(X_0)$, and whose values are values of attribute instances, respectively

□

To illustrate Def.3.1, consider an attributed tree in Fig.3. An attributed tree T_0 in Fig.3 is represented as the following record by Def.3.1:

$$\begin{aligned} T_0 = \langle\ & i_1 = v_{i_1}, \ldots, i_p = v_{i_p}, \\ & s_1 = v_{s_1}, \ldots, s_q = v_{s_q}, \\ & X_1 = T_1, \ldots, X_n = T_n\ \rangle \end{aligned}$$

where $i_j (1 \leq j \leq p)$ is an instance of an inherited attribute in $Inh(X_0)$, $s_k (1 \leq k \leq q)$ is an instance of a synthesized attribute in $Syn(X_0)$, v_a is the value of an attribute instance a, and $T_l (1 \leq l \leq n)$ is a record which represents an attributed subtree rooted in X_l.

For example, the following record represents an attributed tree N in Fig.2.

$$\begin{aligned} &\mathrm{N} = \langle \mathrm{radix} = 8, \mathrm{val} = 156, \\ &\quad \mathrm{I} = \langle \mathrm{scale} = 0, \mathrm{radix} = 8, \mathrm{val} = 156, \\ &\quad\quad \mathrm{I}_2 = \langle \mathrm{scale} = 1, \mathrm{radix} = 8, \mathrm{val} = 152, \\ &\quad\quad\quad \mathrm{I}_2 = \langle \mathrm{scale} = 2, \mathrm{radix} = 8, \mathrm{val} = 128, \\ &\quad\quad\quad\quad \mathrm{D} = \langle \mathrm{scale} = 2, \mathrm{radix} = 8, \mathrm{val} = 128\rangle\rangle, \\ &\quad\quad\quad \mathrm{D} = \langle \mathrm{scale} = 1, \mathrm{radix} = 8, \mathrm{val} = 24\rangle\rangle, \\ &\quad\quad \mathrm{D} = \langle \mathrm{scale} = 0, \mathrm{radix} = 8, \mathrm{val} = 4\rangle\rangle\rangle \end{aligned}$$

3.2 Translating from an AG to a Set of Functions $p_{\mathcal{E}}$

In this section, we define a function $p_{\mathcal{E}}$, which we need later in Section 3.3 to define the semantic function $\mathcal{E}$. A function $p_{\mathcal{E}}$ has the following information.

1. a production rule $p : X_0 \to X_1 \cdots X_n$
2. a set of semantic rules $R(p)$
3. a set of attributes $Syn(X_0) \cup Inh(X_0)$

Roughly speaking, input of $p_{\mathcal{E}}$ is (1) attributed subtrees λ-abstracted with inherited attributes on their roots ('child$_i$' in Def.3.2), and (2) an attributed tree λ-abstracted with inherited attributes on its root (a 'self' in Def.3.2), which is equal to $p_{\mathcal{E}}$'s output. Output (=self) of $p_{\mathcal{E}}$ has the top production rule p on its root. Thus, an attributed tree 'self' is recursively defined by $p_{\mathcal{E}}$, attributed subtrees 'child$_1$',...,'child$_n$', and 'self' itself as follows.

$$\text{self} = p_{\mathcal{E}}(\text{child}_1, \ldots, \text{child}_n, \text{self})$$

(See Def.3.2,3.6 for the formal definitions.)

Definition 3.2 (Function $p_{\mathcal{E}}$)
For a production rule $p : X_0 \to X_1 \cdots X_n$, let $Inh(X_j), Syn(X_j)$ $(0 \le j \le n)$ and $R(p)$ be as follows.

$$\begin{aligned} Inh(X_j) &= \{i_{j,1}, \ldots, i_{j,p_j}\} \\ Syn(X_j) &= \{s_{j,1}, \ldots, s_{j,q_j}\} \end{aligned}$$

$$\begin{aligned} R(p) = \{ & \\ & X_0 \cdot s_{0,1} = e_{0,1}, \ldots, X_0 \cdot s_{0,q_0} = e_{0,q_0}, \\ & X_1 \cdot i_{1,1} = e_{1,1}, \ldots, X_1 \cdot i_{1,p_1} = e_{1,p_1}, \\ & \qquad \vdots \\ & X_n \cdot i_{n,1} = e_{n,1}, \ldots, X_n \cdot i_{n,p_n} = e_{n,p_n} \\ \} & \end{aligned}$$

Then, $p_{\mathcal{E}}$ is defined as follows,

$$\begin{aligned} p_{\mathcal{E}} = & \\ & \lambda\text{child}_1 \cdots \lambda\text{child}_n.\lambda\text{self}.\lambda\text{inh}_1 \cdots \lambda\text{inh}_{p_0}. \\ & \langle \\ & \quad i_{0,1} = \text{inh}_1, \ldots, i_{0,p_0} = \text{inh}_{p_0}, \\ & \quad s_{0,1} = e'_{0,1}, \ldots, s_{0,q_0} = e'_{0,q_0}, \\ & \quad X_1 = \text{child}_1(e'_{1,1}, \ldots, e'_{1,p_1}), \\ & \quad \vdots \\ & \quad X_n = \text{child}_n(e'_{n,1}, \ldots, e'_{n,p_n}) \\ & \rangle \end{aligned}$$

where for any j, k $((j = 0 \wedge 1 \le k \le q_0) \vee (1 \le j \le n \wedge 1 \le k \le p_j))$, $e'_{j,k}$ is a term where all attribute occurrences α are replaced with $\alpha_{\mathcal{E}}$ in $e_{j,k}$(each right-hand side of a production rule p). A term $\alpha_{\mathcal{E}}$ is defined as follows.

$$\alpha_{\mathcal{E}} = \begin{cases} \text{self}(\text{inh}_1, \ldots, \text{inh}_{p_0}).X_i.a \\ \qquad (\textbf{if } \alpha = X_i \cdot a \wedge 1 \le i \le n) \\ \text{self}(\text{inh}_1, \ldots, \text{inh}_{p_0}).a \\ \qquad (\textbf{if } \alpha = X_0 \cdot a) \end{cases}$$

□

In Def.3.2, the reason why 'self' appears recursively in the body of $p_{\mathcal{E}}$ as its argument is:

1. $p_{\mathcal{E}}$ is a constructor i.e. a mapping from attributed subtrees 'child$_i$' to an attributed tree 'self', that is, 'self' depends on 'child$_i$'.
2. Inherited attributes in 'child$_i$' may depend on inherited attributes in 'self', that is, 'child$_i$' depends on 'self'.
3. From (1) and (2), 'self' depends on 'self' itself, so 'self' needs to be defined recursively.

If non-circular AGs are supposed, it is, of course, possible to define AG semantics without μ operator nor other recursive directives like **letrec**, but the definition would be so complicated; using μ operator helps to make the definition simple.

The definition of $\alpha_{\mathcal{E}}$ needs to be divided into two cases in Def.3.2, since "$X_0 \cdot a$ is an attribute *of* 'self', but on the other hand $X_i \cdot a (1 \le i \le n)$ is an attribute *of* 'child$_i$' *of* 'self' ", where *of* corresponds to the record selector '.'.

For function symbols 'self' and 'child$_i$' in Def.3.2, a notation $f(x)$ represents an application of function f to an argument x, and $f(x_1)(x_2)\ldots(x_n)$ is abbreviated to $f(x_1, \ldots, x_n)$. And a nullary function $f()$ will be also abbreviated to f.

3.3 Definition of the Semantic Function $\mathcal{E}$

This section presents the definition of the semantic function $\mathcal{E}$ by using $p_{\mathcal{E}}$ defined in Section 3.2. The semantic function defined here is a mapping from a set of derivation trees $\mathbb{T}$ to a set of attributed trees $\mathbb{R}$. More precisely, $\mathbb{T}$ is a set of terms which represents derivation trees (Def.3.3), and $\mathbb{R}$ is a set of records which represent attributed trees as semantics of the AG.

Before defining $\mathcal{E}$, we provide a term representation for an attributed tree and a definition of $\mathbb{T}$, then we define semantic function $\mathcal{E}$.

Definition 3.3 (Term Representation for an Attributed Tree)
We define a set of sorts N and a set of operators P associated with a set of nonterminals N and a set of production rules P, respectively. For each production rule $p : X_0 \to X_1 \cdots X_n \in P$, we also define its arity and sort as follows.

$$\begin{aligned} \text{arity(p)} &= \mathsf{X}_1\mathsf{X}_2 \cdots \mathsf{X}_n \\ \text{sort(p)} &= \mathsf{X}_0 \end{aligned}$$

In other words, we regard a production rule p as a function symbol p typed:

$$\mathsf{p} : \mathsf{X}_1, \ldots, \mathsf{X}_n \mapsto \mathsf{X}_0$$

Let $X_0 \rightarrow X_1 \cdots X_n$ be a production rule at the root of a derivation tree T_0, and let $t_i (1 \leq i \leq n)$ be a term which represents a derivation subtree rooted in X_i. Then, a term that represents a derivation tree T_0 is defined by $\mathsf{p}(t_1, \ldots, t_n)$.

□

For example, a term that represents a derivation tree in Fig.2 produced by the AG of Example 2.1 is as follows.

$$\mathsf{p_N}(\mathsf{p_{I1}}(\mathsf{p_{I1}}(\mathsf{p_{I2}}(\mathsf{p_{D2}}), \mathsf{p_{D3}}), \mathsf{p_{D4}}))$$

Definition 3.4 (Term Set of Derivation Trees)
We define recursively $\mathcal{T}_{\mathsf{X}_0}(\Sigma_G, \phi)$ (a set of ground terms of sort X_0) produced by a signature $\Sigma_G = (\mathsf{N}, \mathsf{P})$ associated with a grammar G as follows.

1. If arity(p)=ϕ and sort(p)=X_0, then $\mathsf{p} \in \mathcal{T}_{\mathsf{X}_0}(\Sigma_G, \phi)$
2. If arity(p)=$\mathsf{X}_1 \ldots \mathsf{X}_n$,sort(p)=$\mathsf{X}_0$ and $t_i \in \mathcal{T}_{\mathsf{X}_i}(\Sigma_G, \phi)$ where $1 \leq i \leq n$, then $\mathsf{p}(t_1, \ldots, t_n) \in \mathcal{T}_{\mathsf{X}_0}(\Sigma_G, \phi)$

□

$\mathcal{T}_{\mathsf{X}_0}(\Sigma_G, \phi)$ is a set of terms that represents derivation trees whose roots are labeled by X_0. Here ϕ means that $\mathcal{T}_{\mathsf{X}_0}(\Sigma_G, \phi)$ has no variables.

Definition 3.5 ($\mathbb{T}$: Term Set of Grammar G)
A term set $\mathbb{T}$ of derivation trees produced by grammar $G = (N, T, S, P)$ is defined as follows.

$$\mathbb{T} = \bigcup_{\mathsf{X} \in \mathsf{N}} \mathcal{T}_{\mathsf{X}}(\Sigma_G, \phi)$$

□

Note that $\mathbb{T}$ includes not only $\mathcal{T}_{\mathsf{S}}(\Sigma_G, \phi)$ (a term set for derivation trees whose roots are labeled with the start symbol S), but also $\mathcal{T}_{\mathsf{X}}(\Sigma_G, \phi)$ whose roots are labeled with any other nonterminals $X \in N$.

Def.3.3,3.4 and 3.5 were originally defined by Vogt et al. in [26], and we slightly modified them to fit our notations.

Definition 3.6 (A Semantic Function of AGs: $\mathcal{E}$)
A semantic function $\mathcal{E} : \mathbb{T} \rightarrow \mathbb{R}$ is defined here, which is a mapping from a set of terms that represents derivation trees to a set of records that represents attributed trees.

For a production rule $p : X_0 \rightarrow X_1 \cdots X_n$, and $t_i \in \mathcal{T}_{\mathsf{X}_i}(\Sigma_G, \phi)$ where $1 \leq i \leq n$, the semantics of a derivation tree $\mathsf{p}(t_1, \ldots, t_n)$ is defined by using the semantic function $\mathcal{E}$ as follows:

$$\mathcal{E}[\![\mathsf{p}(t_1, \ldots, t_n)]\!] = \mu\mathsf{self}.p_{\mathcal{E}}(\mathcal{E}[\![t_1]\!], \ldots, \mathcal{E}[\![t_n]\!], \mathsf{self})$$

□

Here we illustrate the meaning of Def.3.6. Def.3.6 defines that the semantics $\mathcal{E}[\![\mathsf{p}(t_1, \ldots, t_n)]\!]$ of a derivation tree $\mathsf{p}(t_1, \ldots, t_n)$ is the record that represents its attributed tree. This is obtained by applying a function $p_{\mathcal{E}}$ to attributed subtrees $\mathsf{child}_1, \ldots, \mathsf{child}_n$, and self.

$$\mathcal{E}[\![\mathsf{p}(t_1, \ldots, t_n)]\!] = p_{\mathcal{E}}(\mathsf{child}_1, \ldots, \mathsf{child}_n, \mathsf{self})$$

We can derive Def.3.6 from the three facts: (1) a right-hand side of this expression is equal to 'self' itself, (2) $\mathsf{child}_i = \mathcal{E}[\![t_i]\!]$ $(1 \leq i \leq n)$, and (3) the definition of μ, as follows.

$$\begin{aligned} \mathsf{self} &= p_{\mathcal{E}}(\mathsf{child}_1, \ldots, \mathsf{child}_n, \mathsf{self}) \\ &= \mu\mathsf{self}.p_{\mathcal{E}}(\mathsf{child}_1, \ldots, \mathsf{child}_n, \mathsf{self}) \\ &= \mu\mathsf{self}.p_{\mathcal{E}}(\mathcal{E}[\![t_1]\!], \ldots, \mathcal{E}[\![t_n]\!], \mathsf{self}) \end{aligned}$$

$$\mathcal{E}[\![\mathsf{p}(t_1, \ldots, t_n)]\!] = \mu\mathsf{self}.p_{\mathcal{E}}(\mathcal{E}[\![t_1]\!], \ldots, \mathcal{E}[\![t_n]\!], \mathsf{self})$$

Note that if $Inh(S) \neq \phi$, $\mathcal{E}[\![t]\!]$ is a record λ-abstracted with elements of $Inh(S)$ as defined in Def.3.2. For example, let t be the derivation tree in Fig.2, then we need to apply $\mathcal{E}[\![t]\!]$ to 8 (that is, $\mathcal{E}[\![t]\!](8)$) to obtain the attributed tree of Fig.2 with $\mathsf{N} \cdot \mathsf{radix} = 8$.

3.4 An Example of Evaluating an AG as Records

In this section, we show an evaluating process of an AG as record calculus in order to provide intuitive understanding of $\mathcal{E}$ defined in Section 3.3.

First, we translate the example AG of Section 2.1.1 into a set of functions $p_{\mathcal{E}}$ for all $p \in P$. Then, we apply the semantic function $\mathcal{E}$ to $t = \mathsf{p_N}(\mathsf{p_{I1}}(\mathsf{p_{I1}}(\mathsf{p_{I2}}(\mathsf{p_{D2}}), \mathsf{p_{D3}}), \mathsf{p_{D4}}))$, resulting in a record that represents an attributed tree in Fig. 2.

By Def.3.2, we can translate the example AG into the following functions.

$p_{\mathsf{N}\mathcal{E}} = \lambda\mathsf{child}_1.\lambda\mathsf{self}.\lambda\mathsf{inh}_1.\langle$
  $\mathsf{radix} = \mathsf{inh}_1,\ \mathsf{val} = \mathsf{self}(\mathsf{inh}_1).\mathrm{I.val},$
  $\mathrm{I} = \mathsf{child}_1(0, \mathsf{self}(\mathsf{inh}_1).\mathsf{radix})\rangle$
$p_{\mathsf{I1}\mathcal{E}} = \lambda\mathsf{child}_1.\lambda\mathsf{child}_2.\lambda\mathsf{self}.\ \lambda\mathsf{inh}_1.\lambda\mathsf{inh}_2.\langle$
  $\mathsf{scale} = \mathsf{inh}_1,\ \mathsf{radix} = \mathsf{inh}_2,$
  $\mathsf{val} = \mathsf{self}(\mathsf{inh}_1, \mathsf{inh}_2).\mathrm{I}_2.\mathsf{val}$
    $+ \mathsf{self}(\mathsf{inh}_1, \mathsf{inh}_2).\mathrm{D.val},$
  $\mathrm{I}_2 = \mathsf{child}_1(\mathsf{self}(\mathsf{inh}_1, \mathsf{inh}_2).\mathsf{scale}+1,$
    $\mathsf{self}(\mathsf{inh}_1, \mathsf{inh}_2).\mathsf{radix}),$
  $\mathrm{D} = \mathsf{child}_2(\mathsf{self}(\mathsf{inh}_1, \mathsf{inh}_2).\mathsf{scale},$
    $\mathsf{self}(\mathsf{inh}_1, \mathsf{inh}_2).\mathsf{radix})\rangle$
$p_{\mathsf{I2}\mathcal{E}} = \lambda\mathsf{child}_1.\lambda\mathsf{self}.\lambda\mathsf{inh}_1.\lambda\mathsf{inh}_2.\langle$
  $\mathsf{scale} = \mathsf{inh}_1,\ \mathsf{radix} = \mathsf{inh}_2,$
  $\mathsf{val} = \mathsf{self}(\mathsf{inh}_1, \mathsf{inh}_2).\mathrm{D.val},$
  $\mathrm{D} = \mathsf{child}_1(\mathsf{self}(\mathsf{inh}_1, \mathsf{inh}_2).\mathsf{scale},$

$$
\begin{aligned}
&\quad \text{self}(\text{inh}_1, \text{inh}_2).\text{radix})\rangle \\
p_{\text{D}i\mathcal{E}} &= \lambda\text{self}.\lambda\text{inh}_1.\lambda\text{inh}_2.\langle \\
&\quad \text{scale} = \text{inh}_1,\ \text{radix} = \text{inh}_2, \\
&\quad \text{val} = i \times \text{self}(\text{inh}_1, \text{inh}_2).\text{radix} \\
&\qquad \uparrow \text{self}(\text{inh}_1, \text{inh}_2).\text{scale}\rangle
\end{aligned}
$$

Similar to Def.2.1, '$p_{\text{D}i\mathcal{E}} = \ldots$' is an abbreviation for each definition of $p_{\text{D}0\mathcal{E}}, \cdots, p_{\text{D}9\mathcal{E}}$.

Here we apply the semantic function $\mathcal{E}$ to a term $t = \text{p}_\text{N}(\text{p}_{\text{I}1}(\text{p}_{\text{I}1}(\text{p}_{\text{I}2}(\text{p}_{\text{D}2}), \text{p}_{\text{D}3}), \text{p}_{\text{D}4}))$ that represents the derivation tree for a word "234". That is, we calculate $\mathcal{E}[\![\text{p}_\text{N}(\text{p}_{\text{I}1}(\text{p}_{\text{I}1}(\text{p}_{\text{I}2}(\text{p}_{\text{D}2}), \text{p}_{\text{D}3}), \text{p}_{\text{D}4}))]\!]$ as follows.

$$
\begin{aligned}
&\mathcal{E}[\![\text{p}_{\text{D}2}]\!] \\
&= \mu\text{self}.p_{\text{D}2\mathcal{E}}(\text{self}) && [\text{By Def.3.6}] \\
&= \mathbf{Y}(p_{\text{D}2\mathcal{E}}) \quad \cdots(1) && [\mu x.f(x) = \mathbf{Y}(f)] \\
&= p_{\text{D}2\mathcal{E}}(\mathbf{Y}(p_{\text{D}2\mathcal{E}})) && [\mathbf{Y}(f) = f(\mathbf{Y}(f))] \\
&= \lambda\text{inh}_1.\lambda\text{inh}_2.\langle\ \text{scale} = \text{inh}_1, \text{radix} = \text{inh}_2, \\
&\quad \text{val} = 2 \times \mathbf{Y}(p_{\text{D}2\mathcal{E}})(\text{inh}_1, \text{inh}_2).\text{radix} \uparrow \\
&\qquad \mathbf{Y}(p_{\text{D}2\mathcal{E}})(\text{inh}_1, \text{inh}_2).\text{scale}\rangle \quad \cdots(2) \\
&\qquad\qquad [\text{By applying } p_{\text{D}2\mathcal{E}} \text{ to } \mathbf{Y}(p_{\text{D}2\mathcal{E}})] \\
&= \lambda\text{inh}_1.\lambda\text{inh}_2.\langle\ \text{scale} = \text{inh}_1, \text{radix} = \text{inh}_2, \\
&\qquad\qquad \text{val} = 2 \times \text{inh}_2 \uparrow \text{inh}_1\rangle && [(1)=(2)]
\end{aligned}
$$

Similarly, we can show the rest part of the calculation.

$$
\begin{aligned}
&\mathcal{E}[\![\text{p}_{\text{I}2}(\text{p}_{\text{D}2})]\!] \\
&= \mu\text{self}.p_{\text{I}2\mathcal{E}}(\mathcal{E}[\![\text{p}_{\text{D}2}]\!], \text{self}) \\
&= \mu\text{self}.p_{\text{I}2\mathcal{E}}(\mathcal{E}[\![\text{p}_{\text{D}2}]\!])(\text{self}) \\
&= \mathbf{Y}(p_{\text{I}2\mathcal{E}}(\mathcal{E}[\![\text{p}_{\text{D}2}]\!])) \\
&= p_{\text{I}2\mathcal{E}}(\mathcal{E}[\![\text{p}_{\text{D}2}]\!])(\mathbf{Y}(p_{\text{I}2\mathcal{E}}(\mathcal{E}[\![\text{p}_{\text{D}2}]\!]))) \\
&= \lambda\text{inh}_1.\lambda\text{inh}_2.\langle\text{scale} = \text{inh}_1, \text{radix} = \text{inh}_2, \\
&\quad \text{val} = \mathbf{Y}(p_{\text{I}2\mathcal{E}}(\mathcal{E}[\![\text{p}_{\text{D}2}]\!]))(\text{inh}_1, \text{inh}_2).\text{D.val}, \\
&\quad \text{D} = \mathcal{E}[\![\text{p}_{\text{D}2}]\!] \\
&\qquad (\mathbf{Y}(p_{\text{I}2\mathcal{E}}(\mathcal{E}[\![\text{p}_{\text{D}2}]\!]))(\text{inh}_1, \text{inh}_2).\text{scale}, \\
&\qquad \underline{\mathbf{Y}(p_{\text{I}2\mathcal{E}}(\mathcal{E}[\![\text{p}_{\text{D}2}]\!]))(\text{inh}_1, \text{inh}_2).\text{radix})}\ \rangle \\
&= \lambda\text{inh}_1.\lambda\text{inh}_2.\langle\text{scale} = \text{inh}_1, \text{radix} = \text{inh}_2, \\
&\quad \text{val} = \underline{\mathbf{Y}(p_{\text{I}2\mathcal{E}}(\mathcal{E}[\![\text{p}_{\text{D}2}]\!]))(\text{inh}_1, \text{inh}_2).\text{D}}.\text{val}, \\
&\quad \text{D} = \mathcal{E}[\![\text{p}_{\text{D}2}]\!](\text{inh}_1, \text{inh}_2)\ \rangle \\
&= \lambda\text{inh}_1.\lambda\text{inh}_2.\langle\text{scale} = \text{inh}_1, \text{radix} = \text{inh}_2, \\
&\quad \text{val} = \underline{\mathcal{E}[\![\text{p}_{\text{D}2}]\!](\text{inh}_1, \text{inh}_2).\text{val}}, \\
&\quad \text{D} = \underline{\mathcal{E}[\![\text{p}_{\text{D}2}]\!](\text{inh}_1, \text{inh}_2)}\ \rangle \\
&= \lambda\text{inh}_1.\lambda\text{inh}_2.\langle\text{scale} = \text{inh}_1, \text{radix} = \text{inh}_2, \\
&\quad \text{val} = 2 \times \text{inh}_2 \uparrow \text{inh}_1, \\
&\quad \text{D} = \langle\ \text{scale} = \text{inh}_1, \text{radix} = \text{inh}_2, \\
&\qquad \text{val} = 2 \times \text{inh}_2 \uparrow \text{inh}_1\ \rangle\rangle
\end{aligned}
$$

$$\mathcal{E}[\![\text{p}_{\text{I}1}(\text{p}_{\text{I}2}(\text{p}_{\text{D}2}), \text{p}_{\text{D}3})]\!] = \cdots \text{(omitted)}$$

Underlines in the above calculation mean reducing places. Finally, we obtain the result of this example record calculation shown in Fig.4. By applying the result to '8' as the argument 'inh1', we get a record that represents the attributed tree shown in Fig.2.

4 Formalizing AG extensions

In this section, we formalize as records the following AG extensions: higher-order AGs, recursive AGs and object-oriented AGs. This demonstrates that our denotational semantics fits well with structure-oriented computational models based on AGs.

4.1 Higher-Order AGs

Higher Order AGs (HAGs)[26][23] are a structure-oriented computational model based on AGs. In HAGs, (parts of) attributed trees can be defined by attribute values, and vice versa. As suggested in [23], HAGs should handle both of *attributed* trees and *unattributed* ones. But, to simplify this paper, we assume all trees are attributed. Table 1 summarizes attributed HAGs. In Table 1, e' is the same as e except that all attribute occurrences are replaced as defined in Def.3.2. In Table 1, attribute values $old_1, \ldots, old_n$ are inherited attributes in *old* context, while $new_1, \ldots, new_n$ are those in *new* context.

Note that syntactic reference X_0 has the two meanings: self and self($old_1, \ldots, old_n$), because

- we want e to be attributed in *old* context when accessing to the attributes of e, so the meaning of e should be 'self($old_1, \ldots, old_n$)',
- e should be λ-abstracted to be attributed in *new* context when constructing or grafting trees, so the meaning of e should be 'self'.

Evaluation strategies and techniques like incremental attribute re-evaluation are often important in AG systems, but it is not so straightforward to handle them in record calculus. Of course, it is possible to integrate the two meanings into something like the pair (self, ($old_1, \ldots, old_n$)), which is not given here to keep this paper simple.

4.1.1 An Example HAG

In this section, we show an example HAG to compute factorial numbers, which is almost taken from [26].

Example 4.1 (factorial numbers by HAGs)
$HAG_1 = (G, A, R)$ is defined as follows.

$$
\begin{aligned}
P &= \{p_\text{R} : \text{R} \to \text{F},\ p_{\text{F}1} : \text{F} \to \text{F}, \\
&\quad\ p_{\text{F}2} : \text{F} \to \varepsilon\} \\
R(p_\text{R}) &= \{\text{F} \cdot \text{in} = \text{R} \cdot \text{in}, \text{R} \cdot \text{out} = \text{F} \cdot \text{out}\} \\
R(p_{\text{F}1}) &= \{\text{F}_2 = \text{if}(\text{F}_2 \cdot \text{in} = 0, \text{p}_{\text{F}2}, \text{p}_{\text{F}1}), \\
&\quad\ \text{F}_2 \cdot \text{in} = \text{F}_1 \cdot \text{in} - 1, \\
&\quad\ \text{F}_1 \cdot \text{out} = \text{F}_1 \cdot \text{in} \times \text{F}_2 \cdot \text{out}\} \\
R(p_{\text{F}2}) &= \{\text{F} \cdot \text{out} = 1\}
\end{aligned}
$$

□

The following semantic rule in $R(p_{\text{F}1})$ is important here.

$$\text{F}_2 = \text{if}(\text{F}_2 \cdot \text{in} = 0, \text{p}_{\text{F}2}, \text{p}_{\text{F}1})$$

$$
\begin{aligned}
&\mathcal{E}[\![\mathrm{p_N}(\mathrm{p_{I1}}(\mathrm{p_{I1}}(\mathrm{p_{I2}}(\mathrm{p_{D2}}),\mathrm{p_{D3}}),\mathrm{p_{D4}}))]\!]\\
&= \lambda \mathrm{inh_1}.\langle \mathrm{radix} = \mathrm{inh_1}, \mathrm{val} = ((2 \times \mathrm{inh_1} \uparrow 2) + (3 \times \mathrm{inh_1})) + 4,\\
&\quad \mathrm{I} = \langle \mathrm{scale} = 0, \mathrm{radix} = \mathrm{inh_1}, \mathrm{val} = ((2 \times \mathrm{inh_1} \uparrow 2) + (3 \times \mathrm{inh_1})) + 4,\\
&\qquad \mathrm{I_2} = \langle \mathrm{scale} = 1, \mathrm{radix} = \mathrm{inh_1}, \mathrm{val} = (2 \times \mathrm{inh_1} \uparrow 2) + (3 \times \mathrm{inh_1}),\\
&\qquad\quad \mathrm{I_2} = \langle \mathrm{scale} = 2, \mathrm{radix} = \mathrm{inh_1}, \mathrm{val} = 2 \times \mathrm{inh_1} \uparrow 2,\\
&\qquad\qquad \mathrm{D} = \langle \mathrm{scale} = 2, \mathrm{radix} = \mathrm{inh_1}, \mathrm{val} = 2 \times \mathrm{inh_1} \uparrow 2 \rangle\rangle\\
&\qquad\quad \mathrm{D} = \langle \mathrm{scale} = 1, \mathrm{radix} = \mathrm{inh_1}, \mathrm{val} = 3 \times \mathrm{inh_1} \rangle\rangle\\
&\quad \mathrm{D} = \langle \mathrm{scale} = 0, \mathrm{radix} = \mathrm{inh_1}, \mathrm{val} = 4 \rangle\rangle\rangle
\end{aligned}
$$

Figure 4: Result of Example Record Calculation

expression	record calculus	description
X_0	self/self$(old_1, \ldots, old_n)$	syntactic reference (attributed tree rooted in nonterminal X_0)
X_i	child$_i$/child$_i(old_1, \ldots, old_n)$	syntactic reference (attributed tree rooted in nonterminal $X_i (1 \leq i \leq n)$)
$e.a$	$e'.a$	selection (an attribute value a in the root of attributed tree e)
$p(e_1, \ldots, e_n)$	$\mu \mathrm{self}'.p_{\mathcal{E}}(e'_1, \ldots, e'_n, \mathrm{self}')$	construction (an attributed tree constructed by a constructor p and attributed subtrees $e_1, \ldots, e_n$)
$e\{i_1 = new_1, \ldots, i_n = new_n\}$	$e'(new'_1, \ldots, new'_n)$	attribution expressions (an attributed tree e applied to its inherited attribute values)
semantic rule	**record calculus**	**description**
$X = e$	$X = e'(new'_1, \ldots, new'_n)$	a semantic rule that defines by e the attributed tree rooted in a nonterminal X

Table 1: Extended Syntax and Semantics of Attributed HAGs

It defines the subtree that grows while $\mathrm{F_2} \cdot \mathrm{in} \neq 0$. Fig.5 shows the process of attribute evaluation with R.in = 3. In Fig.5, the leftmost object shows "the attributed tree for $\mathrm{p_R}(\mathrm{p_{F2}})$ where all attributes except nonterminal attributes[3] are evaluated" and white right arrows show "the value of the bottom nonterminal attribute F is bound to $\mathrm{p_{F1}}$ or $\mathrm{p_{F2}}$".

4.1.2 An Example of Calculating Record Semantics of a HAG

We can obtain the following three functions by applying Def.3.2 and Table 1 to the HAG_1 in Example 4.1.

$$
\begin{aligned}
&p_{R\mathcal{E}} = \lambda \mathrm{child_1}.\lambda \mathrm{self}.\lambda \mathrm{inh_1}.\langle\\
&\quad \mathrm{in} = \mathrm{inh_1}, \mathrm{out} = \mathrm{self}(\mathrm{inh_1}).\mathrm{F.out},\\
&\quad \mathrm{F} = \mathrm{child_1}(\mathrm{self}(\mathrm{inh_1}).\mathrm{in})\rangle\\
&p_{F1\mathcal{E}} = \lambda \mathrm{self}.\lambda \mathrm{inh_1}.\langle\\
&\qquad \mathrm{in} = \mathrm{inh_1},\\
&\qquad \mathrm{out} = \mathrm{self}(\mathrm{inh_1}).\mathrm{in} \times \mathrm{self}(\mathrm{inh_1}).\mathrm{F_2.out},\\
&\qquad \mathrm{F_2} = (\mathrm{if}(\mathrm{self}(\mathrm{inh_1}).\mathrm{in} - 1 = 0,\\
&\qquad\quad \mu \mathrm{self_2}.p_{F2\mathcal{E}}(\mathrm{self_2}),\\
&\qquad\quad \mu \mathrm{self_1}.p_{F1\mathcal{E}}(\mathrm{self_1})))(\mathrm{self}(\mathrm{inh_1}).\mathrm{in} - 1)\rangle\\
&p_{F2\mathcal{E}} = \lambda \mathrm{self}.\lambda \mathrm{inh_1}.\langle \mathrm{in} = \mathrm{inh_1}, \mathrm{out} = 1\rangle
\end{aligned}
$$

[3] A nonterminal occurring in the left-hand side of semantic rules is called a *nonterminal attribute*[26].

Now we can compute $\mathcal{E}[\![\mathrm{p}_R(\mathrm{p_{F1}})]\!](3)$. This calculation process corresponds to Fig.5. First, we calculate $\mathcal{E}[\![\mathrm{p_{F1}}]\!](3)$.

$$
\begin{aligned}
&\mathcal{E}[\![\mathrm{p_{F1}}]\!](3)\\
&= \langle \mathrm{in} = 3,\\
&\quad \mathrm{out} = 3 \times (\mu \mathrm{self_1}.p_{F1\mathcal{E}}(\mathrm{self_1}))(2).\mathrm{out},\\
&\quad \mathrm{F_2} = (\mu \mathrm{self_1}.p_{F1\mathcal{E}}(\mathrm{self_1}))(2)\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mu \mathrm{self_1}.p_{F1\mathcal{E}}(\mathrm{self_1}))(2)\\
&= \langle \mathrm{in} = 2,\\
&\quad \mathrm{out} = 2 \times (\mu \mathrm{self_1}.p_{F1\mathcal{E}}(\mathrm{self_1}))(1).\mathrm{out},\\
&\quad \mathrm{F_2} = (\mu \mathrm{self_1}.p_{F1\mathcal{E}}(\mathrm{self_1}))(1)\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mu \mathrm{self_1}.p_{F1\mathcal{E}}(\mathrm{self_1}))(1)\\
&= \langle \mathrm{in} = 1,\\
&\quad \mathrm{out} = 1 \times (\mu \mathrm{self_2}.p_{F2\mathcal{E}}(\mathrm{self_2}))(0).\mathrm{out},\\
&\quad \mathrm{F_2} = (\mu \mathrm{self_2}.p_{F2\mathcal{E}}(\mathrm{self_2}))(0)\rangle
\end{aligned}
$$

$$
\mu \mathrm{self_2}.p_{F2\mathcal{E}}(\mathrm{self_2}))(0) = \langle \mathrm{in} = 0, \mathrm{out} = 1\rangle
$$

$$
\begin{aligned}
&\mathcal{E}[\![\mathrm{p_R}(\mathrm{p_{F1}})]\!](3)\\
&= \langle\ \mathrm{in} = 3, \mathrm{out} = 6,\\
&\quad \mathrm{F} = \langle\ \mathrm{in} = 3, \mathrm{out} = 6,\\
&\qquad \mathrm{F_2} = \langle\ \mathrm{in} = 2, \mathrm{out} = 2,\\
&\qquad\quad \mathrm{F_2} = \langle\ \mathrm{in} = 1, \mathrm{out} = 1,
\end{aligned}
$$

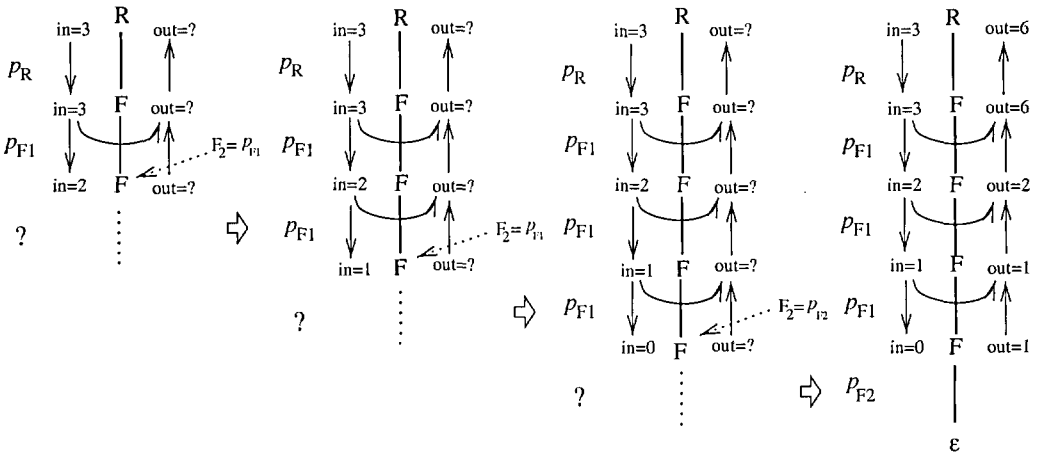


Figure 5: A Calculation of the factorial of 3

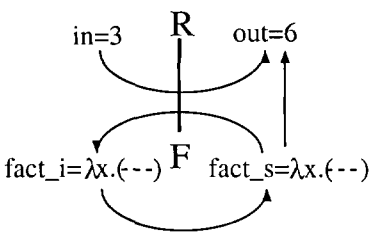


Figure 6: An Attributed Tree of RAG_1 with $\mathrm{R} \cdot \mathrm{in} = 3$

$$\mathrm{F}_2 = \langle\, \mathrm{in} = 0,\ \mathrm{out} = 1\rangle\rangle\rangle\rangle\rangle$$

The result record matches the rightmost attributed tree in Fig.5.

4.2 Recursive AGs

In [8], Farrow pointed out that even a circularly (i.e. recursively) defined AG can be evaluated if its recursive definition has the least fixed-point. Such AGs are called recursive AGs (RAGs). Our formal semantics can deal with RAGs as it is. To show this, this section gives an example RAG and translates the RAG into records.

Example 4.2 (RAG_1**: factorial numbers by RAGs)**
$RAG_1 = (G, A, R)$ is defined as follows.

$$\begin{aligned} P &= \{p_{\mathrm{R}} : \mathrm{R} \to \mathrm{F},\ p_{\mathrm{F}} : \mathrm{F} \to \epsilon,\ \} \\ R(p_{\mathrm{R}}) &= \{\mathrm{R} \cdot \mathrm{out} = \mathrm{apply}(\mathrm{F} \cdot \mathrm{fact\_s}, \mathrm{R} \cdot \mathrm{in}), \\ &\quad\ \mathrm{F} \cdot \mathrm{fact\_i} = \mathrm{F} \cdot \mathrm{fact\_s}\ \} \\ R(p_{\mathrm{F}}) &= \{\mathrm{F} \cdot \mathrm{fact\_s} = \lambda x.\mathrm{if}(x = 0, 1, \\ &\quad\ \mathrm{x} \times \mathrm{F} \cdot \mathrm{fact\_i}(\mathrm{x} - 1))\ \} \end{aligned}$$

□

RAG_1 defines factorial numbers as follows.

$$\begin{aligned} &\mathrm{R} \cdot \mathrm{out} = \mathit{FACT}(\mathrm{R} \cdot \mathrm{in}) \\ &\mathrm{F} \cdot \mathrm{fact\_i} = \mathrm{F} \cdot \mathrm{fact\_s} = \mathit{FACT} \\ &= \lambda x.\mathrm{if}(x = 0, 1, x \times \mathit{FACT}(x - 1)) \end{aligned}$$

Fig.6 shows an attributed tree of RAG_1 with $\mathrm{R} \cdot \mathrm{in} = 3$. As shown Fig.6, fact_i and fact_s are circularly defined. In the following, we represent RAG_1 as record calculus. By Def.3.2, we can translate RAG_1 into the two functions.

$$\begin{aligned} &p_{\mathrm{R}\mathcal{E}} = \lambda\mathrm{child}_1.\lambda\mathrm{self}.\lambda\mathrm{inh}_1.\langle \\ &\ \mathrm{in}{=}\mathrm{inh}_1, \\ &\ \mathrm{out}{=}\mathrm{apply}(\mathrm{self}(\mathrm{inh}_1).\mathrm{F.fact\_s}, \mathrm{self}(\mathrm{inh}_1).\mathrm{in}), \\ &\ \mathrm{F}{=}\mathrm{child}_1(\mathrm{self}(\mathrm{inh}_1).\mathrm{F.fact\_s})\rangle \\ &p_{\mathrm{F}\mathcal{E}} = \lambda\mathrm{self}.\lambda\mathrm{inh}_1.\langle \\ &\ \mathrm{fact\_i}{=}\mathrm{inh}_1, \\ &\ \mathrm{fact\_s}{=}\lambda x.\mathrm{if}(x = 0, 1, \\ &\qquad x \times (\mathrm{self}(\mathrm{inh}_1).\mathrm{fact\_i})(x - 1))\rangle \end{aligned}$$

The following is the record semantics of RAG_1 (calculating steps are omitted).

$$\begin{aligned} &\mathcal{E}[\![p_{\mathrm{R}}(p_{\mathrm{F}})]\!] \\ &= \lambda\mathrm{inh}_1.\langle \mathrm{in} = \mathrm{inh}_1, \mathrm{out} = \mathit{FACT}(\mathrm{inh}_1), \\ &\ \mathrm{F} = \langle \mathrm{fact\_i} = \mathit{FACT}, \mathrm{fact\_s} = \mathit{FACT}\rangle\rangle \end{aligned}$$

where

$$\begin{aligned} &\mathit{FACT} \\ &\equiv \mathbf{Y}(p_{\mathrm{R}\mathcal{E}}(\mathcal{E}[\![p_{\mathrm{F}}]\!]))(\mathrm{inh1}).\mathrm{F.fact\_}s \\ &= \lambda x.\mathrm{if}(x = 0, 1, x \times \mathit{FACT}(x - 1)) \end{aligned}$$

4.3 OOAG

We have introduced a computational model OOAG (Object-Oriented AGs)[22][9], which is an extension of standard AGs by importing message passing and assignment to instance variables (i.e. multiple subtree replacement). This extension makes it easier to describe dynamic aspects of systems such as:

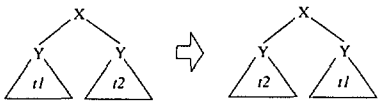


Figure 7: Swapping Subtrees

- *retrieval*: getting attribute values or attributed (sub)trees by sending retrieval messages to attributed trees
- *updating*: changing tree structures depending on their attribute values or by sending updating messages to attributed trees

In OOAG, an attributed tree is regarded as an aggregated object in object-oriented programming, and attributed subtrees and attributes are regarded as instance variables. Message passing may cause subtree replacements. As a result of the subtree replacements, attribute values in the grafting point generally become inconsistent with each other, that is, attribute values do not satisfy their semantic rules. To recover the situation, the evaluation process of attributes will be executed for the entire attributed tree in consistent with the semantic rules. Thus, in OOAG, message passing and attribute evaluation are alternately repeated.

OOAG is similar in nonterminal occurrences to HAGs, but OOAG is definitely different from HAGs. OOAG can update an attributed subtree with another one, while HAGs can not. To show this, here is given an example OOAG where a subtree swapping method is implemented. The method just returns a new object where subtrees are swapped. So, strictly speaking, this implementation given here does not implement mutable objects.

Example 4.3 (swapping subtrees by OOAG)

$$\begin{aligned} P &= \{p_X : X \to Y\ Y,\ p_{Y1} : Y \to \varepsilon, \\ &\quad\ p_{Y2} : Y \to \varepsilon\} \\ R(p_X) &= \{X \cdot \mathrm{swap} = f_{p_X}(Y_2, Y_1)\} \\ R(p_{Y1}) &= \{Y \cdot \mathrm{id} = 1\} \\ R(p_{Y2}) &= \{Y \cdot \mathrm{id} = 2\} \end{aligned}$$

□

By Def.3.2, we obtain the following functions from Example 4.3.

$p_{X\mathcal{E}}$=μmyclass.λchild$_1$.λchild$_2$.λself.⟨
  swap = μself$_2$.myclass(child$_2$, child$_1$, self$_2$),
  Y_1=child$_1$, Y_2=child$_2$⟩
$p_{Y1\mathcal{E}}$ = λself.⟨ id=1 ⟩
$p_{Y2\mathcal{E}}$ = λself.⟨ id=2 ⟩

First, we have the following attributed tree t as a result of evaluating $\mathcal{E}[\![p_X(p_{Y1}, p_{Y2})]\!]$.

$$\begin{aligned} &t = \mathcal{E}[\![p_X(p_{Y1}, p_{Y2})]\!] \\ &= \langle\ \mathrm{swap} = \mu\mathrm{self}_2.p_{X\mathcal{E}}(\langle \mathrm{id} = 2\rangle, \langle \mathrm{id} = 1\rangle, \mathrm{self}_2), \\ &\quad Y_1 = \langle \mathrm{id} = 1\rangle, Y_2 = \langle \mathrm{id} = 2\rangle\rangle \end{aligned}$$

Next, we can obtain the result of t.swap as follows.

$$\begin{aligned} &t.\mathrm{swap} = \mathcal{E}[\![p_X(p_{Y1}, p_{Y2})]\!].\mathrm{swap} \\ &= \langle\ \mathrm{swap} = \mu\mathrm{self}_2.p_{X\mathcal{E}}(\langle \mathrm{id} = 1\rangle, \langle \mathrm{id} = 2\rangle, \mathrm{self}_2), \\ &\quad Y_1 = \langle \mathrm{id} = 2\rangle, Y_2 = \langle \mathrm{id} = 1\rangle\rangle \end{aligned}$$

This record represents an attributed tree where Y_1 and Y_2 are swapped.

5 A Simple Implementation using SML/NJ

Our new formalization introduced in Section 3 and 4 can be straightforwardly implemented in functional programming languages with record types, especially lazy ones. To show this, we provide a simple implementation using Standard ML of New Jersey (SML/NJ for short)[25]. Using this implementation method, readers can easily experiment new ideas of AG extentions as running codes.

Fig. 8 shows a SML code implementing the record semantics of the HAG described in Section 4.1.2[4]. If you load and execute the SML code, you can see the following result, which shows $\mathcal{E}[\![p_R(p_{F1})]\!](5) = 120$.

```
% sml
- use "hag.sml";
(omitted)
val tree = p_R1_ p_F1_ : R_
val atree = p_R1 {F=fn, in1=fn, out=fn} : R
val out = 120 : int
val it = () : unit
-
```

Note that a closure technique of delaying evaluation with functions [20] is used in the SML code in Fig. 8, since SML/NJ uses call-by-value evaluation rather than lazy evaluation. Without this technique, the computation may not terminate on call-by-value evaluators including SML/NJ. The technique is to write:

- "`fn () =>` *expr*", to delay the evaluation of *expr*.

  This represents an anonymous function whose argument `()` is a dummy, unused empty tuple. The type of `()` is `unit`. A function body *expr* is not evaluated until the function is applied.

- "*delayed_expr* `()`", where *delayed_expr* is an expression that evaluated the form "`fn () =>` *expr*", to force the evaluation of the function body *expr*.

  This represents the application of *delayed_expr* to an empty tuple. The body *expr* of *delayed_expr* is evaluated now.

From our simple implementation, we found SML/NJ is powerful enough to describe our AG formalization straightforwardly except the following points.

[4]We implemented all examples in this paper in the same way, but we do not give them for lack of space.

- We have to write many `case` pattern matchings even for records that have the same fields, since SML/NJ has no support for subtyping or inclusion polymorphism.
- We have to define the semantic function $\mathcal{E}$ as several functions (e.g., `eval_R` and `eval_F` in Fig. 8) depending on a given AG. This is because in SML/NJ there is no way to simply describe a function to operate on different user-defined types that can not be parameterized.

6 Conclusion

In this paper, we first presented a new denotational semantics of attribute grammars (AGs) based on Cardelli's record calculus. This semantics is structure-oriented as well as natural and simple. Unlike previous works, an attributed tree is represented as a nested record to preserve the structural information.

Our AG semantics is simple and natural because:

- Our record representation for AGs preserves structures of attributed trees as well as values of attributes. There are no extra fields in records; all fields represent only attributes or attributed subtrees (Def.3.1).
- A function $p_{\mathcal{E}}$ is easily obtained by rewriting p and $R(p)$ (Def.3.2).
- The definition of $\mathcal{E}$ is defined as simple recursion on tree structures as follows (Def.3.6).

$$\mathcal{E}[\![\mathrm{p}(t_1, \ldots, t_n)]\!] = \mu\mathrm{self}.p_{\mathcal{E}}(\mathcal{E}[\![t_1]\!], \ldots, \mathcal{E}[\![t_n]\!], \mathrm{self})$$

- Underlying record calculus is simple.

We think the semantics is a good theoretical groundwork for modeling AG extensions (especially structure-oriented ones). To show this, we represented HAGs, RAGs and OOAG as record calculus in Section 4. We also showed the semantics can be implemented straightforwardly and simply in a functional language SML/NJ.

7 Future Works

The paper emphasizes that Cardelli's record calculus makes it easy to formalize AGs, HAGs, RAGs and OOAG with a structure-oriented view. In [3], however, Cardelli's record calculus is used to formalize inheritance. We will provide some formalization of AG inheritance extending our AG semantics, and compare it with previous works on AG inheritance, e.g., [11][19][18].

Another issue is the generality of our AG semantics. We plan to apply our AG semantics to many other AG extensions, e.g., modularity concepts[16], remote access[10][12], and so on.

Acknowledgment

The authors would like to thank our OOAG project members, especially Takashi Hagiwara, for their efforts on developing MAGE2 system and their useful discussions. We would also like to thank Kikuchi, Yutaka for reading the draft and making a number of helpful suggestions.

References

[1] L. Cardelli. A semantics of multiple inheritance. In *Information and Computation*, pages 138–164, 1988.

[2] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *FPCA '89 Conf. Proc.*, pages 273–281, 1989.

[3] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proc. 17th ACM Sympo. on Principles of Programming Languages*, pages 125–135. ACM, 1990.

[4] W. Cook and J. Palsberg. Denotational semantics of inheritance and its correctness. In *Proc. 4th ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 433–443, 1989.

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. In *ACM Computing Surveys, Vol.17, No.4*, pages 471–522. ACM, 1985.

[6] Pierre Deransart, Martin Jourdan, and Bernard Lorho. Attribute Grammars: Definitions, Systems, and Bibliography, volume 323 of *Lec. Notes in Comp. Sci.* Springer-Verlag, 1988.

[7] R. Farrow. Generating a Production Compiler from an Attribute Grammar. *IEEE Software*, 1(4):77–93, 1984.

[8] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proc. the ACM SIGPLAN '86 Sympo. on Compiler Construction*, pages 85–98, Palo Alto, Calif., 1986. ACM.

[9] Katsuhiko Gondow, Takashi Imaizumi, Yoichi Shinoda, and Takuya Katayama. Change management and consistency maintenance in software development environments using object oriented attribute grammars. In *Object Technologies for Advanced Software (Proc. 1st JSSST Int. Sympo.)*, volume 742 of *Lec. Notes in Comp. Sci.*, pages 77–94. Springer Verlag, 1993.

[10] GrammaTech, Inc., Ithaca, NY. The Synthesizer Generator Reference Manual, fifth edition, 1996.

```
infix 8 >>;
fun record >> label = label record (); (* >>: field selector *)
fun fix f inhs = f (fix f) inhs;       (* fix: fixed-point operator *)

(* type definitions of abstract syntax trees (production rules) *)
datatype R_ = p_R1_ of F_
and      F_ = p_F1_ | p_F2_
  ;
(* type definitions of attributed trees *)
datatype R = p_R1 of {F:unit->F, in1:unit->int, out:unit->int}
and      F = p_F1 of {F2:unit->F, in1:unit->int, out:unit->int}
           | p_F2 of {in1:unit->int, out:unit->int}
  ;
(* semantic rules *)
(* p_Rε *)
val rec p_R_e = fn child1 => fn self => fn inh1 =>
  p_R1 {
          in1 = inh1, (* 'in1' is used instead of 'in' since 'in' is a key-
word in SML/NJ *)
          out = fn () => (case (self inh1) of
                            p_R1 r => (case r >> #F of p_F1 r2 => r2 >> #out)),
                            (* self(inh1).F.out *)
          F   = fn () => child1 (fn () => case (self inh1) of p_R1 r => r >> #in1)
                            (* child1(self(inh1).in) *)
        }
(* p_F1ε *)
and p_F1_e = fn self => fn inh1 =>
  p_F1 {
          in1 = inh1,
          out = fn () => (case (self inh1) of p_F1 r => r >> #in1) *
              (case (self inh1) of p_F1 r =>
                 (case r >> #F2 of p_F1 r2 => r2 >> #out | p_F2 r2 => r2 >> #out)),
               (* self(inh1).in×self(inh1).F2.out *)
          F2  = fn () => (if (case (self inh1) of p_F1 r => r >> #in1)-1=0
                          then fix p_F2_e else fix p_F1_e)
                         (fn () => (case (self inh1) of p_F1 r => r >> #in1)-1)
               (* (if(self(inh1).in − 1 = 0, μself2.p_F2ε(self2), μself1.p_F1ε(self1))) (self(inh1).in − 1) *)
        }
(* p_F2ε *)
and p_F2_e = fn self => fn inh1 =>
  p_F2 {
          in1 = inh1,
          out = fn () => 1
        }
  ;
(* functions that implements the semantic function ℰ *)
fun eval_R t = case t of p_R1_ t1 => fix (p_R_e (eval_F t1))
and eval_F t = case t of p_F1_ => fix p_F1_e | p_F2_ => fix p_F2_e
  ;
(* for sample execution *)
val tree  = p_R1_ p_F1_;
val atree = eval_R tree (fn () => 5);
val out   = case atree of p_R1 r => r >> #out;
```

Figure 8: A Simple Implementation in SML/NJ of the HAG Example 4.1

[11] Görel Hedin. An object-oriented notation for attribute grammars. In S. Cook, editor, *Proc. ECOOP '89*, pages 329–345, Nottingham, July 1989. Cambridge University Press.

[12] Görel Hedin. Reference Attributed Grammars. In *Proc. Second Workshop on Attribute Grammars and their Applications (WAGA99)*, pages 153–172, 1999.

[13] Thomas Johnsson. Attribute Grammars as a Functional Programming Paradigm, volume 274 of *Lec. Notes in Comp. Sci.*, pages 154–173. Springer Verlag, 1987.

[14] U. Kastens, B Hutt, and E Zimmermann. GAG: A Practical Compiler Generator, volume 141 of *Lec. Notes in Comp. Sci.* Springer Verlag, 1982.

[15] D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[16] D.E. Knuth. Semantics of context-free languages: Correction. *Mathematical Systems Theory*, 5(1):95–96, 1971.

[17] Uwe Kastens and William M. Waite. Modularity and Reusability in Attribute Grammars. *Acta Informatica*, 31:601–627, 1994.

[18] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. Multiple Attribute Grammar Inheritance. In *Proc. Second Workshop on Attribute Grammars and their Applications (WAGA99)*, pages 57–76, 1999.

[19] Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. In *ACM Computing Surveys, Vol.27, No.2*, pages 196–255. ACM, 1995.

[20] Chris Reade. Elements of Functional Programming. Addison Wesley, 1989.

[21] Thomas W. Reps and Tim Teitelbaum. The Synthesizer Generator. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[22] Yoichi Shinoda and Takuya Katayama. Object Oriented Extension of Attribute Grammars and Its Implementation Using Distributed Attribute Evaluation Algorithm. In *Proc. Int. Workshop on Attribute Grammars and their Applications,* Lec. Note in Comp. Sci. *Vol. 461*, pages 177–191. Springer-Verlag, 1990.

[23] Tim Teitelbaum and R. Chapman. Higher-order attribute grammars and editing environments. In *Proc. ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation, Vol. 25, No. 6*, pages 197–208, White Plains, NY, 1990.

[24] Masayuki Takeda and Takuya Katayama. On defining denotational semantics for attribute grammars. *Journal of Information Processing*, 5(1):21–29, 1982.

[25] Lucent Technologies and Bell Laboratories. SML/NJ: Standard ML of New Jersey. http://cm.bell-labs.com/cm/cs/what/smlnj/index.html., 1996.

[26] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proc. ACM SIGPLAN'89 Conf. on Programming Language Design and Implementation, Vol. 25, No. 6*, pages 131–145, 1989.

Reference Attributed Grammars

Görel Hedin
Dept. of Computer Science
Lund University
Sweden
E-mail: Gorel.Hedin@cs.lth.se

Keywords: Attribute Grammars, Object-Oriented Languages, Reference Attributes, Remote Attribute Access

Edited by: Marjan Mernik and Didier Parigot
Received: June 8, 1999 **Revised:** September 4, 2000 **Accepted:** September 11, 2000

An object-oriented extension to canonical attribute grammars is described, permitting attributes to be references to arbitrary nodes in the syntax tree, and attributes to be accessed via the reference attributes. Important practical problems such as name and type analysis for object-oriented languages can be expressed in a concise and modular manner in these grammars, and an optimal evaluation algorithm is available. An extensive example is given, capturing all the key constructs in object-oriented languages including block structure, classes, inheritance, qualified use, and assignment compatibility in the presence of subtyping. The formalism and algorithm have been implemented in APPLAB, an interactive language development tool.

1 Introduction

Canonical attribute grammars (AGs), as introduced by Knuth [26], is an appealing formalism that allows context-sensitive properties of individual constructs in a language to be described in a declarative way, and to be automatically computed for any program in the language. Important applications include defining context-sensitive syntax and code generation for a language.

A major problem with canonical AGs is that the specifications often become too low-level when dealing with non-local dependencies, i.e., situations where a property of one syntax tree node is dependent on properties of nodes far away in the tree. For example, the type of an identifier use site depends on the type of the declaration which may be located arbitrarily far away in the tree.

Many researchers have suggested different extensions to attribute grammars to solve this problem, e.g. [3, 4, 5, 12, 14, 15, 16, 17, 19, 20, 23, 33, 34, 38]. Our approach is in the line of our earlier work [12, 14, 15, 16], of Poetzsch-Heffter [33, 34], and of Boyland [4] in that we propose an extension that permits attributes to be explicit references denoting nodes arbitrarily far away in the syntax tree, and attributes of those nodes to be accessed via such reference attributes. Similar to Poetzsch-Heffter and Boyland we propose a recursive evaluation algorithm that allows optimal evaluation for non-circular AGs with such extensions. The formalism we propose, Reference Attributed Grammars (RAGs), casts these extensions into an object-oriented form, allowing advanced static-semantic analysis problems to be expressed in a concise and modular manner. We give an extensive example of this by providing a complete specification of PicoJava, a small subset of Java including key constructs found in object-oriented languages such as block structure, classes, inheritance, qualified use, and assignment compatibility in the presence of subtyping. We have implemented the formalism and evaluation algorithm in our interactive language development tool APPLAB (APPlication language LABoratory) [6, 7].

The rest of this paper is structured as follows. In Section 2 a background is given on canonical AGs and their drawbacks. Section 3 introduces the basic RAG formalism, discusses the evaluation algorithm, and compares RAGs to canonical AGs. Section 4 discusses additional object-oriented features of RAGs, including a class hierarchy for nonterminals and support for virtual function attributes. Section 5 shows an extensive example of name and type analysis for an object-oriented language, PicoJava. Section 6 discusses our tool APPLAB, Section 7 relates to other work, and Section 8 concludes the paper and suggests future research.

2 Background

2.1 Canonical attribute grammars

A canonical attribute grammar consists of a context-free grammar extended with attributes for the nonterminals and semantic rules for the productions. The attributes are characterized as *synthesized* or *inherited*, depending on if they are used to transmit information upwards or downwards in the syntax tree. Given a production $X_0 \rightarrow X_1 \ldots X_n$, a semantic rule is written $a_0 = f(a_1, \ldots, a_m)$ and defines a_0 as the value of applying the *semantic function* f to the attributes $a_1, \ldots, a_m$. The attribute a_0 must be either a synthesized attribute of X_0 or an inherited attribute

of $X_j, 1 \leq j \leq n$. I.e., a semantic rule defines either a synthesized attribute of the left-hand symbol of the production, or an inherited attribute of one of the symbols on the right hand side of the production. A function argument, $a_k, 1 \leq k \leq m$, must be an attribute of $X_j, 0 \leq j \leq n$. I.e., a rule is local, depending only on information available in the attributes of the symbols of the production.

A grammar is considered to be *well-formed* if each attribute in any syntax tree of the grammar has exactly one defining semantic rule. This is obtained by restricting the start symbol to have synthesized attributes only, and by requiring a production $X_0 \rightarrow X_1 \ldots X_n$ to have exactly one rule for each synthesized attribute of X_0 and one rule for each inherited attribute of $X_j, 1 \leq j \leq n$.

The assignment of values to attributes of a syntax tree is called an *attribution*. An attribution is called a *solution* if all semantic rules are satisfied. A well-formed grammar is considered to be *well-defined* if there exists exactly one solution (or one *best* solution according to some criteria) for each syntax tree of the grammar.

If an attribute a_1 is used for defining another attribute a_2 we say that there is a *dependency* (a_1, a_2). If the dependency graph for a syntax tree is non-circular, the attribution can be obtained simply by applying the semantic functions in topological order, provided that the semantic functions terminate. If each syntax tree derivable from a grammar will have a non-circular dependency graph, the grammar is said to be *non-circular*. Usually, canonical AGs are required to be non-circular, but there are also extensions which allow circular dependencies. The usual requirement for such grammars is that the values in the domain of an attribute on a cyclic dependency chain can be arranged in a lattice of finite height, and that all semantic functions are monotonic with respect to these lattices. In this case, there will be at least one solution, and the solution with the "least" attribute values is taken to be the best one. For such circular grammars, the attribution can be obtained by iteratively applying the semantic functions, giving the attributes on the cycle the lattice bottom values as start values. See, e.g. [10, 21].

2.2 Problems with canonical attribute grammars

Canonical AGs are well-suited for description of problems where the dependencies are local and follow the syntax tree structure. For example, in type analysis, the type of an operator may depend on the types of its operands. Canonical AGs are less suited for description of problems with non-local dependencies, such as name analysis problems where properties of an identifier use site depends on properties of an identifier declaration site. Typically, the use and declaration sites can be arbitrarily far away from each other in the tree, and any information propagated between them needs to involve all intermediate nodes. There are several drawbacks with this.

One drawback is that the information about declarations in the syntax tree needs to be replicated in the attributes: To do static semantic analysis, all declared names in a scope, together with their appropriate type information, need to be bundled together into an aggregate attribute, the "environment", and distributed to all potential use sites. At each use site, the appropriate information is looked up.

A second drawback is that the aggregate attributes with information replicated from the syntax tree can become very complex. The distribution of the aggregate information works well for procedural languages with Algol-like scope rules (nested scopes), but is substantially more difficult for languages with more complex scope rules, for example modular languages and object-oriented languages. For example, the use of qualified access in a language implies that it is not sufficient with a single environment attribute at each use site—it is necessary to provide access to all potentially interesting environments and select the appropriate one depending on the type of the qualifying identifier. The aggregate attributes thus need to become more complex, and to contain also information about relations between different declarations. The semantic functions working on these complex attributes naturally also become more complex. The AG formalism does not itself support the description of these complex attributes and functions.

A third drawback is that it is difficult to extend the grammar. Suppose we have a grammar with a working name analysis for extracting types, and we want to extend it by propagating also the declaration kind, i.e. information about if the declaration is a constant or a variable. There are two alternatives for modelling this. Either we introduce an additional environment attribute which maps names to kinds and is defined analogously to the environment mapping names to types. Just like the type environment, the definition of the kind environment needs to involve all intermediate nodes. A second alternative is to modify the original type environment to also include kind information. None of these alternatives is very attractive since we cannot describe the extension in a clean concise way.

A fourth drawback with canonical grammars is that they are not suited for incremental evaluation. This is partly because there is no mechanism for incremental updating of the aggregated attributes (environments) and partly because a change to a declaration typically affects attributes all over the syntax tree (i.e., the environments), even though the extracted information is unchanged. Incremental evaluation based on this model does thus not scale up.

In this paper we address the first three of these drawbacks.

3 Reference Attributed Grammars (RAGs)

3.1 Reference attributes

Canonical attribute grammars assume value semantics for the attributes. I.e., an attribute cannot (conceptually) be a reference to an object, or have a value containing such references. From an implementation point of view it is possible, and common, to implement two attributes with the same value as references to the same object. However, this is merely an implementational convenience for saving space, and the fact that these two attributes refer to the same object cannot be used in the grammar. I.e., the implementation is referentially transparent, preserving the value semantics of the grammar.

In our extension to canonical attribute grammars, attributes are allowed to be references to nodes in the syntax tree. Thus, we abandon the value semantics and introduce reference semantics. Structured attributes like sets, dictionaries, etc., may also include reference values. As we will illustrate in Section 5, the use of reference values makes attribute grammars well-suited for expressing problems with non-local dependencies that do not necessarily follow the syntax tree structure.

A reference value denoting a node in the syntax tree may be dereferenced to access the attributes of that node. This way, a reference attribute constitutes a direct link from one node to another node arbitrarily far away in the syntax tree, and information can be propagated directly from the referred node to the referring node, without having to involve any of the other nodes in the syntax tree. We call an attribute grammar extended with this capability a *reference attributed grammar* (RAG).

3.2 TINY: an example RAG

Figure 1 shows the RAG specification of TINY, a tiny language made up to illustrate some central concepts in RAGs. TINY is so simple that it has only one possible syntax tree, which is shown with its attribution in Figure 2.

Nonterminal	*Attributes*	*Productions*	*Semantic rules*
A		A → B C	B.rC = C C.rB = B
B	↓ rC: **ref** (C) ↑ b: **integer**	B →	B.b = B.rC.c
C	↓ rB: **ref** (B) ↑ c: **integer**	C →	C.c = 7

Figure 1: RAG specification of TINY

The example illustrates important aspects of RAGs. First, by considering the reference attributes in addition to the tree links, the syntax tree can be viewed as a (syntax)

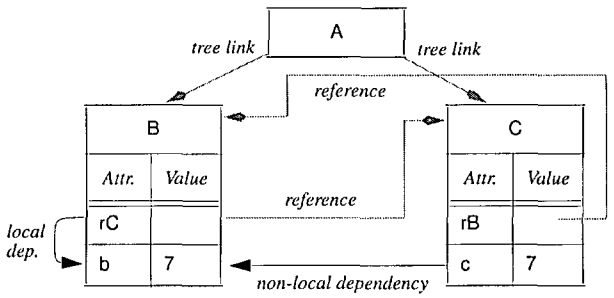


Figure 2: RAG attribution of TINY (non-circular)

graph. The syntax graph may contain cycles: the B node contains a reference attribute rC denoting the C node which in turn contains a reference attribute rB referring back to the B node. However, although the syntax graph contains a cycle, the dependencies between the attributes form a non-circular graph, and the RAG is thus non-circular. Since all semantic functions terminate, the RAG is well-defined, and a unique solution has been found for the tree by evaluating the attributes in topological order, e.g., rB, c, rC, b.

The value of a reference attribute is the (unique) identity of the denoted node, drawn as an arrow in the figure. This value can be computed before the attributes of the denoted node are evaluated, and does thus not depend on those attributes. In the example, the semantic rules defining rC and rB depend only on constant values (the identities of nonterminals B and C), and rB and rC do therefore not have any incoming dependency edges.

In a canonical AG all dependencies are local, i.e., they occur because an attribute of a nonterminal X_1 in a production is defined using an attribute of a nonterminal X_2 in the same production. For any given syntax tree, it is possible to determine the complete dependency graph without evaluating any attributes. In a RAG, there are non-local dependencies in addition to the local dependencies. A non-local dependency (a, b) occurs when b is defined by a semantic function that accesses a via a reference attribute r. The dependency (a, b) can be determined only after evaluating the reference attribute r. In the TINY example, the non-local dependency from c to b can be determined only after rC has been given a value.

As will be shown in Section 5, practical grammars for complex problems, like name analysis for object-oriented languages, can be written concisely using a non-circular RAG.

3.3 Attribute evaluation

Similar to a non-circular canonical AG, a non-circular RAG can be evaluated simply by following the dependencies, evaluating the attributes in topological order. As noted above, the dependency graph for a RAG cannot, in contrast to canonical AGs, be completely determined before evaluation, it has to be determined *during* the evaluation. Algo-

	Attributes		*Semantic rules*
A	↑ id: **integer** ↑ ct: <> ↑ subCt: **array[tuple]** ↑ allCt: **array[tuple]**	A → B C	B.rC = C.id C.rB = B.id A.id = 1 B.id = A.id+1 C.id = B.maxId+1 A.ct = <> A.subCt = [A.id → A.ct] ∪ B.subCt ∪ C.subCt A.allCt = A.subCt B.allCt = A.allCt C.allCt = A.allCt
B	↓ rC: **integer** ↑ b: **integer** ↓ id: **integer** ↑ maxId: **integer** ↑ ct: <**integer, integer**> ↑ subCt: **array[tuple]** ↓ allCt: **array[tuple]**	B →	B.b = B.allCt[B.rC](2) B.maxId = B.id B.ct = <B.rC, B.b> B.subCt = [B.id → B.ct]
C	↓ rB: **integer** ↑ c: **integer** ↓ id: **integer** ↑ maxId: **integer** ↑ ct: <**integer, integer**> ↑ subCt: **array[tuple]** ↓ allCt: **array[tuple]**	C →	C.c = 7 C.maxId = C.id C.ct = <C.rB, C.c> C.subCt = [C.id → C.ct]

Figure 3: Table-translated specification of TINY (canonical AG form)

rithms based on static computation of dependency graphs, such as for OAGs [24] are therefore not immediately applicable to RAGs. However, demand-driven algorithms, i.e., where each attribute access is replaced by a call to the corresponding semantic function, can be directly used for RAGs and will work for any non-circular RAG, as also noted by [33] and [4]. By caching an attribute value at the first access and returning the cached value at subsequent accesses, this evaluation algorithm becomes optimal. Setting a flag for attributes under evaluation allows circularities in the grammar to be found at evaluation time. Several implementations of this algorithm have been presented for canonical attribute grammars [27, 18, 22]. In our system (APPLAB), we have implemented the algorithm for RAGs by using techniques from object-oriented programming, as described for canonical AGs in [13]. This technique fits well with the object-oriented extensions we have done to RAGs (see Section 4) and makes the translation particularly simple.

3.4 Translation of a RAG to a canonical AG

To show the relation between a RAG and a canonical AG we will discuss two different ways a RAG can be translated into a canonical (but in general circular) AG: table translation and substitution translation.

3.4.1 Table translation

In *table translation*, the idea is to model references as indices into a large table, with one entry per node in the syntax tree, and where each entry contains the attributes of the respective node. This table can itself be described as an attribute and be made available throughout the syntax tree so that dereferencing a reference attribute can be replaced by indexing into the table. The table translation will lead to a circular AG, but which may still be well-defined and possible to evaluate with iterative methods. The detailed steps of the table translation are as follows.

- For each symbol X in the grammar, an attribute `id` is defined in such a way that the `id` attributes enumerate the nodes in the syntax tree in a preorder traversal. I.e., the root will have `id` = 1, its leftmost son `id` = 2, and so on. To define `id`, a help attribute `maxId` is introduced which contains the maximum `id` used in the subtree of X.
- An attribute `ct` (the "contents") is defined for each symbol X as a tuple $\langle a_1, \ldots, a_k \rangle$ where $a_1, \ldots, a_k$ are the original attributes in X. The i'th field in the tuple can be accessed by the notation `ct`(i).
- An attribute `allCt` is defined for each symbol X as an array of size $|T|$, where `allCt`$[n.\mathtt{id}] = n.\mathtt{ct}$ for any node n in the syntax tree T. To define `allCt`, array slices are collected bottom up using a synthesized attribute `subCt`. The `allCt` attribute is equal to `subCt` of the root, and that value is propagated down to each node using inherited `allCt` attributes.
- Each reference attribute r is replaced by an integer attribute r.
- In semantic rules, an access to a symbol X (used as a reference value) is replaced by the expression $X.\mathtt{id}$, i.e. the `id` attribute of the X node.
- In semantic rules, a dereferencing expression $r.a$, where r is a reference denoting a node of nonterminal X and a is an attribute of the denoted node, is replaced by the expression `allCt`$[r](i)$, where a is the ith attribute of X.

While this translation is straight-forward, it introduces circular attribute dependencies which are not allowed in canonical attribute grammars. In particular, any attribute a defined using attribute dereferencing introduces a circular dependency since it depends on T, and the definition of T in turn depends on a. However, although the translated grammar is in general circular, it is well-defined (provided that the RAG is non-circular), and possible to evaluate using iterative algorithms.

Figure 3 shows the specification of TINY, translated by table translation to canonical AG form. Figure 4 shows the resulting syntax tree and its attribution solution (some values are left out for brevity). The dereferencing of the

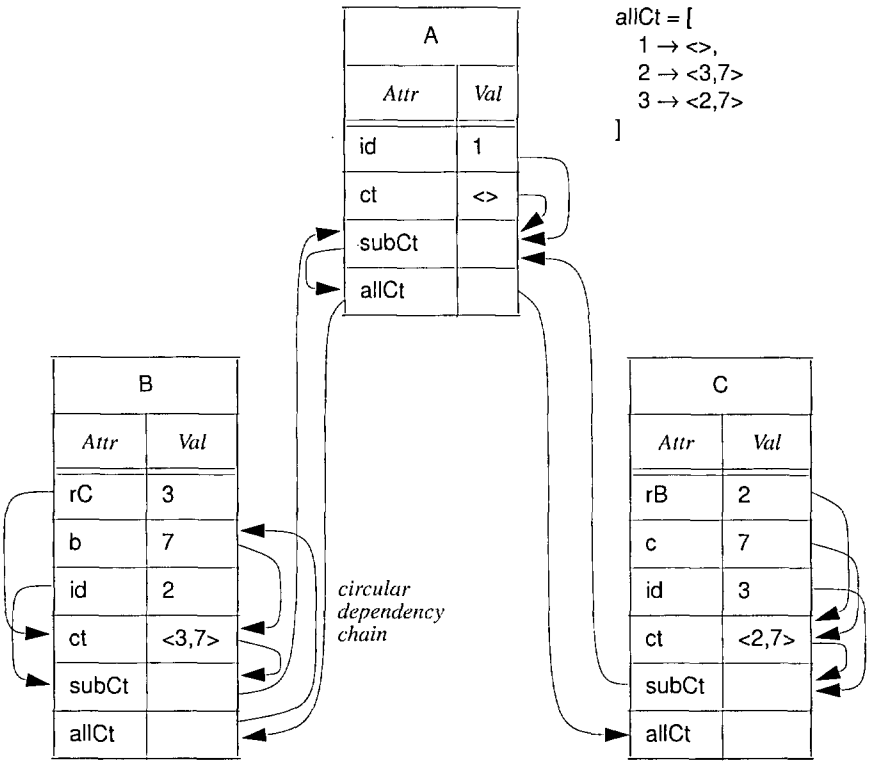


Figure 4: Attribution of TINY for table-translated specification (circular)

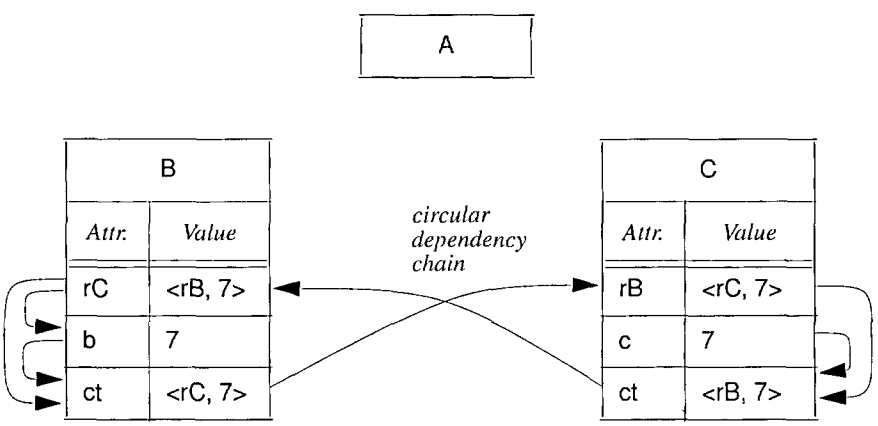


Figure 5: Attribution of TINY for substition-translated specification (circular). Attributes `rC` and `rB` have infinite attribute values

reference attribute `rC` leads to a circular dependency chain. However, the grammar is well-defined: a unique solution has been found for the tree.

3.4.2 The substitution translation

An alternative to the table translation is to translate RAGs by replacing each reference attribute by the corresponding `ct` attribute, i.e. the tuple containing the attributes of the denoted syntax node. In this translation, the `allCt` attribute is not needed. We refer to this translation method as the *substitution translation*. The problem with this method is that if a reference attribute is part of a circular data structure, it will have an infinite value in the translated canonical AG, and also give rise to a circular dependency chain. Figure 5 shows the attribution for TINY for such a translation. We might consider a refinement of this method where `ct` would include only the subset of attributes that are accessed via references. For TINY, such a translation would yield a non-circular canonical AG. However, there are other non-circular RAGs for which such a refinement will still produce a circular AG with infinite attribute values. Consider, e.g., extending `C` with an attribute `d = rB.b`.

4 Object-oriented features of RAGs

In this section, we will introduce some features of RAGs which make specifications more concise. These features are based on an object-oriented view of attribute grammars, where nonterminals are viewed as superclasses and productions as subclasses. In particular, we will discuss the use of virtual function attributes and an extended class hierarchy of nonterminals.

4.1 Virtual function attributes

Canonical AGs have a straight-forward translation to object-oriented programming [13]. In particular, a synthesized attribute is equivalent to a parameterless virtual function: The declaration of a synthesized attribute a of a nonterminal X is modelled by a declaration of a virtual function $a()$ in a class X; and a semantic rule defining a in a production p is modelled by a virtual function implementation in a class p which is a subclass of X.

With this view, it is close at hand to make a generalization: to allow virtual functions *with* parameters. However, for a canonical AG, such a generalization is not necessary. This is because the number of accesses to an attribute is always bounded, so if parameters are desired, they can be modelled by inherited attributes. For RAGs, the situation is different. Because of the reference attributes, there may be an unbounded number of accesses to a given attribute. For example, in a typical RAG an identifier use site has a reference attribute denoting the appropriate declaration node. Since a declaration can be used in an unbounded number of places in the syntax tree, the number of references to a given declaration node, and thereby also the number of accesses to attributes in the declaration node, is not bounded by the grammar. In RAGs, parameters to virtual functions can therefore not be modelled by inherited attributes.

We therefore generalize synthesized attributes by allowing nonterminals to have *virtual function attributes*. A virtual function attribute $v(b_1, \ldots, b_k)$ of a nonterminal X_0, is similar to a synthesized attribute in that it must be defined by a semantic rule of each production $X_0 \rightarrow X_1 \ldots X_n$. A semantic rule for $v(b_1, \ldots, b_k)$ is written $v(b_1, \ldots, b_k) = f(b_1, \ldots, b_k, a_1, \ldots, a_m)$, where $a_i, 1 \leq i \leq m$, is an attribute of $X_j, 1 \leq j \leq n$. From this we see that a parameterless virtual function attribute $w()$ is equivalent to a synthesized attribute.

It is possible to eliminate virtual function attributes and replace them by auxiliary functions. Each semantic rule defining the attribute is then replaced by an auxiliary function, and type case analysis is used at each call site to call the correct auxiliary function. This translation is analogous to translating object-oriented programs to procedural programs. Thus, virtual function attributes are not strictly necessary. However, they make the grammar more modular and easy to extend and change, by allowing the call site expressions to be written in a polymorphic way (being able to handle objects of different types without having to mention these types explicitly).

4.2 Extended class hierarchy

The object-oriented view on attribute grammars gives a two-level class hierarchy where nonterminals are viewed as superclasses, i.e. general concepts, and productions as subclasses, i.e. specialized concepts. Taking this view, it is natural to expand the class hierarchy into more levels. In doing this we differ between *abstract nonterminals* and *concrete nonterminals*. An abstract nonterminal differs from a concrete nonterminal in that it may not occur in any production and it may not have a concrete nonterminal as its superclass. Abstract nonterminals are thus irrelevant for the context-free part of the grammar. They are introduced in order to simplify the description of the attribution, allowing common behavior (in the form of attributes and semantic rules) to be factored out. They are also useful as types for reference attributes.

We make use of a rooted single-inheritance class hierarchy, i.e. each nonterminal has exactly one nonterminal as its superclass, except for the root nonterminal `ANY` which has no superclass. Each node in the syntax tree will thus be an instance of a subclass to `ANY` which models the behavior common to all nodes in the tree. The class hierarchy will thus be a tree rooted at `ANY`, with a top region of abstract nonterminals, lower subtrees of concrete nonterminals, and productions at the leaves.

Abstract nonterminals are similar to the notion of symbol inheritance in [25], but makes use of single rather than multiple inheritance. We have chosen single inheritance because we find it conceptually simpler and because we re-

place the use of multiple inheritance by composition, using so called semantic nodes as explained in Section 5.2.

To be able to refer to each class in the class hierarchy, the productions are named. If a nonterminal X has exactly one production, that production will also be named X, and both the nonterminal and production are mapped to the same class.

As a generalization of associating attributes with nonterminals and semantic rules with productions, it is possible to also associate attributes with individual productions (local attributes) and semantic rules with nonterminals. A semantic rule in a nonterminal constitutes a default definition that may be overridden by a semantic rule defining the same attribute in a subclass (production or other nonterminal). This notion of overriding is analogous to overriding of virtual functions in object-oriented programming languages.

In order to make sure that the grammar is well-formed, a production or concrete nonterminal C_1 that has a concrete nonterminal C_2 as a superclass may not declare any inherited attributes. All the inherited attributes of C_1 must be declared further up in the class hierarchy, either in an abstract nonterminal or in the topmost concrete nonterminal.

5 PicoJava—an example

To illustrate the utility of RAGs we will demonstrate how name and type analysis can be defined for an object-oriented language. From the point of view of this analysis, our demonstration language PicoJava, a small subset of Java [1], includes the major features of an object-oriented programming language: classes, inheritance, variables, qualified access, and reference assignment. For brevity, methods are omitted but the language allows nested class definitions [28, 37] and global variables, in order to show the combination of block structure and inheritance. The goal of the name analysis is to define a reference attribute `decl` of each identifier use site, which denotes the corresponding declaration. The goal of the type analysis is to define an attribute `tp` modelling the type of each expression. We also show how type compatibility for assignments can be specified, in the presence of object-oriented subtyping. The example grammar is non-circular and has been implemented in our language tool APPLAB.

5.1 Context-free grammar

Figure 6 shows the context-free grammar of PicoJava in RAG form. Some remarks about the notation: A nonterminal X appearing to the left of the table cell of another nonterminal or production C is a superclass of C. A production $p : X_0 \rightarrow X_1 \ldots X_n$ is written "$p \rightarrow X_1 \ldots X_n$" and appears to the right of the table cell for X_0. If a nonterminal X_0 has only one production, the production takes on the same name as the nonterminal, and is written simply "$\rightarrow X_1 \ldots X_n$". `ID` is a predefined nonterminal modelling an identifier. The productions for `Decls` and `Stmts` make use of a shorthand for lists. The topmost concrete nonterminal, `Program`, is the start symbol.

5.2 Semantic nodes

Several of the nonterminals in the context-free grammar have the prefix `SEM`. This is a convention for marking so called *semantic nonterminals*, i.e., nonterminals that are not motivated from the context-free syntax point of view, but from an attribution point of view. Semantic nonterminals always have only one production. Thus, by including a semantic nonterminal S on the right hand side of a production p, a corresponding p-node will get an extra S node as a son, a so called *semantic node*. As an example, the production `ClassDecl` has a right hand side starting with `ID SuperOpt Block`, as one would expect, modelling the name of the class, an optional superclass, and a block consisting of declarations and statements. The production continues with two semantic nonterminals: `SEMClassStaticEnv SEMClassClassEnv`. These latter two nonterminals have only one production each, and a `ClassDecl` node in the syntax tree will thus always have two extra sons of type `SEMClassStaticEnv` and `SEMClassClassEnv`, respectively. Rather than locating all attributes relevant to class declarations directly in `ClassDecl`, some attributes with a specific purpose can be packaged into a separate semantic nonterminal, e.g. `SEMClassStaticEnv`. This technique allows an ordinary node to be provided with several interfaces. A reference attribute r can be defined to denote either the `ClassDecl` node directly, or one of its semantic nodes, depending on what part of the information is relevant to the clients of r. This technique is somewhat similar to the use of *part objects* in object-oriented programming [29], where parts of the behavior of an object are delegated to a separate object, that nevertheless forms an integral part of the original object.

5.2.1 Constant semantic nodes

When reference attributes are used, it may be the case that an appropriate "real" node cannot be found in the syntax tree. For instance, suppose there is a use of an identifier x in a PicoJava program, but no corresponding declaration. In this case, there is no `Decl` node that the `decl` attribute of the use site can denote. One solution could be to give the `decl` attribute the special value `null`, denoting no node. However, it is often a nicer design to avoid `null` and instead make use of constant "null objects" [39]. In this case, we introduce a constant node `SEMMissingDecl`, modelling a missing declaration. This allows clients of the `decl` attribute to, e.g., access the type of the `decl`, regardless of if there is a real declaration or not. The type of a missing declaration can be modelled by another "null object", the constant node `SEMUnknownType`, modelling that the type of the identifier is unknown. An abstract nonterminal `SEMDecl` is introduced as a common superclass

Abstract nonterminals		*Concrete nonterminals*		*Productions*
ANY		Program		→ Block SEMGlobalConstants SEMProgramStaticEnv
		Block		→ Decls Stmts
		Decls		→ Decl*
		Stmts		→ Stmt*
		SEMGlobalConstants		→ SEMEmptyEnv SEMUnknownType
	SEMEnv	SEMEmptyEnv		→
		SEMProgramStaticEnv		→
		SEMClassStaticEnv		→
		SEMClassClassEnv		→
	SEMType	SEMUnknownType		→
		DeclType		RefDeclType: → UnQualUse
				IntDeclType: →
	SEMDecl	SEMMissingDecl		→
		Decl		ClassDecl: → 'class' ID SuperOpt '{' Block '}' SEMClassStaticEnv SEMClassClassEnv
				VarDecl: → DeclType ID
		Stmt		AssignStmt: → Use '=' Exp
				WhileStmt: → 'while' Exp 'do' Stmt
		Exp	Use	UnQualUse: → ID
				QualUse: → Use '.' UnQualUse
		SuperOpt		Super: → 'extends' Use
				NoSuper: →

Figure 6: Context-free syntax for PicoJava

to `Decl` and `SEMMissingDecl` in order to be used as the type for the `decl` attribute. The same pattern is used for `SEMUnknownType`, where `SEMType` is introduced as a common superclass of `DeclType` and `SEMUnknownType`.

5.2.2 Global access to constant nodes

In many cases, it is useful to make the constant nodes globally accessible, i.e., throughout the syntax tree. This is accomplished by collecting all constant nodes under a semantic nonterminal `SEMGlobalConstants` which is made a semantic node under the start symbol `Program`. A reference to the `SEMGlobalConstants` node is propagated down throughout the syntax tree, thus giving access to all the constant nodes. Figure 7 shows how this can be done conveniently by defining a default semantic rule in the abstract nonterminal `ANY` which is overridden in `Program`. The semantic rule in `ANY` propagates the value of its inherited `globals` attribute down to all its son nodes of type `ANY`. Since this holds for all nodes (except for the root `Program` node which overrides the rule), the reference is propagated down throughout the syntax tree. The overriding rule in `Program` instead defines `globals` of its son nodes as denoting the `SEMGlobalConstants` son node of the `Program` node. Note that we permit inherited attributes of the start symbol as long as they are not accessed. In this case, `Program` has an inherited attribute `globals` since it is a subclass of `ANY`, but this attribute is never accessed for `Program` nodes since `Program` overrides the rule in `ANY`.

Non-terminal	*Attributes*	*Semantic rules*
ANY	↓globals: SEMGlobalConstants	ANY*.globals = globals
Program		ANY*.globals = SEMGlobalConstants

Figure 7: Specification of the propagation of a reference to global constants

Remarks about the notation. In Figure 7, the sub/superclass relationships between nonterminals and productions are not shown. Please refer to Figure 6 for these relation-

Nonterminals and productions	*Attributes and Semantic Rules*
Decl	↑name: **string**
ClassDecl	name = ID.val
VarDecl	name = ID.val
Exp	↑tp: SEMType
Use	↑decl: SEMDecl
ClassDecl	↑isCircular: **boolean**

Figure 8: Module declaring `name`, `tp`, `decl`, and `isCircular`

Nonterminals/ Productions	*Attributes and Semantic Rules*
ANY	↓env: SEMEnv
SEMEnv	SEMDecl **func** lookup(str: **string**)
Decls	↑decldict: **dictionary** (string → Decl) = {(d.name → d) | d ∈ Decl* ∧ (d.name ∉ {d'.name | d'∈ Decl* - {d}}) }
Block	SEMDecl **func** lookup(str: **string**) = **inspect** $D := Decls.decldict(str) **when** Decl **do** $D **otherwise** globals.SEMMissingDecl

Figure 9: Module declaring `env` and `lookup`

ships. In semantic rules, an attribute a of the left hand side nonterminal (or the production) is written without any qualifying name, i.e. simply "a", whereas an attribute b of a nonterminal X of the right-hand side is written "$X.b$". A semantic rule $X*.b = exp$ means that the b attribute of each right-hand side nonterminal of type X is defined to have the value exp. The keyword `ref` that we used in Section 3.2 is left out here. Any attribute declared with a nonterminal type is assumed to be a reference.

5.3 Modularization

In PicoJava, name and type analysis are dependent on each other. For example, in order to find the type of a use site, we first need to know its declaration, and in order to find the declaration of a qualified use site, we need to first know the type of the qualifying use site. In order to modularize the definition of this attribution, we first define an interface module consisting of the attributes declared in Figure 8. The `Decl.name` attribute is simply the name of a `Decl` node, and the definition of this attribute is so simple that it is given directly in the figure. The definitions of the other three attributes are a bit more complex and are therefore given in separate modules, making use of the attributes in the interface module. The `Exp.tp` attribute is a reference to the `SEMType` node modelling the type for the expression. For expressions where the type is unknown, e.g. uses of undeclared names, the constant node `SEMUnknownType` is used. The `Use.decl` attribute is a reference to a `SEMDecl` node. For declared names, this will be the corresponding `Decl` node, and for undeclared or multiply declared names it will be the constant node `SEMMissingDecl`. The `ClassDecl.isCircular` attribute is a boolean attribute which is *true* if the `ClassDecl` is part of a circularly defined class hierarchy (which is illegal in PicoJava, but cannot be ruled out by the context-free syntax), and *false* otherwise (the normal case). In the following sections, these attributes are defined.

5.4 Name analysis

The goal of the *name analysis* module is to define the `Use.decl` attribute. The key idea for doing this is to define data structures, constituting of syntax tree nodes and reference attributes, to support the scope rules of PicoJava. For each block-like construct in the language, an attribute `decldict` containing a dictionary of references to the `Decl` nodes for local declarations is defined, excluding references to multiply declared identifiers. The blocks are connected to each other so that the declaration of an identifier can be located by doing lookups in block dictionaries in an appropriate order. For Algol-like block structure, a block is connected by a reference attribute to its outer block. For object-oriented inheritance, a class node is connected by a reference attribute to its superclass node. Semantic nodes that are subclasses of the abstract nonterminal `SEMEnv` encapsulate these connections and define the function attribute `lookup` for finding a `Decl` node for a given identifier. For each node n in the syntax tree, an attribute `env` is defined which refers to a `SEMEnv` node that connects to the visible identifiers at the point of n. The declaration for a `Use` can be found by calling the `lookup` function in `Use.env`. The attribute `env` thus represents the environment of visible identifiers, similar to the common solution used in canonical attribute grammars, but here `env` is a reference to a node, possibly connecting to other nodes, rather than a large aggregate attribute.

Figure 9 shows the declaration of `ANY.env`, the `lookup` function of `SEMEnv`, and the definition of `decldict`. Actually, `decldict` is an attribute of the `Decls` node, but is accessed via the function `lookup` in `Block` which returns the constant node `SEMMissingDecl` in case no declaration was found in `decldict`.

Remarks about the notation. The definition of `Block.lookup` makes use of an inspect-expression "`inspect $V:= exp...`", which is similar to a let-expression, but in addition performs a type case analysis. Within each case "when T do exp" the named value V is guaranteed to have the type T. A catch-all clause "`oth-`

Nonterminals/ Productions	*Attributes and Semantic Rules*
SEMEmptyEnv	lookup(str: **string**) = globals.SEMMissingDecl
SEMProgramStaticEnv	↑ blk: Block = **parent** Program.Block
	lookup(str: **string**) = blk.lookup(str)
SEMClassClassEnv	↑ blk: Block = **parent** ClassDecl.Block
	↑ superE: SEMEnv = **if parent** ClassDecl.isCircular **then** globals.SEMEmptyEnv **else** **parent** ClassDecl.SuperOpt.classE
	lookup(str: **string**) = **inspect** $D := blk.lookup(str) **when** Decl **do** $D **otherwise** superE.lookup(str)
SEMClassStaticEnv	↑ thisE: SEMEnv = **parent** ClassDecl.SEMClassClassEnv
	↑ outerE: SEMEnv = env
	lookup(str: **string**) = **inspect** $D := thisE.lookup(str) **when** Decl **do** $D **otherwise** outerE.lookup(str)
SuperOpt	↑ classE: SEMEnv
Super	classE = **inspect** $D := UnQualUse.decl **when** ClassDecl **do** $D.SEMClassClassEnv **otherwise** globals.SEMEmptyEnv
NoSuper	classE = globals.SEMEmptyEnv

Figure 10: Module defining `lookup`

erwise *exp*" is needed to make sure there is always an applicable case.

Figure 10 shows the definition of the `SEMEnv` connections and the `SEMEnv.lookup` function. There are two block constructs in PicoJava: `Program` containing global declarations, and `ClassDecl`, containing declarations local to a class. Algol-like block structure is obtained by nesting a class inside another class. `Program` has a single semantic node `SEMProgramStaticEnv` connecting to the `Block` of the `Program` (`blk`). `ClassDecl` has two semantic nodes; `SEMClassClassEnv` handles inheritance by connecting to `Block` of the class (`blk`) and to the `SEMClassClassEnv` of the superclass (`superE`); and `SEMClassStaticEnv` combines inheritance with Algol-like block structure by connecting to the `SEMClassClassEnv` of the class (`thisE`) and to the environment (`outerE`). Figure 11 shows these connections for an example PicoJava program. The `lookup` function in `SEMClassClassEnv` is defined to give preference to local declarations over those in the superclass (a declaration in the class will shadow declarations of the same name in superclasses). The `lookup` function in `SEMClassStaticEnv` is defined to give preference to inheritance over block structure (a declaration in a superclass will shadow declarations of the same name in an outer block).

Nonterminals/ Productions	*Attributes and Semantic Rules*
ANY	ANY*.env = env
Program	Block.env = SEMProgramStaticEnv
ClassDecl	Block.env = SEMClassStaticEnv
QualUse	UnQualUse.env = **inspect** $T := Use.tp **when** RefDeclType **do** **inspect** $D := $T.UnQualUse.decl **when** ClassDecl **do** $D.SEMClassClassEnv **otherwise** globals.SEMEmptyEnv **otherwise** globals.SEMEmptyEnv

Figure 12: Module defining `env`

Remarks about the notation. The expression "`parent` *T*" is a reference denoting the parent node which must be of type *T*. This is a shorthand for using an inherited attribute `parent` defined by the parent node. To assure that this expression is always well defined, it is only applicable for nonterminals that appear on the right-hand side of exactly one production.

A PicoJava program may contain an (illegal) circular class structure. Therefore, care must be taken so that the recursively defined `lookup` function does not lead to endless recursion. To prevent this, a test on the `isCircular` attribute (declared in the interface module) is performed when defining the connections between the `SEMClassClassEnv` nodes. In case the class hierarchy is cyclic, the attribute `superE` is defined as a reference to the constant node `SEMEmptyEnv` rather than to the `SEMClassClassEnv` of the superclass. This way, the graph consisting of `SEMClassClassEnv` nodes and `superE` attributes can never be cyclic, and their `lookup` functions will therefore terminate.

Figure 12 shows the definition of `env`. For most nodes, the environment is the same as for the enclosing node, as defined by the default semantic rule in `ANY`. This default behavior is overridden in three productions. In `Program` and `ClassDecl`, the environment for the `Block` is defined as a reference to the `SEMProgramStaticEnv` and `SEMClassStaticEnv`, respectively. In the `QualUse` production, the environment of the second operand depends on the type of the first operand which should be a reference variable.

The definition of the `decl` attribute is now simple, as shown in Figure 13.

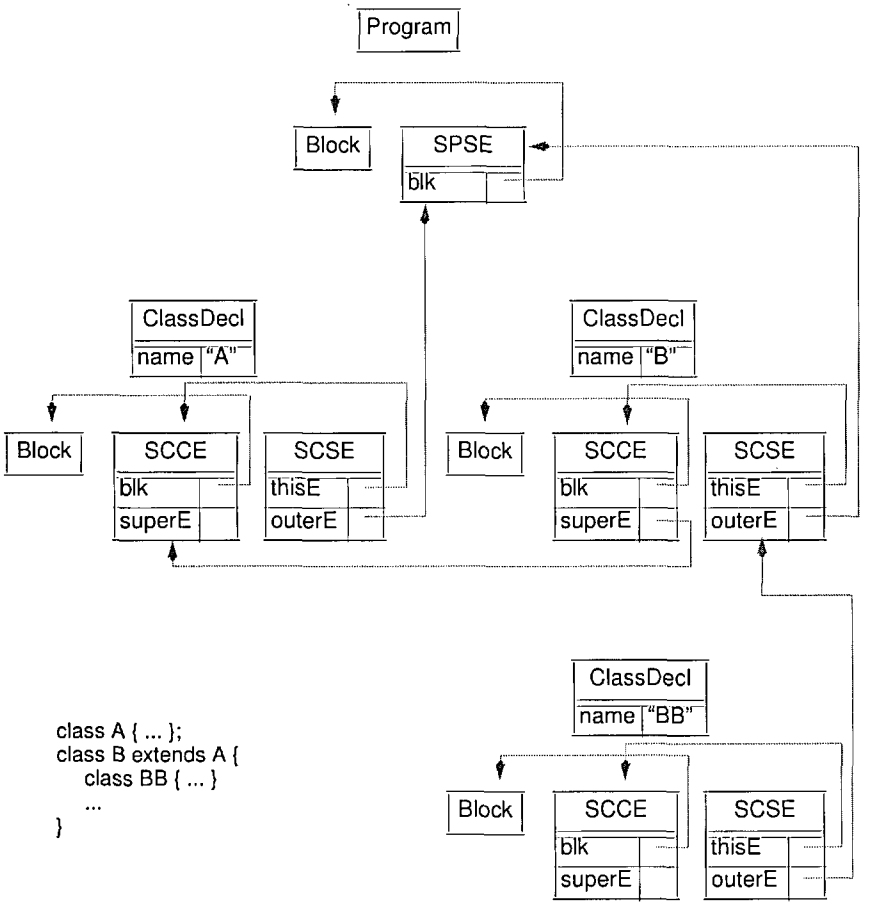


Figure 11: Connections between SEMEnv nodes for a small program

Nonterminals/ Productions	*Attributes and Semantic Rules*
UnQualUse	decl = env.lookup(ID.val)
QualUse	decl = UnQualUse.decl

Figure 13: Module defining decl

Nonterminals/ Productions	*Attributes and Semantic Rules*
ClassDecl	isCircular = SuperOpt.circularClass({**self**})
	boolean func circularClass (s: **set of** ClassDecl) = **if self** ∈ s **then true** **else** SuperOpt.CircularClass(s ∪ {**self**})
SuperOpt	**boolean func** circularClass (s: **set of** ClassDecl)
NoSuper	circularClass(s: **set of** ClassDecl) = **false**
Super	circularClass(s: **set of** ClassDecl) = **inspect** $D := UnQualUse.decl **when** ClassDecl **do** $D.circularClass(s) **otherwise false**

Figure 14: Module defining isCircular

5.5 Check of circular class hierarchy

Figure 14 shows the definition of the isCircular attribute declared in Figure 8 which says if a class is circularly defined or not. The idea is to use a help function circularClass(s) which is called recursively for each ClassDecl in the superclass chain. The argument s contains the set of references to already visited ClassDecl nodes. The recursion is terminated either when the top of the class hierarchy is reached (the normal case), or when a ClassNode is reached that is already in s (a cycle is found in the hierarchy).

Remark on the notation. The construct "self" in a rule means a reference to the left-hand nonterminal of the production. E.g., in Figure 14, self refers to the ClassDecl node.

5.6 Type analysis

Figure 15 shows the definition of the `tp` attribute declared in Figure 8. For illegal uses of identifiers, e.g. where the declaration is missing, the constant node `SEMUnknownType` is used.

Nonterminals/ Productions	*Attributes and Semantic Rules*
Use	tp = **inspect** $D := decl **when** VarDecl **do** $D.DeclType **otherwise** globals.SEMUnknownType
QualUse	tp = UnQualUse.tp

Figure 15: Module defining `tp`

The `tp` attribute can be used to perform type checking, e.g., checking that the types of the left and right hand side of an assignment are compatible. For an object-oriented language, this check is rather more involved than for procedural languages, due to the subtype compatibility rules. For a reference assignment *Use = Exp* in PicoJava, the class of *Exp* must be the same or a subclass of the class of *Use*. To further show the expressiveness of RAGs, Figure 16 shows how a boolean attribute `typesCompatible` can be defined for `Assignment`, taking into account both ordinary types and reference types with subtyping. The `typesCompatible` attribute is *true* if the assignment statement is type correct. A help function `assignableTo` is defined in `SEMType` such that `T1.assignableTo(T2)` is *true* if it is legal to assign a value of type `T1` to a variable of type `T2`. For reference types (`RefDeclType`), this function checks if the class of `T1` is a subclass of that of `T2`. To perform this check, the class hierarchy is traversed using a recursive function `recSubclassOf` in `ClassDecl`. However, in order to make sure that this function terminates, even in the case of an illegal circular class hierarchy, the attribute `isCircular` is checked before calling the recursive function (in `ClassDecl.subclassOf`).

Nonterminals/ Productions	*Attributes and Semantic Rules*
SEMType	**boolean func** assignableTo(T: SEMType)
SEMUnknownType	assignableTo(T: SEMType) = **false**
IntDeclType	assignableTo(T: SEMType) = T **in** IntDeclType
RefDeclType	assignableTo(T: SEMType) = **inspect** $T := T **when** RefDeclType **do** **inspect** $D := UnQualUse.decl **when** ClassDecl **do** **inspect** $DT := $T.UnQualUse.decl **when** ClassDecl **do** $D.subclassOf($DT) **otherwise false** **otherwise false** **otherwise false**
ClassDecl	**boolean func** subclassOf(C: ClassDecl) = **if** isCircular **then false** **else** recSubclassOf(C)
	boolean func recSubclassOf(C: ClassDecl) = **if** C = **self** **then true** **else** **inspect** $Super := SuperOpt.superClass **when** ClassDecl **do** $Super.recSubclassOf(C) **otherwise false**
AssignStmt	↑typesCompatible: **boolean** = Exp.tp.assignableTo(Use.tp)
SuperOpt	↑superClass: ClassDecl
NoSuper	superClass = **null**
Super	superClass = **inspect** $D := SimpleUse.decl **when** ClassDecl **do** $D **otherwise null**

Figure 16: Module defining `typesCompatible`

6 Experimental system

We have implemented RAGs in our language tool APPLAB and used RAGs to specify a number of languages, including an extended version of PicoJava described in Section 5 (the extended version includes also methods and some additional basic types, operators, and statements). We are also working with specification of worst-case execution time analysis [31, 32], robot languages [7], state transition languages [11], visualization [30], design patterns [8, 9], and the RAG formalism itself.

The APPLAB system is an interactive language tool where both programs and grammars for the programming language can be edited at the same time, resulting in a highly flexible and interactive environment for language design. Changes to the grammars, e.g. changes to the context-free syntax or changes to the attributes and semantic functions, are immediately reflected in the language-based program editor, allowing the user to get immediate feedback on the effects of changes to the grammar specification. The details of APPLAB are covered in [6, 7] (although these papers do not focus on reference attributes which is a later addition).

Figure 17 shows a screendump from the APPLAB system, showing the editing of an example program in PicoJava, and parts of the grammar specification. In the `ExampleProgram` window, the user has selected the statement `g=rB` in class `BB`, where `BB` is an inner class of `B` which in turn is a subclass of `A`. The example illustrates both block structure (`g` is declared globally, i.e. two levels outside of

BB) and combined block structure and inheritance (rB is declared one level outside of BB in a superclass of B). The assignment is type correct (the value of typesCompatible is TRUE) since B (the class of rB) is a subclass of A (the class of g). The value of the attribute is shown in a separate attribute window at the user's request (after selecting the attribute in a popup-menu). The subsequent assignment rB=g is not type correct since A (the class of g) is not equal to or a subclass of B (the class of rB), and a request for the typesCompatible attribute of that statement would display a corresponding attribute window showing that typesCompatible has the value FALSE.

7 Related work

The idea to support non-local dependencies has been suggested in a number of systems in various ways. Early approaches provided special support for nested scopes (supporting Algol-like block structure) such as [19, 20, 3, 23, 17, 2], but fail to handle more complex scope combinations such as inheritance or qualified access of identifiers. Later approaches support explicit reference attributes and remote attribute access, in a similar way as described here, and allows scope mechanisms to be defined without being restricted to predefined combinations. In particular:

- In our previous work on Door Attribute Grammars [14, 15, 16] dereferencing of reference attributes is supported, but must be delegated to special nonterminals called doors. This way, the non-local dependencies are encapsulated in a so called door package. Door AGs also support remote definition where collection values can be defined remotely via references. Door AGs support efficient incremental attribute evaluation, but the implementation is not fully automatic because the door package needs to be implemented manually. Door AGs allows object-oriented languages to be specified in a way very similar to for RAGs, using similar techniques for connecting environments and traversing inheritance graphs, but RAGs are considerably more compact because the non-locally accessed information does not need to be propagated to door nonterminals, but can be accessed directly, thus avoiding replication of information. RAGs offer fully automatic evaluation, but not (currently) incremental attribute evaluation.

- The MAX system by Poetzsch-Heffter [33, 34] supports reference attributes and remote access, and develops an extension to term algebras called *occurrence algebras* to formalize the approach. A demand-based evaluation technique is used, and in addition an approximate static dependency analysis is developed which allows many function calls to be eliminated and thereby speed up the evaluation [34].

- Boyland also developed a system supporting both remote access and remote definition, and making use of a demand-algorithm for attribute evaluation [4]. He has also addressed the problem of computing static evaluation schemes for grammars with both remote access and remote definition via reference attributes in order to apply visit-oriented evaluation algorithms. However, the scope of this latter technique is unclear. It has been applied only for simple example grammars and does not seem to be implemented [5].

- Sasaki and Sassa have developed a static evaluation scheme for circular grammars with reference attributes and remote access [36]. Their motivating example is liveness analysis in the presence of gotos where the goto links are modelled by reference attributes in the AST. In their evaluation scheme remote dependencies are added conservatively, causing cycles in the production dependency graphs that correspond to real or potential cycles in an actual tree. Cycles are evaluated iteratively.

The underlying principles of remote access and attribute evaluation are the same in RAGs as in MAX and in Boyland's system. However, the RAG formulation is radically different, expressing the specification using object-oriented concepts like inheritance and virtuals.

Other related approaches include the following:

- The Synthesizer Generator supports syntactic references, i.e., an attribute may be a reference to a syntax tree node [35]. However, attributes of the referenced node may not be accessed via the reference attribute. I.e., the syntactic references are considered to stand for unattributed subtrees. There are certain similarities to RAGs in that the syntax tree can itself be used as e.g. symbol tables, rather than having to construct such information in a separate attribute domain. However, RAG reference attributes are much more powerful than syntactic references in that the attributes of the referenced nodes may be accessed, allowing attribute information to be propagated along non-locally paths. The Synthesizer Generator also allows attributes to be defined as references to other attributes. This is used to define cyclic graphs in code generation, e.g. for linking the last instruction of a while statement back to the first instruction. However, for the purpose of the attribute evaluation, these references are just treated as constants and may not be dereferenced. Dereferencing can only be done after the attribution is complete, by an interpreter written directly in C.

- The Elegant system [2] also supports the construction of a cyclic program construct graph which is essentially the syntax tree extended with edges from use sites to declaration sites. However, the additional edges cannot be dereferenced in order to define other attributes. They may, however, be dereferenced after the attribution is complete, in order to check context conditions. The resulting program construct graph can

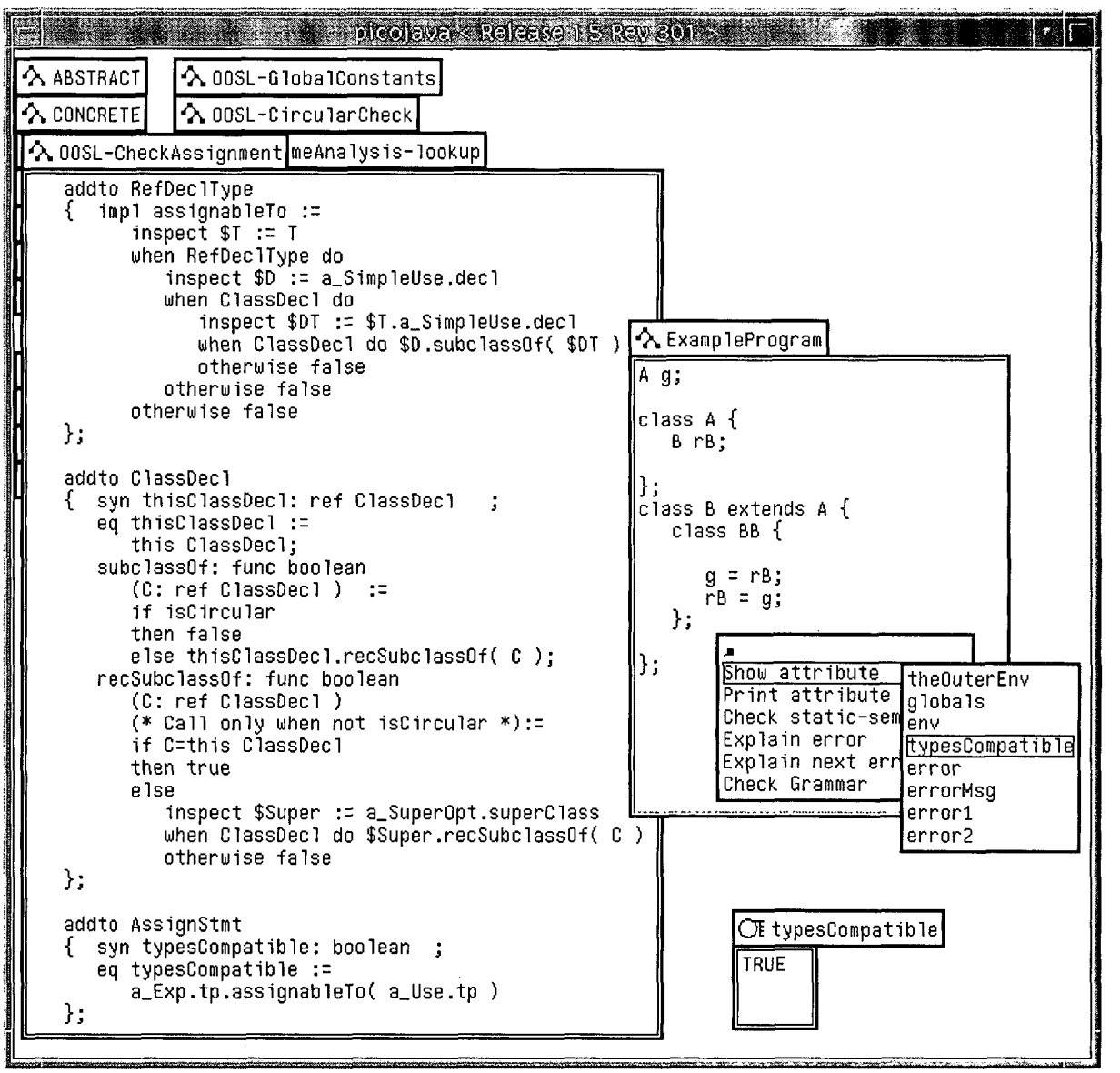


Figure 17: Screendump from APPLAB. The attribute `typesCompatible` is shown for the current focus in the `ExampleProgram` window (the assignment statement `g = rB` in class `BB`)

also be processed by a special-purpose code generation formalism.

- Vorthmann has developed a graphical technique called *visibility networks* for describing name analysis and use-declaration bindings in programming languages, and exemplified the technique for Ada [38]. The focus is on providing efficient incremental evaluation. This technique might be interesting to integrate with RAGs in order to provide support for incremental attribute evaluation for certain classes of RAGs.

8 Conclusions

We have presented Reference Attributed Grammars (RAGs) and showed how they can be applied to an advanced problem: name and type analysis for an object-oriented language, yielding a simple and concise non-circular specification. Figures 6–10 and 12–16 constitute a complete static-semantic specification of PicoJava, a language with all the key object-oriented constructs: block structure, classes (including inner or nested classes), inheritance, qualified use, and assignment compatibility in the presence of subtyping.

The use of reference attributes allows cyclic structures to be constructed on top of the syntax tree substrate. We have demonstrated how attributes can be used to check for such cyclic structures to ensure that semantic functions terminate, thus allowing the RAG to remain non-circular, although it works on cyclic structures. (See the definition and use of the `ClassDecl.isCircular` attribute in Section 5.)

We have implemented the RAG formalism and an evaluation algorithm that can handle any non-circular RAG. In our tool for language experimentation, APPLAB, it is possible to experiment with RAG specifications and immediately try out changes to the attribution rules, e.g. by asking for the values of attributes in an example program.

We have demonstrated advantages of RAGs over canonical AGs. First, there is no need in RAGs to replicate the information available in the syntax tree into attributes. By using reference attributes the syntax tree itself can be used as the information source. The syntax nodes can be connected using reference attributes to form suitable data structures, also cyclic ones, without the need for introducing data structures and functions in auxiliary languages. Second, the semantic functions working on a complex data structure can be split into smaller functions, delegated to the different syntax nodes making up the data structure, and specified completely within the RAG formalism. Third, it is easy to extend an existing grammar with additional functionality. This was shown in the PicoJava example where the test for type compatibility of assignments was added in a very concise way, although it included advanced rules for subtype compatibility.

In our experience, RAGs are of immediate practical use and we have a number of current projects concerning language specification using this technique. There are many interesting areas for further research, including the following.

- Efficient incremental evaluation of RAGs is an open problem. However, RAGs are a much better starting point for incremental evaluation than canonical AGs since large aggregate attributes are not needed in RAGs, and the number of affected attributes after a change is much lower than for a canonical AG.

- It would be useful to develop algorithms for deciding statically if a RAG is non-circular. This is an open problem. The APPLAB system currently tests circularity dynamically and reports circular dependencies at evaluation time.

- It would be useful to develop algorithms for deciding if a RAG contains nonterminating semantic functions. In the PicoJava example there are two cases where special care is taken in order to make sure that the semantic functions terminate, namely when using recursive functions that traverse the class hierarchy. The attribute `isCircular` was introduced in order to be able to terminate the recursion in case of a cyclic class hierarchy. During grammar development it would be useful if potential circular structures and nonterminating functions could be automatically spotted by the system.

- The formalism should be extended so that semantic nonterminals and nodes can be added in extension modules, i.e. without having to modify the context-free syntax. We expect this to be straight-forward, making use of object-oriented concepts like part objects and inner (anonymous) classes as available in BETA and recently also in Java [28, 29, 37].

- Since RAGs allow arbitrary data structures to be built using syntax tree nodes and references it should be interesting to extend the technique to allow graph-based grammars, working on syntax graphs rather than trees. This would be relevant for building language-based editors for, e.g., UML class diagrams or state-transition diagrams.

Acknowledgements

This work was supported by NUTEK, the Swedish National Board for Industrial and Technical Development. Elizabeth Bjarnason implemented the major parts of the APPLAB system, making it easy to add the support for reference attributes.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley. 1996.

[2] L. Augusteijn. The Elegant Compiler Generator System. In *Attribute Grammars and their Applications*, pp 238–254, LNCS 461, Springer-Verlag, September 1990.

[3] G. M. Beshers and R. H. Campbell. Maintained and constructor attributes. In Proceedings of the *SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pp 34–42, Seattle, Wa., 1985. ACM SIGPLAN Notices, 20(7).

[4] J. T. Boyland. *Descriptional Composition of Compiler Components*. PhD thesis, Report No. UCB//CSD-96-916, Computer Science Division (EECS), University of California, Berkeley, California, September 1996.

[5] J. Boyland. Analyzing Direct Non-Local Dependencies in Attribute Grammars. In *Proceedings of CC '98: International Conference on Compiler Construction*, pp 31–49, LNCS 1383, Springer-Verlag, 1998.

[6] E. Bjarnason. *Interactive Tool Support for Domain-Specific Languages*. Licentiate thesis. Dept. of Computer Science, Lund University, December 1997.

[7] E. Bjarnason, G. Hedin, K. Nilsson. Interactive Language Development for Embedded Systems. *Nordic Journal of Computing* 6(1999), 36–55.

[8] A. Cornils and G. Hedin. Statically Checked Documentation with Design Patterns. In *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS Europe 2000)*, St. Malo, France, June 2000. IEEE.

[9] A. Cornils and G. Hedin. Tool Support for Design Patterns using Reference Attributed Grammars. In D. Parigot and M. Mernik, editors, *Third Workshop on Attribute Grammars and their Applications, WAGA'2000*, Ponte de Lima, Portugal, July 2000.

[10] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp 85–98, Palo Alto, Ca., July 1986. ACM SIGPLAN Notices, 21(7).

[11] S. Gestegård. *Emulation Software for Executable Specifications*. Master's thesis. LU-CS-EX:99-6. Dept. of Computer Science, Lund University, Sweden, April 1999.

[12] G. Hedin. Incremental Attribute Evaluation with Side-effects. In D. Hammer, editor, *Compiler Compilers and High Speed Compilation (2nd CCHSC Workshop)*, pp 175–189, Berlin, GDR, October 1988, LNCS 371, Springer-Verlag.

[13] G. Hedin. An object-oriented notation for attribute grammars. *ECOOP'89*. BCS Workshop Series, pp 329–345, Cambridge University Press. 1989.

[14] G. Hedin. Incremental static-semantic analysis for object-oriented languages using Door attribute grammars. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems*, pp 374–379, Prague, June 1991, LNCS 545, Springer-Verlag.

[15] G. Hedin. *Incremental Semantic Analysis*. PhD thesis, Department of Computer Science, Lund University, Sweden, March 1992.

[16] G. Hedin. An overview of door attribute grammars. *International Conference on Compiler Construction (CC'94)*. LNCS 786, Springer Verlag. 1994.

[17] R. Hoover. *Incremental Graph Evaluation*. PhD thesis, Cornell University, Ithaca, N.Y., May 1987. Tech. Rep. 87-836.

[18] F. Jalili. A general linear time evaluator for attribute grammars. *ACM SIGPLAN Notices*, Vol 18(9):35–44, September 1983.

[19] G. F. Johnson and C. N. Fischer. Non-syntactic attribute flow in language based editors. In *Proc. 9th POPL*, pp 185–195, Albuquerque, N.M., January 1982. ACM.

[20] G. F. Johnson and C. N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *Proc. 12th POPL*, pp 141–151, New Orleans, La., January 1985. ACM.

[21] L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM TOPLAS*, 12(3):429–462, 1990.

[22] M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In M. Paul and B. Robinet, editors, *International Symposium on Programming, 6th Colloquium*, LNCS 167, pp 167–178. Springer-Verlag, 1984.

[23] G. Kaiser. *Semantics for Structure Editing Environments*. PhD thesis, Carnegie-Mellon University, Pittsburgh, Pa., May 1985. CMU-CS-85-131.

[24] U. Kastens. Ordered attributed grammars. *Acta Informatica*, 13:229–256, 1980.

[25] U. Kastens and W. M. Waite. Modularity and Reusability in Attribute Grammars. *Acta Informatica*, 31:601–627, 1994.

[26] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2), 127–145, June 1968.

[27] O. L. Madsen. On defining semantics by means of extended attribute grammars. In *Semantics-Directed Compiler Generation*, pp 259–299, LNCS 94, Springer-Verlag, January 1980.

[28] O. L. Madsen. Block structure and object-oriented languages. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. The MIT Press, 1987.

[29] O. L. Madsen and B. Møller-Pedersen. Part objects and their location. Proceedings of the *7th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 7)*. Dortmund, 1992. Prentice Hall.

[30] E. Magnusson and G. Hedin. Program visualization using reference attributed grammars. In *Proceedings of NWPER'2000, the 9th Nordic Workshop on Programming and Software Development Environment Research*, Bergen, Norway, 2000.

[31] P. Persson and G. Hedin. Interactive Execution Time Predictions Using Reference Attributed Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pp 173–184, Amsterdam, The Netherlands, March 1999.

[32] P. Persson and G. Hedin. An Interactive Environment for Real-Time Software Development. In *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS Europe 2000)*, St. Malo, France, June 2000. IEEE.

[33] A. Poetzsch-Heffter. Programming language specification and prototyping using the MAX System. In M. Bruynooghe, J. Penjam, editors, *Programming Language Implementation and Logic Programming*, pp 137–150. LNCS 714, Springer-Verlag, 1993.

[34] A. Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica* 34, 737–772 (1997).

[35] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator. A system for constructing language-based editors*. Springer Verlag. 1989.

[36] A. Sasaki and M. Sassa. Circular attribute grammars with remote attribute references. In D. Parigot and M. Mernik, editors, *Third Workshop on Attribute Grammars and their Applications, WAGA'2000*, Ponte de Lima, Portugal, July 2000.

[37] Sun Microsystems. *Inner Classes Specification*. 1996. URL: `http://java.sun.com/products/-jdk/1.1/docs/guide/innerclasses`

[38] S. A. Vorthmann. *Modelling and Specifying Name Visibility and Binding Semantics*. CMU-CS-93-158. Carnegie Mellon University, Pittsburgh, Pa., July 1993.

[39] B. Woolf. Null Object. In R. Martin et al. (eds), *Pattern Languages of Program Design 3*, pp 5–18, Addison Wesley, 1998.

Multiple Attribute Grammar Inheritance

Marjan Mernik, Mitja Lenič, Enis Avdičaušević and Viljem Žumer
University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova 17, 2000 Maribor, Slovenia
marjan.mernik@uni-mb.si

Keywords: object-oriented attribute grammars, inheritance, incremental language design

Edited by: Didier Parigot
Received: June 10, 1999 **Revised:** August 28, 2000 **Accepted:** September 11, 2000

The language design process should be supported by modularity and abstraction in a manner that allows incremental changes as easily as possible. To at least partially fulfill this ambitious goal a new object-oriented attribute grammar specification language which supports multiple attribute grammar inheritance is introduced. Multiple attribute grammar inheritance is a structural organization of attribute grammars where the attribute grammar inherits the specifications from ancestor attribute grammars, may add new specifications or may override some specifications from ancestor specifications. With the proposed approach a language designer has the chance to design incrementally a language or reuse some fragments from other programming language specifications. The multiple attribute grammar inheritance is first introduced using an example, and thereafter by a formal model. The proposed approach is successfully implemented in the compiler/interpreter generator tool LISA ver. 2.0.

1 Introduction

We have developed a compiler/interpreter generator tool LISA which automatically produces a compiler or an interpreter from the ordinary attribute grammar specifications [1, 2]. But in this version of the tool, incremental language development was not supported, so the language designer had to design new languages from scratch or by scavenging old specifications. Other deficiencies of ordinary attribute grammars become apparent in specifications for real programming languages. Such specifications are large and unstructured, and are hard to understand, modify and maintain. Yet worse, small modifications of some parts in the specifications have widespread effects on the other parts of the specifications. Therefore specifications are not modular, extensible and reusable. Compared to modern programming languages, such as object-oriented or functional languages, the attribute grammar specification languages are far less advanced, specifically concerning the possibilities of abstraction, modularization, extensibility and reusability. Therefore, the integration of specification languages with various programming paradigms has developed in recent years. A detailed survey of attribute grammar based specification languages is given in [3]. We applied inheritance, a characteristic feature of object-oriented programming, in attribute grammars. A new object-oriented specification language with the paradigm Attribute grammar = Class, which is not included in [3], is presented in the paper. In [4] the new concept is introduced only in the informal manner through examples of a simple calculator language. We have incrementally designed various small programming languages, such as COOL and PLM, with multiple attribute grammar inheritance. Our experience with these non-trivial examples shows that multiple inheritance in attribute grammars is useful in managing the complexity, reusability and extensibility of attribute grammars. The benefit of this approach is also that for each language increment a compiler can be generated and the language tested. In this paper the reasons for introducing multiple inheritance into attribute grammars and the formal definition of multiple attribute grammar inheritance are presented. The multiple attribute grammar inheritance approach is successfully implemented in the newly developed version of the tool LISA ver. 2.0.

2 Background

Attribute grammars have been introduced by D.E. Knuth and since then have proved to be very useful in specifying the semantics of programming languages, in automatic constructing of compilers/interpreters, in specifying and generating interactive programming environments and in many other areas. Attribute grammars [5, 6, 7] are a generalization of context-free grammars in which each symbol has an associated set of attributes that carry semantic information, and with each production a set of semantic rules with attribute computation is associated. An attribute grammar consists of:

- A context-free grammar $G = (T, N, S, P)$, where T and N are the set of terminal symbols and nonterminal symbols; $S \in N$ is the start symbol, which doesn't

appear on the right side of any production rule; and P is the set of productions. Now set $V = T \cup N$.

- A set of attributes $A(X)$ for each nonterminal symbol $X \in N$. $A(X)$ is divided into two mutually disjoint subsets, $I(X)$ of inherited attributes and $S(X)$ of synthesized attributes. Now set $A = \bigcup A(X)$. Let $Type$ denote a set of semantic domains. For each $a \in A(X)$, $a : type \in Type$ is defined which is the set of possible values of a.

- A set of semantic rules R. Semantic rules are defined within the scope of a single production. A production $p \in P, p : X_0 \rightarrow X_1 \ldots X_n$ $(n \geq 0)$ has an attribute occurrence $X_i.a$ if $a \in A(X_i)$, $0 \leq i \leq n$. A finite set of semantic rules R_p is associated with the production p with exactly one rule for each synthesized attribute occurrence $X_0.a$ and exactly one rule for each inherited attribute occurrence $X_i.a, 1 \leq i \leq n$. Thus R_p is a collection of rules of the form $X_i.a = f(y_1, \ldots, y_k), k \geq 0$, where $y_j, 1 \leq j \leq k$, is an attribute occurrence in p and f is a semantic function. In the rule $X_i.a = f(y_1, \ldots, y_k)$, the occurrence $X_i.a$ depends on each attribute occurrence $y_j, 1 \leq j \leq k$. Now set $R = \bigcup R_p$. For each production $p \in P, p : X_0 \rightarrow X_1 \ldots X_n$ $(n \geq 0)$ the set of defining occurrences of attributes is $DefAttr(p) = \{X_i.a | X_i.a = f(\ldots) \in R_p\}$. An attribute $X.a$ is called synthesized ($X.a \in S(X)$) if there exists a production $p : X \rightarrow X_1 \ldots X_n$ and $X.a \in DefAttr(p)$. It is called inherited ($X.a \in I(X)$) if there exists a production $q : Y \rightarrow X_1 \ldots X \ldots X_n$ and $X.a \in DefAttr(q)$.

Therefore, an attribute grammar is a triple $AG = (G, A, R)$ which consists of a context free grammar G, a finite set of attributes A and a finite set of semantic rules R.

3 Reasons for Introducing Multiple Inheritance into Attribute Grammars

The language design process should be supported by modularity and abstraction in a manner that allows to make incremental changes as easily as possible. This is one of the strategic directions of further research on programming languages. When introducing a new concept the designer has difficulties in integrating it into the language in an easy way. Therefore inheritance can be very helpful since it is a language mechanism that allows new definitions to be based on the existing ones. A new specification inherits the properties of its ancestors, and may introduce new properties that extend, modify or defeat its inherited properties. When a new concept is added/removed in/from a language, not only is the semantic part changed, but the syntax rules and the lexicon may also need to be modified. Therefore, such incremental modifications usually do not preserve upward language compatibility. A language designer needs a formal method which enables incremental changes and usage of specification fragments from various programming languages. We accomplish these goals by introducing the object-oriented concepts, i.e. multiple inheritance and templates, into attribute grammars [4]. Let us look at the informal definition of multiple attribute grammar inheritance and templates. Multiple attribute grammar inheritance is a structural organization of attribute grammars where the attribute grammar inherits the specifications from ancestor attribute grammars, may add new specifications, may override some specifications from ancestors or even defeat some ancestor specifications. With inheritance we can extend the lexical, syntax and semantic parts of the programming language specification. Therefore, regular definitions, production rules, attributes, semantic rules and operations on semantic domains can be inherited, specialized or overridden from ancestor specifications. The language is specified in the following manner:

```
language L1 [extends L2, ..., LN] {
  lexicon {
    [[P] overrides | [P] extends] R regular expr.
      ⋮
  }
  attributes type A1, ..., AM
    ⋮
  rule [[Y] extends | [Y] overrides] Z {
    X ::= X11 X12 ... X1p compute {
        semantic functions }
      ⋮
    |
      Xr1 Xr2 ... Xrt compute {
        semantic functions }
    ;
  }
    ⋮
  method [[N] overrides | [N] extends] M {
    operations on semantic domains
  }
  ⋮
}
```

In object-oriented languages the properties that consist of instance variables and methods are subject to modification. Since in attribute grammars semantic rules are tightly coupled with particular production rules, properties in multiple attribute grammar inheritance consist of lexical regular definitions, attribute definitions, rules which are generalized syntax rules that encapsulate semantic rules and methods on semantic domains. The benefits of multiple attribute grammar inheritance are:

- specifications are extensible since the language designer writes only new and specialized specifications,

- specifications are reusable since specifications are inherited from ancestor specifications, and

- the language designer can construct the programming language specification from multiple specifications.

In our opinion the main weakness of multiple attribute grammar inheritance approach is that it does not help the designer in the case when languages have similar semantics and a totally different syntax. One possible solution to this problem is that of Composable Attribute Grammars (CAG) [11]. CAG consists of component attribute grammars and glue grammar. In component attribute grammar phrase structure and its semantics are expressed in terms of abstract, language independent context free grammar. The concrete syntactic structure is specified only in a glue grammar. In our approach templates are introduced with a similar goal. When studying semantic specifications for various programming languages common patterns can be noticed. Patterns like value distribution, list distribution, value construction, list construction, bucket brigade, propagate value and many others are independent of the structure of production rules. Such patterns are described with templates. A template in attribute grammars is a polymorphic abstraction of a semantic rule parameterized with attribute occurrences which can be associated with many production rules with different nonterminal and terminal symbols. Since a nonterminal symbol can be considered as a class in object-oriented attribute grammars [3], a template in attribute grammars is a kind of polymorphism. Further, at template instantiation appropriate semantic rules are generated at compiler generation time which is similar to templates in object-oriented languages where the code is generated at compile time. Templates are also independent of a number of attribute occurrences which participate in semantic rules. For this purpose a variable list of arguments is proposed. As an example, a value distribution pattern is described as:

```
Y ::= X1 X2 ... XN
  { X1.in = Y.in; X2.in = Y.in; ...
    XN.in = Y.in; }
```

A template describing the value distribution pattern is:

```
template <attributes Y_in, X_in*>
  compute valueDistribution {
    { X_in* = Y_in; }
  }
```

The formal argument `X_in*` in the template `valueDistribution` is a variable list of arguments. Such arguments are denoted with an asterisk after the name. At template instantiation a part of semantic rules enclosed with braces is generated for each argument in the variable list. Together with a variable list of arguments some functions are defined which can be used for variable list manipulation (`first, last, succ, pred`). A successor for the last argument and a predecessor for the first argument do not exist. The usage of the above functions is presented in the next example. A pattern bucket brigade left is described as:

```
Y ::= X1 X2 ... XN
  { X1.in = Y.in; X2.in = X1.out;
      ⋮
    XN.in = XN-1.out; Y.out = XN.out; }
```

A template describing the pattern bucket brigade left is:

```
template <attributes Y_in, Y_out, X_in*,
  X_out*>
  compute bucketBrigadeLeft {
    if (empty(X_in*) && empty(X_out*))
      Y_out = Y_in;
    else
      first(X_in*) = Y_in;
      { X_in* = pred(X_out*);}
      Y_out = last(X_out*);
    endif
  }
```

One of the drawbacks of attribute grammars pointed out by several researchers are the less readable semantic rules since essential computations are mixed with a lot of copy and propagation rules. To understand the work that has been accomplished by semantic rules can take a lot of time, as shown in the next example.

```
DECLS ::= DECL \; DECLS compute {
  DECL.isGlobal = DECLS[0].isGlobal;
  DECL.inEnv = DECLS[0].inEnv;
  DECLS[1].isGlobal = DECLS[0].isGlobal;
  DECLS[0].outEnv = DECLS[1].outEnv;
  DECLS[1].inEnv = DECL.outEnv;
}
```

Specifications with templates are on higher abstraction level and hence more readable. In the semantic rules above, attribute computations are composed of bucket brigade and value distribution patterns.

```
DECLS ::= DECL \; DECLS compute {
  bucketBrigadeLeft<DECLS[0].inEnv,
    DECLS[0].outEnv,
    [DECL.inEnv, DECLS[1].inEnv],
    [DECL.outEnv, DECLS[1].outEnv]>
  valueDistribution<DECLS[0].isGlobal,
    [DECL.isGlobal, DECLS[1].isGlobal]>
}
```

The benefits of templates are:

- specifications are more readable and maintainable since templates are on higher abstraction level than assignment statements,
- specifications are reusable since the templates are independent of the structure of grammar productions, and
- language designers can create their own templates.

Let us look at the example of a simple language with assignment statements which may seem trivial, but a more concrete language would require several pages (for example COOL specifications are written on 25 pages). In the first attempt expressions have no side effects. The meaning of the program:

```
a := 5
b := a + 1 + a + a
```

is the following values: a=5, b=16. Let us develop the language without side effects in an incremental way. In each language increment only one semantic aspect are covered. In the first specification, only the rules for attribute `val` are given which reflect semantic aspect for value of an expression.

```
language Expr

  lexicon {
    Number     [0-9]+
    Operator   \+
    ignore     [\0x09\0x0A\0x0D\ ]
  }
  attributes int *.val;
  rule Expression1 {
  EXPR ::= EXPR + TERM compute {
    EXPR[0].val = EXPR[1].val + TERM.val;
    };
    }
    rule Expression2 {
      EXPR ::= TERM compute {
          EXPR.val = TERM.val;
        };
    }
    rule Term1 {
      TERM ::= #Number compute {
          TERM.val = Integer.valueOf(
            #Number.value()).intValue();
        };
    }

  } //language Expr
```

The language ExprEnv is an extension of the language Expr where regular definitions Number, **ignore**, and attribute val are inherited and reused. The regular definition operator, and rules Expression1, Expression2, and Term1 are extended. Regular definition Identifier, and rules Start, Statements, Statement, and Term2 are added. In this language semantic aspect of symbol table management is covered.

```
language ExprEnv extends Expr
  lexicon {
    Identifier        [a-z]+
    extends Operator  :=
  }
  attributes Hashtable *.inEnv, *.outEnv;
  rule Start {
    START ::= STMTS compute {
        STMTS.inEnv = new Hashtable();
        START.outEnv = STMTS.outEnv;
      };
  }
  rule Statements {
    STMTS ::= STMT STMTS compute {
        bucketBrigadeLeft<STMTS[0].inEnv,
          STMTS[0].outEnv,
          [STMT.inEnv, STMTS[1].inEnv],
          [STMT.outEnv, STMTS[1].outEnv]>
      }
    | compute {//epsilon
        bucketBrigadeLeft<STMTS.inEnv,
          STMTS.outEnv, [], []>
      };
  }
  rule Statement {
    STMT ::= #Identifier := EXPR compute {
        EXPR.inEnv = STMT.inEnv;
        STMT.outEnv = put(STMT.inEnv,
          #Identifier.value(), EXPR.val);
      };
  }
  rule extends Expression1 {
    EXPR ::= EXPR + TERM compute {
      //production can be omitted as in
      //Expression2
        valueDistribution<EXPR[0].inEnv,
          [TERM.inEnv, EXPR[1].inEnv]>
      };
  }
  rule extends Expression2 {
    compute {
        valueDistribution<EXPR.inEnv,
          [TERM.inEnv]>
      }
  }
  rule Term2 {
    TERM ::= #Identifier compute {
        TERM.val = ((Integer)
          TERM.inEnv.get(
          #Identifier.value())).intValue();
      };
  }
  method Environment{
    import java.util.*;
    public Hashtable put(Hashtable env,
        String name, int val) {
        env = (Hashtable)env.clone();
        env.put(name, new Integer(val));
        return env;
      }
  }
}//language ExprEnv
```

If later the designer needs expressions with side effects he/she must change only those parts which differ from ancestor specifications. In our example we have to use the bucket brigade left pattern instead of the value distribution pattern in rules: Expression1, Expression2, Term1, and Term2. Also, a new rule Term3, which produces a side effect with the following expression construct [id := EXPR] is introduced. The value of id is changed and propagated in further expressions. For example the next program:

```
a := 5
b := a + 1 + [a := 8] + a
```

produces the following values: a = 8 and b = 22. The language ExprSideEffect is an extension of the language ExprEnv where regular definitions Number, Operator, Identifier and **ignore**, attributes inEnv, outEnv and val, rules Start, Statements and method Environment are inherited and reused. The rules Statement, Expression1, Expression2, Term1 and Term2 are extended, and the regular definition Separator and the rule Term3 are added.

```
language ExprSideEffect extends ExprEnv
  lexicon {
```

```
    Separator \[ | \]
  }
  rule extends Start {
    compute { }
  } //for starting production
  rule extends Statement {
    compute {
        bucketBrigadeLeft<STMT.inEnv,
          STMT.outEnv,
          [EXPR.inEnv], [put(EXPR.outEnv,
          #Identifier.value(), EXPR.val)])>
      }
  }
  rule extends Expression1 {
    compute {
        bucketBrigadeLeft<EXPR[0].inEnv,
          EXPR[0].outEnv,
          [EXPR[1].inEnv, TERM.inEnv],
          [EXPR[1].outEnv, TERM.outEnv]>
      }
  }
  rule extends Expression2 {
    compute {
      bucketBrigadeLeft<EXPR.inEnv,
        EXPR.outEnv, [TERM.inEnv],
        [TERM.outEnv]>
    }
  }
  rule extends Term1 {
    compute {
        bucketBrigadeLeft<TERM.inEnv,
          TERM.outEnv, [], []>
      }
  }
  rule extends Term2 {
    compute {
        bucketBrigadeLeft<TERM.inEnv,
          TERM.outEnv, [], []>
      }
    }
  rule Term3 {
    TERM ::= [ #Identifier \:= EXPR ]
        compute
        {
        bucketBrigadeLeft<TERM.inEnv,
          TERM.outEnv, [EXPR.inEnv],
          [put(EXPR.outEnv,
          #Identifier.value(), EXPR.val)]>
        TERM.val = EXPR.val;
      }
  }
} // language ExprSideEffect
```

Let us look what semantic rules are generated from the template in rule Term3:

```
EXPR.inEnv = TERM.inEnv;
TERM.outEnv = put(EXPR.outEnv,
  #Identifier.value(), EXPR.val);
```

Language ExprSideEffect inherits properties from single parent. An example where language inherit properties from several parents can be found in [4, 21]. In [21] incremental development of PLM language is presented.

4 Formal Definition of Multiple Attribute Grammars Inheritance

Formally, inheritance can be characterized as $R = P \oplus \triangle R$ [8], where R denotes a newly defined object or class, P denotes the properties inherited from an existing object or class, $\triangle R$ denotes the incrementally added new properties that differentiate R from P, and $\oplus$ denotes an operation that combines $\triangle R$ with the properties of P. As a result of this combination, R will contain all the properties of P, except that the incremental modification part $\triangle R$ may introduce properties that overlap with those of P so as to redefine or cancel certain properties of P. Therefore, R may not always be fully compatible with P. The form of inheritance where properties are inherited from a single parent is known as single inheritance, as opposite to multiple inheritance where inheritance from several parents is allowed at the same time. Multiple inheritance can be formally characterized as $R = P_1 \oplus P_2 \oplus ... \oplus P_n \oplus \triangle R$. Before inheritance on regular definitions, context-free grammars and on attribute grammars are defined, let us look at the semantic domains used in formal definitions.

$ProdSem$ is a finite set of pairs (p, R_p), where p is a production and R_p is finite set of semantic rules associated with the production p.

$$\begin{aligned} ProdSem = \{&(p, R_p) | p \in P, \\ &p : X_0 \rightarrow X_1 X_2 ... X_n, \\ &R_p = \{X_i.a = f(X_{0.b}, \ldots, X_{j.c}) | \\ &X_i.a \in DefAttr(p)\}\} \end{aligned}$$

Properties in attribute grammars consist of lexical regular definitions, attribute definitions, rules which are generalized syntax rules that encapsulate semantic rules, and methods on semantic domains.

$$\begin{aligned} Property = &RegdefName + AttributeName + \\ &RuleName + MethodName \end{aligned}$$

For each language l, an $Ancestors(l)$ is a set of ancestors of the language l.

$$\begin{aligned} &Ancestors : Language \rightarrow \{Language\} \\ &Ancestors(l) = \{l_1, l_2, \ldots, l_n\} \end{aligned}$$

For each language l, a $LexSpec(l)$ is a set of mappings from regular definitions to regular expressions of the language l. A regular definition is a named regular expression.

$$\begin{aligned} &LexSpec : Language \rightarrow RegdefName \\ &\quad \rightarrow RegExp \\ &LexSpec(l) = \{d_1 \mapsto rexp_1, \ldots, d_n \mapsto rexp_n\} \end{aligned}$$

For each language l, an $Attributes(l)$ is a set mappings from attributes to their types of the language l.

$$\begin{aligned} &Attributes : Language \rightarrow AttributeName \\ &\quad \rightarrow Type \\ &Attributes(l) = \{a_1 \mapsto type_1, \ldots, a_n \mapsto type_n\} \end{aligned}$$

For each rule r in the language l, $Rules(l)(r)$ is a finite set of pairs (p, R_p), where p is a production and R_p is finite set of semantic rules associated with the production p.

$$
\begin{aligned}
&Rules : Language \rightarrow RuleName \rightarrow ProdSem\\
&Rules(l)(r) = \{(p, R_p) | p \in P,\\
&\ p : X_0 \rightarrow X_1 X_2 \ldots X_n,\\
&\ R_p = \{X_i.a = f(X_{0.b}, \ldots, X_{j.c} |\\
&\ X_i.a \in DefAttr(p))\}\}
\end{aligned}
$$

A set of properties of the language l_2, which are not accessible (and hence overridden) in the language l_1, is denoted with $OverriddenId(l_1, l_2)$.

$$
\begin{aligned}
&OverriddenId : (Language \times Language)\\
&\quad \rightarrow \{Property\}\\
&OverriddenId(l_1, l_2) = \{pr_1, pr_2, \ldots, pr_n\}
\end{aligned}
$$

Rules inherited from ancestors must be merged with the rules in the specified language so that the underlying attribute grammar remains well defined. If production p exists in current and in inherited rules, then semantic rules must be merged $R_p = merge(R_{pC}, R_{pI})$. Otherwise rules are simply copied from inherited or current rules.

$$
\begin{aligned}
&Merge : ProdSem \times ProdSem \rightarrow ProdSem\\
&Merge(CurrentProd, InhProd) =\\
&\ \{(p, R_p) | ((p, R_{pI}) \in InhProd \wedge\\
&\ (p, R_{pC}) \in CurrentProd \wedge\\
&\ R_p = merge(R_{pC}, R_{pI})) \vee\\
&\ ((p, R_p) \in InhProd \wedge\\
&\ (p, R_{pC}) \notin CurrentProd) \vee\\
&\ ((p, R_p) \in CurrentProd \wedge\\
&\ (p, R_{pI}) \notin InhProd))\}
\end{aligned}
$$

$merge(R_{pC}, R_{pI})$ is a set of semantic rules associated to production p where the semantic rule for the same attribute redefines the inherited ones.

$$
\begin{aligned}
&merge(R_{pC}, R_{pI}) =\\
&\ \{X_i.a = f(X_{0.b}, \ldots, X_{j.c})\\
&\ \ | X_i.a \in DefAttr(p_C)\\
&\ \ \vee (X_i.a \in DefAttr(p_I) \wedge\\
&\ \ X_i.a \notin DefAttr(p_C))\}
\end{aligned}
$$

For the function $f : A \rightarrow B$, we let $f[a/b]$ be the function that acts just like f except that it maps specific value $a \in A$ to $b \in B$. That is:

$$
\begin{aligned}
&(f[a/b])(a) = b\\
&(f[a/b])(a_0) = f(a_0); \forall a_0 \in A \wedge a_0 \neq a
\end{aligned}
$$

4.1 Regular definition inheritance

The input string can be recognized with different regular expressions even in monolitic lexical specifications. In such cases the first match rule is commonly used and the order of regular expressions becomes important. The concept of inheritance of regular definitions causes further problems as presented in the following example [4]:

```
AddSubCalc.digit        [0-9]
Dec.int                 [0-9]+
```

For example, the input string '7' is recognized as `AddSubCalc.digit`. If reference to `Dec.int` was made in the syntax specifications, the error would be reported, despite the correctness of specifications. If the order of regular definitions were different, the same problem would appear with reference to `AddSubCalc.digit`. Our solution to this problem is to find all matching regular definitions for the input string. For example, the result of lexical analyses for the input string '7' would be the set `{AddSubCalc.digit, Dec.int}`. In that case reference to both regular definitions can be made and therefore the sequence of regular definitions becomes irrelevant. For these reasons the inheritance of regular definitions is defined in the following way:
Let $E_1, E_2, ..., E_m$ be sets of mappings from regular definitions to regular expressions of languages $l_1, l_2, \ldots l_m$ formally defined as

$$
\begin{aligned}
E_1 &= \{d_{11} \mapsto e_{11}, d_{12} \mapsto e_{12}, \ldots, d_{1k} \mapsto e_{1k}\}\\
E_2 &= \{d_{21} \mapsto e_{21}, d_{22} \mapsto e_{22}, \ldots, d_{2l} \mapsto e_{2l}\}\\
&\vdots\\
E_m &= \{d_{m1} \mapsto e_{m1}, \ldots, d_{mn} \mapsto e_{mn}\}
\end{aligned}
$$

where d_{ij} is a regular definition and e_{ij} is a regular expression, then $E = E_2 \oplus \ldots \oplus E_m \oplus \triangle E_1$, where E_1, which inherits from $E_2, \ldots, E_m$, is defined as:

$$E = E_1 \cup \ldots \cup E_m.$$

4.2 Context-free grammar inheritance

Let $G_1, G_2, \ldots, G_m$ be context-free grammars, formally defined as

$$
\begin{aligned}
&G_1 = (T_1, N_1, S_1, P_1),\\
&G_2 = (T_2, N_2, S_2, P_2),\\
&\quad \vdots\\
&G_m = (T_m, N_m, S_m, P_m), \text{ then}
\end{aligned}
$$

$$
\begin{aligned}
&G = G_2 \oplus \ldots \oplus G_m \oplus \triangle G_1,\\
&\quad \text{where } G_1, \text{ which inherits from}\\
&\quad G_2, \ldots, G_m, \text{ is defined as}
\end{aligned}
$$

$$
\begin{aligned}
&G = (T, N, S_1, P), \text{ where}\\
&T = T_1 \oslash \ldots \oslash T_m,\\
&N = N_1 \oslash \ldots \oslash N_m,\\
&P = P_1 \odot \ldots \odot P_m.
\end{aligned}
$$

Note that the start nonterminal symbol of context free grammar G is the start nonterminal of context-free grammar G_1. Since the incrementally added new productions P_1 may override some productions where terminal and nonterminal symbols are defined, the final set of terminal symbols T and the set of nonterminal symbols N are not simply a union of inherited terminal and nonterminal symbols. The operation $\oslash$ is defined as:

$V_1 \oslash V_2 \oslash \ldots \oslash V_m =$
$V_1 \cup (V_2 \setminus \{x | x \in OverriddenSym(l_1, l_2)\})$
$\cup \ldots \cup$
$(V_m \setminus \{x | x \in OverriddenSym(l_1, l_m)\})$.

Where, $OverriddenSym(l_1, l_2)$ is a set of overridden symbols of the language l_2 which are not accessible from language l_1. Also, the set of productions P is not simply a union of inherited productions since some productions may be overridden or cause horizontal overlap [8]. The operation is defined as:

$P = P_1 \odot \ldots \odot P_m = P_1 \cup$
$(P_2 \setminus \{p | p \in fst(Rules(l_2)(r)) \wedge$
$r \in OverriddenId(l_1, l_2)\}) \cup \ldots \cup$
$(P_m \setminus \{p | p \in fst(Rules(l_m)(r)) \wedge$
$r \in OverriddenId(l_1, l_m)\}) \wedge$
$dom(Rules(l_i)) \cap dom(Rules(l_j)) = \emptyset,$
$i = 2..m, j = 2..m \wedge i \neq j.$

4.3 Multiple attribute grammar inheritance

Let $AG_1, AG_2, \ldots, AG_m$ be attribute grammars formally defined as:

$AG_1 = (G_1, A_1, R_1),$
$AG_2 = (G_2, A_2, R_2),$
$\vdots$
$AG_m = (G_m, A_m, R_m)$, then

$AG = AG_2 \oplus \ldots \oplus AG_m \oplus \triangle AG_1$,
where AG_1, which inherits from
$AG_2, \ldots, AG_m$, is defined as

$AG = (G, A, R)$, where
$G = G_2 \oplus \ldots \oplus G_m \oplus \triangle G_1,$
$A = A_1 \ominus \ldots \ominus A_m,$
$R = R_1 \otimes \ldots \otimes R_m$.

Since each attribute has a type, a set of attributes A_i is defined as:

$A_i = \{a_{i1} \mapsto type_{i1}, \ldots, a_{in} \mapsto type_{in}\}.$

Then, $A = A_1 \ominus \ldots \ominus A_m$ can not be defined simply as a union, since the same attribute can be of different type in a different set A_i. This situation denotes horizontal or vertical overlapping. Since unordered inheritance is used, horizontal overlapping is forbidden and vertical overlapping is resolved by asymmetric descendant-driven lookup [8]. Hence, $A = A_1 \ominus \ldots \ominus A_m$ is defined as:

$A = A_1 \cup (A_2 \setminus \{a_{1p} \mapsto type_{1p} | a_{1p} \in fst(A_1)\})$
$\cup \ldots \cup (A_m \setminus \{a_{1p} \mapsto type_{1p} | a_{1p} \in$
$fst(A_1)\}) \wedge (\neg \exists a_{ji}, j = 2..m, i = 1..n,$
$k \neq l : (a_{ji} \mapsto type_{jk}) \wedge$
$(a_{ji} \mapsto type_{jl}) \wedge (type_{jk} \neq type_{jl})).$

The set of semantic rules R is not a simple union of inherited semantic rules, since some semantic rules may be overridden or may cause horizontal overlap. In any case, current semantic rules have to be merged with inherited semantic rules.

$R = R_1 \otimes \ldots \otimes R_m =$
$R_1 \cup snd(Merge((P_1, R_1), (P_2, R_2 \setminus$
$\{R_p | R_p \in snd(Rules(l_2)(r)) \wedge$
$r \in OverriddenId(l_1, l_2)\})))$
$\cup \ldots \cup snd(Merge((P_1, R_1),$
$(P_m, R_m \setminus \{R_p | R_p \in snd(Rules(l_m)(r)) \wedge$
$r \in OverriddenId(l_1, l_m,)\})))$
$\wedge dom(Rules(l_i)) \cap dom(Rules(l_j)) = \emptyset,$
$i = 2..m, j = 2..m, i \neq j.$

Let us look in more detail what is the result of operation $merge(R_{Expression1}, ExprEnv.R_{Expression1})$. Semantic rules asociated to production `EXPR → EXPR + TERM` in language `ExprEnv` are semantic rules obtained from operation $merge(R_{Expression1}, Expr.R_{Expression1})$.

$merge(R_{Expression1}, Expr.R_{Expression1}) = \{$
EXPR[0].val=EXPR[1].val + TERM[0].val,
TERM[0].inEnv = EXPR[0].inEnv,
EXPR[1].inEnv = EXPR[0].inEnv$\}$

Defined attributes in production `EXPR → EXPR + TERM` of the language `ExprEnv` are: `EXPR[0].val`, `TERM[0].inEnv`, `EXPR[1].inEnv`, and in production `EXPR → EXPR + TERM` of the language `ExprSideEffect` the defined attributes are `EXPR[1].inEnv`, `Term[0].inEnv`, `EXPR[0].outEnv`. Attributes `TERM[0].inEnv` and `EXPR[1].inEnv` are defined in both productions, however redefined semantic rules are used. Therefore the result of operation $merge(R_{Expression1}, ExprEnv.R_{Expression1})$ is

```
{EXPR[0].val=EXPR[1].val + TERM[0].val,
  EXPR[1].inEnv = EXPR[0].inEnv,
  TERM[0].inEnv = EXPR[1].outEnv,
  EXPR[0].outEnv = TERM[0].outEnv}
```

5 Tool LISA ver 2.0

Multiple attribute grammar inheritance is successfully implemented in the compiler/interpreter generator tool LISA ver. 2.0. The tool LISA is compiler generator with the following features:

- LISA is platform independent since it is written in Java
- it offers the possibility to work in a textual or visual environment
- it offers an integrated development environment (fig. 1) where users can specify - generate - compile-on-the-fly - execute programs in a newly specified language
- lexical, syntax and semantic analysers can be of different types and can operate standalone; the current version of LISA supports LL, SLR, LALR, and LR parsers, tree-walk, parallel, L-attribute and Katayama evaluators

- visual presentation of different structures, such as finite state automata, BNF, syntax tree, semantic tree, dependency graph
- animation of lexical, syntax and semantic analysers
- the specification language supports multiple attribute grammar inheritance and templates which enable to design a language incrementally or reuse some fragments from other programming language specifications.

6 Related Work

There has been a lot of research on augmenting ordinary attribute grammars with extensions to overcome deficiencies of attribute grammars such as lack of modularity, extensibility and reusability [9, 10, 11, 12, 13, 14, 15].

Modular attribute grammars MAG [16] are proposed as a solution to attribute pragmatic problems. The whole language specification consists of several MAGs. A single MAG is a set of patterns and associated templates. For each match between a production and pattern a set of attribute computations is generated. Both, the matching and the generation process are further constrained to generate only useful and meaningful attribute computations. As in our template approach, MAG too specifies the semantic rules for sets of productions rather than for a particular production. We are convinced that our template approach offers a better abstraction of attribute computation since our template is a generic module parameterized by attribute instances, which is not the case with MAG modules. Also, in our approach the attribute computation generation is explicitly stated by the designer, and in MAG by the pattern matching process which is very difficult to follow. On the other hand, MAG has no counterpart to our multiple inheritance approach.

We borrowed the idea of grammar inheritance from [17] where the only property is a production rule, and extended it to multiple attribute grammar inheritance. The difference between the approaches is also in the granularity of modification. In the approach of [17] modification is possible only for the whole production rule, since the name of the property is left hand nonterminal.

In object-oriented attribute grammars [3, 18] the concepts of class and class hierarchies have been introduced where nonterminals act as classes and class hierarchies have been derived from the context free grammar. Inheritance could be applied to attributes, attribute computations and syntactic patterns within one attribute grammar. It is also well known that inherited attributes and class hierarchies produce some conflicts on well-definedness of attribute grammars and hence multiple inheritance is not allowed, and also inherited attributes can not be used in dynamic classes. In our approach a different view is chosen where the whole attribute grammar is a class without the above mentioned conflicts.

In the report [19], extensible attribute grammars are used to generate integrated programming systems in an incremental way. In order to perform incremental generation as quickly and as easily as possible, the restricted form of extension is used. For example, nonterminal symbols can not disappear on the right hand of productions upon extensions. At most they can be replaced by extended nonterminals which must contain all attributes of its respective base nonterminal. Therefore, extensible attribute grammars support some form of strict inheritance while our approach supports nonstrict inheritance.

In our opinion the only widely accepted approach with reusability of attribute grammars is the approach presented in [20] and incorporated in the Eli compiler generator, where with remote attribute access and inheritance, an attribution module is defined which can be reused in a variety of applications. But with this approach the attribution module can be only constructed for those attribute computations where the attribute depends only on remote attributes. In this case computation is associated to a symbol rather than to production. With the inheritance described in [20] an attribute computation can be further independent from symbols used in particular language definitions.

Recently some new attempts to better modularity and extensibility of attribute grammars have been proposed also in functional paradigm [23] where Bird example [22] and its modification were presented. With our approach the same example can be easily implemented using attribute grammar inheritance. First we write the attribute grammar for Bird example:

```
language Bird {
 lexicon {
  tip      \-?[0-9]+
  nodeop   \( | \) | ,
  ignore   [\0x0D\0x0A\ \0x09]+
 }
 attributes int *.min, *.inMin;
  String *.val;
 rule Start {
  START ::= TREE compute {
    START.val = TREE.val;
    TREE.inMin = TREE.min;
  };
 }
 rule Tree {
  TREE ::= #tip compute {
    TREE.min =
    Integer.valueOf(#tip.value()).intValue();
    TREE.val= ""+TREE.inMin;
  } | ( TREE , TREE ) compute {
    TREE.min = TREE[1].min<TREE[2].min?
      TREE[1].min:TREE[2].min;
    TREE[1].inMin = TREE.inMin;
    TREE[2].inMin = TREE.inMin;
    TREE.val = "( "+TREE[1].val+", "
      +TREE[2].val+")";
  };
 }
}
```

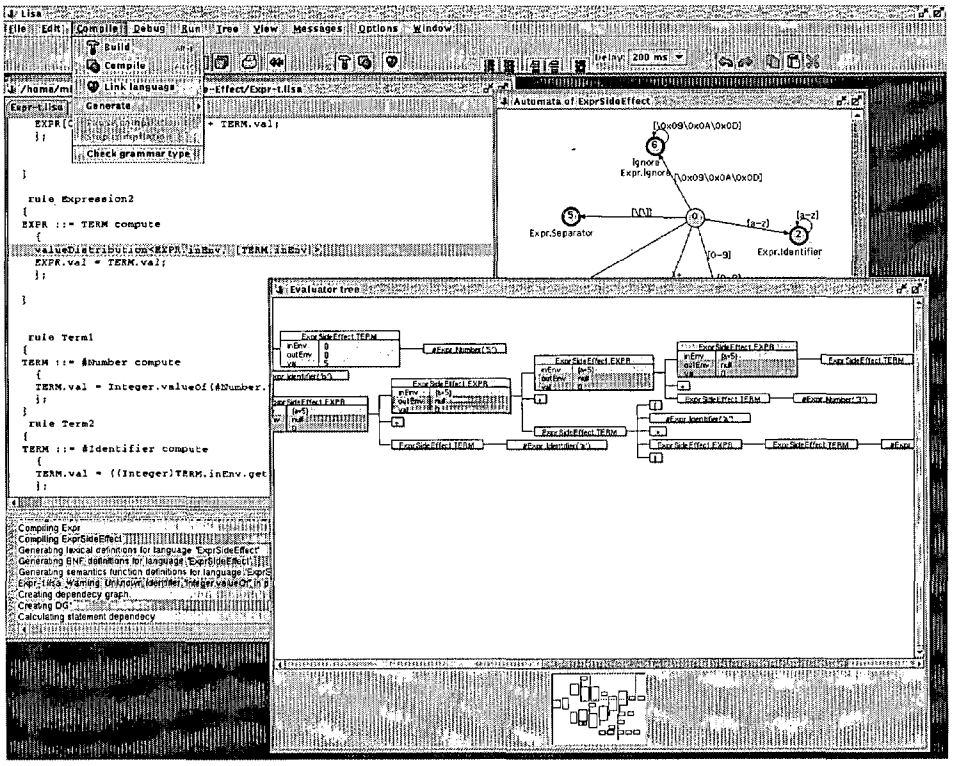


Figure 1: LISA Integrated Development Environment

Evaluation of the attribute grammar requires two tree passes. In first pass the global minimum value of the tree is computed. In second pass a new tree with the same topology is constructed by replacing all leafs with global minimum value from the first pass.

For the input tree:

```
(2, ((3,(-10, 2)), (-10, 5)))
```

the generated result is:

```
(-10, ((-10,(-10, -10)), (-10, -
10))).
```

As presented in [23] we modify the attribute grammar for different problem with grammar inheritance. Modification is to replace the each leaf with the number of global minimum occurrences in the left of the leaf.

For the same input tree the output is:

```
(0, ((0,(1, 1)), (2, 2))).
```

Extension of attribute grammar Bird:

```
language ExtBird extends Bird {
 rule extends Start {
 compute {
  TREE[0].inMinCount = 0;
 }
}
attributes int *.minCount, *.inMinCount;
rule extends Tree {
 TREE ::= #tip compute {
  TREE.minCount = TREE.inMinCount +
   (Integer.valueOf(#tip.value()).intValue()
   == TREE.inMin ?1:0);
  TREE.val = String.valueOf(TREE.minCount);
 } | ( TREE , TREE ) compute {
  TREE[1].inMinCount = TREE[0].inMinCount;
  TREE[2].inMinCount = TREE[1].minCount;
  TREE[0].minCount = TREE[2].minCount;
  };
 }
}
```

The only modifications of the original attribute grammar are semantics functions for computation of minimum value occurrence and redefinition of the leaf computation. This is very easily done using attribute grammar inheritance as presented. Attribute grammar inheritance is very natural approach since the notion of inheritance is close to developers from object oriented programming languages.

7 Conclusion

When introducing a new concept, the designer has difficulties in integrating it into the language in an easy way. To enable incremental language design we introduce a new object oriented attribute grammar specification language based on the paradigm Attribute Grammar = Class. In multiple attribute grammar inheritance the properties which can be inherited or overridden are regular definitions, attributes, rules which encapsulate productions and semantic rules, and methods. Therefore, with multiple attribute grammar inheritance we can extend the lexical, syntax and semantic part of language definition. In the paper

an example and the formal definition of multiple attribute grammar inheritance is given. The main advantages of the proposed approach are:
– simplicity and clearness of the approach,
– the object concept is simply transposed on the basic objects of attribute grammars at the specification level, and
– incremental language development is enabled.

We have incrementally designed various small programming languages, such as COOL and PLM with multiple attribute grammar inheritance. Our experience with these non-trivial examples shows that multiple inheritance in attribute grammars is useful in managing the complexity, reusability and extensibility of attribute grammars. The benefit of this approach is also that for each language increment a compiler can be generated and the language tested.

References

[1] Mernik M., Korbar N., Žumer V. LISA: A Tool for Automatic Language Implementation. *ACM Sigplan Notices*, Vol. 30, No. 4, pp. 71 – 79, 1995.

[2] Žumer V., Korbar N., Mernik M. Automatic Implementation of Programming Languages using Object Oriented Approach. *Journal of Systems Architecture*, Vol. 43, No. 1–5, pp. 203 – 210, 1997

[3] Paakki J. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Comp. Surveys*, Vol. 27, No. 2, pp. 196–255, 1995.

[4] Mernik M., Lenič M., Avdičaušević E., Žumer V. The Template and Multiple Inheritance Approach into Attribute Grammars. *International Conference on Computer Languages*, Chicago, pp. 102 – 110, 1998.

[5] Knuth D. E. Semantics of contex-free languages. *Math. Syst. Theory*, Vol. 2, No. 2, pp. 127 – 145, 1968.

[6] Deransart P., Jourdan M. (Eds.) Attribute Grammars and their Applications. *1st Workshop WAGA, Lecture Notes in Comp. Science 461*, Springer-Verlag, 1990.

[7] Alblas H., Melichar B. (Eds.) Attribute Grammars, Applications, and Systems. *Lecture Notes in Computer Science 545*, Springer-Verlag, 1991.

[8] Taivalsaari A. On the Notion of Inheritance. *ACM Comp. Surveys*, Vol. 28, No. 3, pp. 438 – 479, 1996.

[9] Jourdan M., Parigot D., Julie C., Durin O., LeBellec C. Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System. *ACM Sigplan Notices*, Vol. 25., No. 6., pp. 209 – 222, 1990.

[10] Swierstra S.D., Vogt H.H. Higher order attribute grammars: a merge between functional and object oriented programming. *Report RUU-CS-90-12*, Utrecht University, 1990.

[11] Farrow R., Marlowe T.J., Yellin D.M. Composable Attribute Grammars: Support for modularity in translator design and implementation. *19th Annual ACM Sigplan - Sigact Symposium on Principles of Programming Languages*, pp. 223 – 234, 1992.

[12] Boyland J., Graham S. Composing tree attributions. *21th Annual ACM Sigplan - Sigact Symposium on Principles of Programming Languages* , pp. 375 – 388. 1994.

[13] Efremidis S.G., Mughal K.A., Soraas L., Reppy J.H. AML: Attribute Grammars in ML. *Nordic Journal of Computing*, March 1997.

[14] Correnson L., Duris E., Parigot D., Roussel G. Generic programming by program composition. *In Workshop on Generic programming*, Marstrand, Sweden, June 1998.

[15] Lämmel R. Functional meta–programs towards reusability in the declarative paradigm. *Ph.D. thesis*, University of Rostock, Department of Computer Science, 1999.

[16] Dueck G.D.P., Cormack G.V. Modular Attribute Grammars. *Computer Journal*, Vol. 33, No. 2, pp. 164 – 172, 1990.

[17] Aksit M., Mostert R., Haverkort B. Grammar Inheritance. *Technical Report*, Department of Computer Science, University of Twente, 1991.

[18] Hedin G. An overview of door attribute grammars. *5th International Conference on Compiler Construction (CC'94)*, Lecture Notes in Computer Science 786, Springer-Verlag, pp. 31 – 51, 1994.

[19] Marti R., Murer T. Extensible Attribute Grammars *Institut fur Technische Informatik und Kommunikationsnetze*, ETH Zurich, Report No. 92–6, 1992.

[20] Kastens U., Waite W.M. Modularity and reusability in attribute grammars, *Acta Informatica*, Vol. 31, 1994, pp. 601 – 627, 1994.

[21] Mernik M., Lenič M., Avdičaušević E., Žumer V. A reusable object-oriented approach to formal specifications of programming languages. *L'object*, Vol. 4, No. 3, pp. 273 – 306, 1998.

[22] Bird R. S. Using circular programs to eliminate multiple traversals of data *Acta Informatica*, Vol. 21, pp. 239 – 250, 1984.

[23] Oege de Moor, Backhouse K., Swiestra D. First-class Attribute Grammars *In D. Parigot and M. Mernik, editors, Third Workshop on Attribute Grammars and Their Applications, WAGA2000*, pp. 1 – 20, 2000.

First-class Attribute Grammars

Oege de Moor and Kevin Backhouse
Programming Research Group
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{oege,kevinb}@comlab.ox.ac.uk
AND
S. Doaitse Swierstra
Department of Computer Science
PO Box 80.089, 3508 TB Utrecht, The Netherlands
doaitse@cs.uu.nl

Keywords: Attribute Grammar, Functional Programming, Component, Aspect

Edited by: Marjan Mernik and Didier Parigot
Received: July 28, 2000 **Revised:** August 20, 2000 **Accepted:** September 16, 2000

In this paper, we show how attribute grammars can be divided into components. We introduce three types of component, called families, rules and aspects. We use the programming language Haskell [4] to give these components (and their composition) a concise executable semantics. We also show how our semantics makes it easy to define a number of generic attribution patterns such as chained attributes [16].

1 Introduction

This paper is a contribution to the ongoing quest for modular descriptions of language processors, with the specific aim of rapidly prototyping domain-specific languages [21]. Some might argue that this problem was solved in the eighties, with the development of a proliferation of language processors based on attribute grammars [11, 15, 22]. Others might argue that functional programming languages such as ML are adequate for the purpose, without any further extensions. We believe that functional programming languages do not offer enough specialised support for implementing compilers. However, attribute grammars are not in widespread use, despite their many advantages. This may be due to restrictions imposed by attribute definition languages, which are often less flexible than general purpose functional programming languages. Such general languages tend to yield descriptions that are compact, but they lack the dedicated structuring mechanisms of attribute grammars.

In this paper we initiate a systematic study of such structuring mechanisms, by giving them a compositional semantics. The semantics is expressed in the vocabulary of functional programming. Our semantics thus opens the way towards combining the powerful structuring mechanisms for attribute grammars with the flexibility of a general purpose programming language. In particular, it is easy to define new structuring operators in our semantics. Furthermore, because the semantics is a functional program, one immediately obtains a prototype for experimenting with newly defined features. Naturally the results of this paper do not stand on their own, and many of the ideas have been gleaned from the attribute grammar literature, in particular [5, 6, 16, 19, 20, 23, 26]. Especially the thesis by Stephen Adams [1] has been an inspiration for this work.

Attribute grammars and functional programming There exists a well-known encoding of attribute grammars into programming languages that have lazy evaluation [14, 18]. This encoding has been dismissed by others on the following grounds:

- Lazy evaluation is inherently inefficient, and therefore an attribute evaluator based on it must be inefficient.
- The resulting programs are highly convoluted and much less modular than standard attribute grammars.

The first objection has been refuted by the work of Augusteijn, who has built an attribute grammar evaluator based on lazy evaluation: he reports that its performance is on a par with other systems that do a sophisticated analysis of dependencies, and produce a schedule for the attribute computations based on that analysis. Augusteijn's system, named *Elegant*, has been widely used within Philips for implementing domain-specific languages [2]. Because our primary objective is a compositional semantics, the efficiency issue is not really important in the present context.

The second objection remains valid, however, and indeed the Elegant system 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 reuse the same set of attribution rules, and make them contribute to different productions. Elegant is particular in this respect: many other attribute grammar systems do allow such groupings, but only at a syntactic and not at a

semantic level. If we wanted to provide the same functionality in a general purpose programming language, so that rules, productions and grammars are all first-class citizens of the language, we would have to give a type to each reusable component.

The purpose of types is to guarantee the absence of certain run-time errors. In choosing an appropriate type system for composing attribute grammars from smaller components, we need to decide what run-time errors we wish to avoid. There are a number of common errors that are typically caught by attribute grammar systems: a mismatch between productions as used in the attribute definitions and in the context-free grammar, the use of an attribute that has not been defined, a cyclic dependency between attribute definitions, and the use of an attribute in a context that does not match its type. In this paper, we only aim to avoid the last kind of error.

The idea of embedding domain-specific languages directly into a more general *host language* is a buoyant area of research. Recent examples include languages for pretty-printing [13], reactive animation [9], and musical composition [12]. This paper adds the example of attribute grammars to that list. All these works, including our own, can be seen as providing an executable semantics for a domain-specific language. While studying semantics, one is not concerned with matters of concrete syntax, and indeed we shall defer the choice of appropriate notations to later work.

As argued by Swierstra *et al.* in [25], some of the above examples of embedded domain-specific languages could be more nicely structured in an attribute grammar style. In that paper, an attribute grammar preprocessor is used for achieving the desired structure. The present paper provides a semantics of that preprocessor.

Overview The structure of the paper is as follows. First we introduce a small attribute grammar example that we shall use throughout to illustrate the ideas. We show how we might simplify the attribute grammar by using "aspects". Next, we introduce our notation, which is loosely based on the lazy functional programming language Haskell [4]. The notation is illustrated by defining the basic types of trees, productions and attributes. They provide the preliminaries for discussing *families*, *rules* and *aspects*: the building blocks of our semantics. To illustrate these building blocks in a practical setting, we revisit our introductory example. We then show how easily they can be mapped into an executable implementation. Finally, we discuss directions for future work, in particular how we can provide more sophisticated static checks on the composed attribute grammar.

It is assumed that the reader has some degree of familiarity with a modern functional programming language, as well as the basic concepts of attribute grammars. The traditional encoding of attribute grammars in a lazy functional language is described by [14]. A passing acquaintance with this encoding will be helpful, but is not necessary. A good introduction to the style of functional definition in this paper can be found in [4].

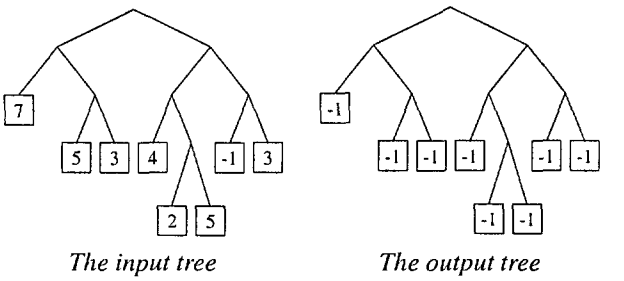


Figure 1: An example of the use of *repmin*

2 An example: *repmin*

Consider binary trees, whose internal *nodes* are unlabelled, and whose *leaves* are labelled with integer values. We aim to replace all leaf values by the minimum leaf value. An example of this is given in Figure 1. This is known as the *repmin* problem, and it was first introduced by [3]. As noted by [18], the *repmin* problem is easily expressed as an attribute grammar, which we will now present in an anonymous but hopefully self-explanatory notation.

As a first step we introduce two synthesised attributes, named *ntree* (for *new tree*), and *locmin* (for *local minimum*). Furthermore there is one inherited attribute, named *gmin* (for *global minimum*). The strategy is to recursively compute the local minimum on all nodes. The global minimum equals the local minimum of the root. This value is broadcast to all the leaves. The new tree is then built recursively.

The production named *Root* rewrites the start symbol *Start* to *Tree*. The resulting tree of parent *Start* is the resulting tree of the child *Tree*. It is here that the global minimum is defined:

$$\begin{array}{lcl} Root & : & Start \rightarrow Tree \\ Start.ntree & = & Tree.ntree \\ Tree.gmin & = & Tree.locmin \end{array}$$

At each binary node, the local minimum is obtained by taking the minimum of both subtrees; the global minimum is broadcast from the parent to both children. Here and below, we use indices to refer to successive occurrences of the nonterminal *Tree*:

$$\begin{array}{lcl} Node & : & Tree_0 \rightarrow Tree_1\ Tree_2 \\ Tree_0.ntree & = & Node\ Tree_1.ntree \\ & & \quad Tree_2.ntree \\ Tree_0.locmin & = & min\ Tree_1.locmin \\ & & \quad Tree_2.locmin \\ Tree_1.gmin & = & Tree_0.gmin \\ Tree_2.gmin & = & Tree_0.gmin \end{array}$$

Finally, the local minimum of a leaf is its value, and the new tree is a leaf with the global minimum as its value:

$$
\begin{array}{lcl}
Leaf & : & Tree \rightarrow Val \\
Tree.ntree & = & Leaf\ Tree.gmin \\
Tree.locmin & = & Val.value
\end{array}
$$

Despite the simplicity of this example, there is already quite a lot of tedious detail to take care of, most notably the copying of the *gmin* attribute from the root of the tree to the leaves. It is also a little annoying that the definition of each attribute is smeared out over several productions, making it difficult to see the flow of information at a glance. It is for that reason that practical attribute grammar systems provide better structuring mechanisms, of the kind that we shall discuss below.

Let us plod on, however, and consider how the above attribute grammar would have to be modified for a slightly different problem. Instead of replacing each leaf L by the global minimum, we aim to replace it by the number of times the global minimum occurred to the left of L in the inorder traversal of the tree. For this we introduce an additional chained attribute *count* that keeps track of that number. The new root production initialises the count to zero:

$$
\begin{array}{lcl}
Root & : & Start \rightarrow Tree \\
Tree.count & = & 0
\end{array}
$$

At a node, we chain the count from left to right. As is often the case with chained attributes, there is some subtle punning going on with the names: the first mention of $Tree_0.count$ is the *inherited* attribute *count*, whereas its second occurrence is the *synthesised* attribute of the same name:

$$
\begin{array}{lcl}
Node & : & Tree_0 \rightarrow Tree_1\ Tree_2 \\
Tree_1.count & = & Tree_0.count \\
Tree_2.count & = & Tree_1.count \\
Tree_0.count & = & Tree_2.count
\end{array}
$$

Finally, at a leaf we compare the value to the global minimum, and if they coincide, the counter is incremented. We also redefine the computation of *ntree*:

$$
\begin{array}{lcl}
Leaf & : & Tree \rightarrow Val \\
Tree.ntree & = & Leaf\ Tree.count \\
Tree.count & = & \\
\quad \textbf{if}\ Val.value = Tree.gmin\ \textbf{then} & & \\
\quad\quad Tree.count + 1 & & \\
\quad \textbf{else} & & \\
\quad\quad Tree.count & &
\end{array}
$$

To obtain a program for the modified *repmin* problem, we now have to paste these new definitions into the original grammar. This involves adding the new rules for *count* to each of the productions, and overriding the original definition of the *ntree* attribute in the *Leaf* production. Indeed, most attribute grammar systems treat structuring mechanisms in this syntactic way. Furthermore, they introduce syntactic abbreviations for common patterns such as chained attributes [16]. We aim to show how these structuring operations can be given a precise semantics.

In our semantic view, the only essential difference between the above two grammars is the presence of the *count* attribute: the rest of the semantics is shared. The overriding of the *ntree* attribute is modelled by making *ntree* a parameterised attribute. Furthermore, the semantics facilitates easy definitions of oft-occurring patterns (such as that of *chained attributes*, and *broadcasting of inherited attributes*). Making such patterns explicit removes a lot of the tedium involved in writing attribute grammars, and also makes them easier to read. It is the compositional semantics (of well known structuring mechanisms) that is the contribution of this paper. The fact that the semantics is an executable prototype is a pleasant side effect of expressing ourselves in a lazy functional programming language. Having an executable prototype makes it easy to experiment with new structuring operators, giving an opportunity to explore beyond the fixed vocabulary of typical attribute grammar systems.

With some syntactic sugar for increased readability, the new formulations of the *repmin* problem and its variation are as follows. First we introduce the inherited attribute *gmin*. It is introduced through a so-called *attribute aspect* that groups several definitions for an attribute together. This aspect stipulates that *gmin* is copied at the *Node* production, and a specialised definition is given at the root:

$$
\begin{array}{l}
gmins = \\
\quad \textbf{inherit}\ gmin \\
\quad \textbf{copy at}\ Node \\
\quad \textbf{define at}\ Root[Tree] : Tree.locmin
\end{array}
$$

Here, the notation $Root[Tree]$ specifies that we are defining an attribute on the *Tree* nonterminal of the *Root* production.

One can also define attribute aspects for synthesised attributes. The default behaviour here is to collect multiple occurrences of the attribute from the children. In the case of the local minimum, we collect with the minimum function, and its value at a leaf is simply the original label:

$$
\begin{array}{l}
locmins = \\
\quad \textbf{synthesise}\ locmin \\
\quad \textbf{collect with}\ min\ \textbf{at}\ Node \\
\quad \textbf{define at}\ Leaf[Tree] : Val.value
\end{array}
$$

To cater for later variation, the construction of the new tree is parameterised by the attribute that we substitute for leaves:

$$
\begin{array}{l}
ntrees = \\
\quad \textbf{synthesise}\ ntree(a : AttributeInt) \\
\quad \textbf{collect with}\ Node\ \textbf{at}\ Node \\
\quad \textbf{define at}\ Leaf[Tree] : Leaf\ Tree.a \\
\qquad\qquad\qquad Root[Start] : Tree.ntree
\end{array}
$$

Finally, the chained counter has special definitions in two productions. It is inherited in *Root*, and synthesised in *Leaf*:

$$
counts =
$$

```
chain count
define at
  Root[Tree] : 0
  Leaf[Tree] :
    if Val.value = Tree.gmin then
        Tree.count + 1
    else
        Tree.count
```

The solution to the original problem is now obtained by assembling the above aspects with the following Haskell expression:

$$repmin_0 = compiler\ [gmins, locmins, ntrees\ gmin]\ ntree$$

The final argument indicates that we want to return the *ntree* attribute as the result of compilation. The more complicated variation of the *repmin* problem is assembled by including the counter:

$$repmin_1 = compiler\ [gmins, locmins, ntrees\ count, counts]\ ntree$$

We should stress that each of the attribute aspects *gmins*, *locmins*, *ntrees* and *counts* are first-class values that can be passed as parameters and returned as results. To define precisely what those values are is the goal of the remainder of this paper.

3 Preliminaries: Trees, Productions and Attributes

To set the scene, and to introduce some Haskell vocabulary through familiar concepts, we start by defining trees, productions and attributes. Most of these definitions are extremely straightforward. It is only in our definition of attributes that we have to exercise some foresight. This will facilitate easy composition at a later stage. Readers who are familiar with Haskell may wish to skim the subsection on attributes, and then proceed to the next section, which is the core of the paper.

3.1 Trees

For simplicity, our attribute grammars will operate on a rather primitive kind of tree, whose type is independent of the underlying context-free grammar. As said in the introduction, that makes our semantic definitions simpler, but it does carry the risk of run-time errors when an attribute grammar is applied to a particular tree. A safer approach would be to define a separate type of tree for each grammar.

A tree is either a *Fork* labelled with a value of type α and a list of descendants that are also trees, or it is a *Val* labelled with a β:

$$\textbf{data}\ Tree\ \alpha\ \beta = Fork\ \alpha\ [Tree\ \alpha\ \beta]\ |\ Val\ \beta$$

Typically, the type α represents the names of productions, and β is the type of attributions that were computed by the scanner or parser. The most common type of tree is therefore *Tree ProdName Attrs*, where *ProdName* denotes the type of names of productions, and *Attrs* that of attributions. Both of these types will be formally defined below. Sometimes it is convenient to vary the instantiations of α and β in the definition of trees, however, and that is why we abstract from the concrete type. An example where that flexibility will come in handy is the definition of a function that decorates a tree with all relevant attribute values.

In our running example, we have a grammar with three productions named *Root*, *Node* and *Leaf*. Together these names make up the data type of production names that may occur at *Fork* nodes of a tree:

$$\textbf{data}\ ProdName = Root\ |\ Node\ |\ Leaf$$

3.2 Attributes and attributions

An *attribution* is a finite mapping from attribute names to attribute values. We shall exercise a little notational freedom when discussing finite mappings, and write $A \mapsto B$ for the set of finite maps from A to B. Accordingly, the type of attributions is defined:

$$\textbf{type}\ Attrs = AttrName \mapsto AttrValue$$

Note that in contrast to previous types (which were new types, introduced with the Haskell keyword **data**) this type is merely an abbreviation for an existing type (which is indicated by using **type** in lieu of **data**). The choice to model attributions as finite maps implies that we cannot guarantee, by exploiting the type system of Haskell, that certain names are present in an attribution: such a check could have been enforced by modelling attributions through record types. Note also that all attributes map to values of the same type, namely *AttrValue*. As we shall see below, *AttrValue* is defined as the disjoint union of all possible attribute types in a particular grammar.

We shall often write $\{(n_0, v_0), (n_1, v_1), \ldots, (n_{k-1}, v_{k-1})\}$ for the attribution that sends each name n_i to the value v_i. Strictly speaking this is not valid Haskell syntax, but it will ease the presentation of concrete examples.

It is our goal to make all concepts in our semantics composable, and that entails introducing a union, join or merge operation wherever we can. In the case of attributions, the obvious choice is the *join* of finite maps. For finite maps f and g, the join $f \oplus g$ is defined by:

$$\begin{aligned}(f \oplus g)\ x &= f\ x, \textbf{ if } x \in domain\ f \\ &= g\ x, \textbf{ otherwise}\end{aligned}$$

In this definition, we are again taking a notational liberty, namely writing application of finite maps as ordinary function application. In Haskell, a special operator has to be

introduced. Furthermore, in Haskell, application of a finite map to an element outside its domain will result in a run-time error. Note that the join operator is associative, and it has an identity element, namely the empty map.

While we have shown how to combine attributions, as yet we do not have a way of putting elements (*i.e.* (name,value) pairs) into an attribution. We define such embedding functions, one for each attribute, along with the corresponding projection. In fact, we take such an embedding/projection pair as the *definition* of an attribute. To wit, the type of attributes whose values are of type α is:

$$\textbf{type}\ At\,\alpha = (\alpha \to Attrs,\ Attrs \to \alpha)$$

The first component of such a pair is the embedding, and the second is the projection:

$$\begin{aligned} &embed && :: At\,\alpha \to \alpha \to Attrs \\ &embed\,(e,p) && = e \\ \\ &project && :: At\,\alpha \to Attrs \to \alpha \\ &project\,(e,p) && = p \end{aligned}$$

We shall ensure that for any attribute a, we have $project\ a \cdot embed\ a = id$. The opposite composition $embed\ a \cdot project\ a$ will usually not be the identity, because it always produces an attribution with only a single element.

One way to think of the expression $project\ a$ is as the function that maps a grammar symbol S to $S.a$: we project the a attribute from the attribution associated with S. Conversely, the embedding is what is used to define the attribute of a grammar symbol. Admittedly it may appear a little odd to define attributes in this way, but by encoding them as an embedding/projection pair, we avoid having to pass attribute names separately to many of the functions defined below.

Attributes are created using the function $mkAt$. It takes an attribute name, an embedding from α into the type of attribute values, and a coercion that goes in the opposite direction. The result is an attribute of type $At\,\alpha$:

$$\begin{aligned} &mkAt :: (AttrName, \\ &\qquad \alpha \to AttrValue, \\ &\qquad AttrValue \to \alpha) \to At\,\alpha \\ &mkAt\,(n,e,p) = (\lambda a \to \{(n, e\ a)\}, \\ &\qquad\qquad \lambda as \to p\,(as\ n)) \end{aligned}$$

That is, to embed an attribute value we wrap it in a singleton map, that only maps the name n to the value $e\ a$. Conversely, given an attribution as, we look up the corresponding value and project it to the type a.

In the running example, there are five attributes in all. First, there is the integer valued attribute of leaves – this is filled in by the scanner when the tree is read in. Furthermore, we have the local minimum, the global minimum, the newly created tree, and the counter. For each of these attributes, we introduce an identifier:

$$\begin{aligned} \textbf{data}\ AttrName = &\ ValueId \mid LocminId \\ \mid &\ GminId \mid NtreeId \\ \mid &\ CountId \end{aligned}$$

The type of attribute values is the disjoint union of values for each of these five attributes. For each attribute, we have a *constructor* that embeds the value into the union, and a *destructor* that projects it out of the union. In Haskell syntax, this reads:

$$\begin{aligned} &\textbf{data}\ AttrValue = \\ &\quad Value\{unvalue :: Int\} \mid \\ &\quad Locmin\{unLocmin :: Int\} \mid \\ &\quad Gmin\{unGmin :: Int\} \mid \\ &\quad Count\{unCount :: Int\} \mid \\ &\quad Ntree\{unNtree :: TreeProdNameAttrs\} \end{aligned}$$

Note the type of the attribute *ntree*: it is a tree whose *Fork* nodes are labelled with names of productions, and whose value nodes carry an attribution. Using the above constructors and destructors for $AttrValue$, we can now define the five attributes using the $mkAt$ function:

$$\begin{aligned} value &= mkAt\,(ValueId,\ Value,\ unvalue) \\ locmin &= mkAt\,(LocminId,\ Locmin, \\ &\qquad unlocmin) \\ gmin &= mkAt\,(GminId,\ Gmin,\ ungmin) \\ ntree &= mkAt\,(NtreeId,\ Ntree,\ unntree) \\ count &= mkAt\,(CountId,\ Count,\ uncount) \end{aligned}$$

We note once again that our use of embedding/projection pairs neatly hides the internal structure of an attribute, namely its name and its type. In an early version of this paper, we did not do so, and consequently we had to pass the triples of (name, constructor, destructor) around in many functions. That is rather clumsy, and it breaks the abstraction of an 'attribute' — we wish to hide the implementation detail as much as possible. The practical benefit is that the Haskell type system guarantees that each attribute can only be assigned values of the appropriate type.

To illustrate the above definitions, let us consider an example attribution from the *repmin* problem. An internal node might have the following inherited attribution:

$$\begin{aligned} &(embed\ gmin\ 4 \oplus embed\ count\ 5) = \\ &\quad \{(GminId, Gmin4), (CountId, Count5)\} \end{aligned}$$

Using the material presented so far we define the following tree construction functions for the *repmin* example:

$$\begin{aligned} leaf\ a &= Fork\ Leaf\ [Val\ (embed\ value\ a)] \\ node\ ss &= Fork\ Node\ ss \\ root\ t &= Fork\ Root\ [t] \end{aligned}$$

Note how the leaf data is stored in the *value* attribute. Now, one can construct an example tree thus:

$$\begin{aligned} &example :: Tree\ ProdName\ Attrs \\ &example = \end{aligned}$$

$$root\ (node\ [node\ [leaf\ 3, leaf\ 1],\\ node\ [leaf\ 4,\\ node\ [leaf\ 1, leaf\ 2]]])$$

It is worthwhile to reflect for a moment which parts of the semantics so far are dependent on the particular example at hand. The types of production names, attribute names and attribute values are specific. To get the semantics for other examples, new definitions have to be substituted.

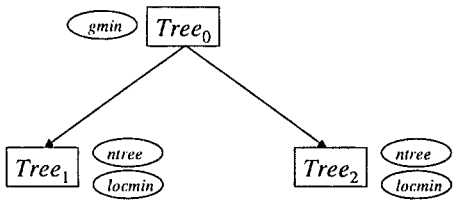
4 Composing attribute grammars

What are the building blocks of an attribute grammar? In their purest form, they are composed only of productions, and for each production, all attributes are defined simultaneously. Many attribute grammar systems also allow one to group definitions by *aspect*, where a number of related attributes are defined together, but not necessarily all attributes for each production. We have seen several examples of aspects in our running example. These aspects are however special in the sense that each defines only a single attribute. Aspects can be *woven* together to form a pure attribute grammar. Naturally one could see this as a syntactic operation, performed by a preprocessor, that simply collects all attribute definitions for each production from all aspects. We believe that it is beneficial to give a semantics to aspects, so that they are first-class values that can be returned as the result of functions, and passed as arguments. This section describes such a semantics. Experienced functional programmers may wish to glance ahead at Figure 3, which gives a summary of the types introduced in this section.

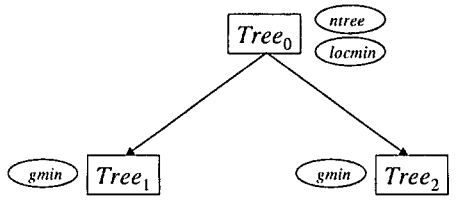
The first step towards defining a semantics is to use Haskell functions to model attribute definitions. Let us recall the traditional form of attribute definitions. The following code was used in our *repmin* example:

$$\begin{array}{lcl} Node & : & Tree_0 \rightarrow Tree_1\ Tree_2 \\ Tree_0.ntree & = & Node\ Tree_1.ntree \\ & & \quad Tree_2.ntree \\ Tree_0.locmin & = & min\ Tree_1.locmin \\ & & \quad Tree_2.locmin \\ Tree_1.gmin & = & Tree_0.gmin \\ Tree_2.gmin & = & Tree_0.gmin \end{array}$$

There are two ways of viewing this code. The first view is that there are four semantic functions that each define a single attribute. The second view is that there is one semantic rule that defines a set of attributes (containing four elements). We prefer the second view because it will allow us to easily define the *composition* of two semantic rules. So we shall define semantic rules to be functions from the set of *input attributes* to a subset of the *output attributes*. Input attributes are attributes such as $Tree_1.locmin$ above. They are the attributes that we are allowed to read from. The output attributes are the attributes that we are defining. This is illustrated in Figure 2.



The family of input attributes



The family of output attributes

Figure 2: Input and Output Attribute Families

It is useful to have a name for the sets of attributes used above. We shall call them families. The first subsection below formally defines families, and operators for composing them. Next, we turn to the definition of rules. A rule is a mapping between families, namely from the input attributes of a production to some of its output attributes. Once rules are defined, it is possible to formalise the notion of an aspect. An aspect assigns rules to a number of production names. The remainder of this section shows how aspects can be built and combined in various ways, including those that were illustrated in the introduction.

4.1 Families

Families are used to model sets of input attributes or sets of output attributes. Therefore, a *family* consists of an attribution for a parent node, and an attribution for each of its children. That is, it is a pair that consists of an attribution, and a list of attributions:

$$\mathbf{type}\ Fam\ =\ (Attrs, [Attrs])$$

For concreteness, let us consider the family of input attributes associated with the *Node* production in *repmin*. These attributes are depicted in Figure 2. Below is an instantiation of the family, in which values have been assigned to all the attributes:

$$(\ \{\ (GminId,\ Gmin\ 2)\ \},$$

$$
\begin{array}{l}
[\,\{\,(\mathit{NtreeId},\ \mathit{Ntree}\ (\mathit{Fork}\ \ldots)),\\
\quad (\mathit{LocminId},\ \mathit{Locmin}\ 3)\,\},\\
\quad \{\,(\mathit{NtreeId},\ \mathit{Ntree}\ (\mathit{Val}\ \ldots)),\\
\quad (\mathit{LocminId},\ \mathit{Locmin}\ 5)\,\}\,]\,)
\end{array}
$$

Following our design principle that each new concept should have a corresponding join operation, we now define the empty family, and joining of families. Not surprisingly, we can do so by lifting the earlier definitions on attributions in an appropriate way.

The simplest family of all has empty attributions, and an infinite number of children:

$$
\begin{array}{l}
\emptyset\ ::\ \mathit{Fam}\\
\emptyset\ =\ (\emptyset,\ \mathit{repeat}\ \emptyset)
\end{array}
$$

The function $\mathit{repeat}\ ::\ \alpha \rightarrow [\alpha]$ generates an infinite list of copies of its argument. Again we are taking a minor notational liberty here, by overloading the notation for the empty map to also apply to the empty family. In Haskell, the two would have to be separated, or overloaded via a so-called *type class*. We shall use the same illicit overloading in the definition of the join operator on families.

Two families can be *joined* by joining their parents, and joining their children position-wise. Informally, we have:

$$
\begin{array}{l}
(s, [cs_0, cs_1, \ldots]) \oplus (t, [ct_0, ct_1, \ldots])\\
=\\
(s \oplus t,\ [cs_0 \oplus ct_0,\ cs_1 \oplus ct_1,\ \ldots])
\end{array}
$$

In Haskell, this is achieved via the function call $\mathit{zipWith}\ f\ xs\ ys$ which applies the function f to corresponding elements of the lists xs and ys. Furthermore, the length of the result of $\mathit{zipWith}\ f$ is the minimum of the length of its arguments. We have:

$$
\begin{array}{lll}
(\oplus) & ::\ \mathit{Fam} \rightarrow \mathit{Fam} \rightarrow \mathit{Fam}\\
(s, cs) \oplus (t, ct) & = (s \oplus t,\\
& \quad \mathit{zipWith}\ (\oplus)\ cs\ ct)
\end{array}
$$

Again the operator $(\oplus)$ is associative, and has unit $\emptyset$.

4.2 Rules

As we discussed earlier, a rule is a mapping from the *input attributes* of a production to some of its *output attributes*. Since both input and output attributes can be modelled as families, we define the type of attribute definition rules as:

$$\textbf{type}\ \mathit{Rule} = \mathit{Fam} \rightarrow \mathit{Fam}$$

To illustrate, consider the rule that defines the *locmin* attribute in the *Node* production of *repmin*. In the traditional notation we employed in the introduction, it reads:

$$
\begin{array}{lll}
\mathit{Tree}_0.\mathit{locmin} & = & \mathit{min}\ \mathit{Tree}_1.\mathit{locmin}\\
& & \quad \mathit{Tree}_2.\mathit{locmin}
\end{array}
$$

Encoded as an element of the above type, it becomes the function:

$$
\begin{array}{l}
\lambda\,(\mathit{tree}_0, [\mathit{tree}_1, \mathit{tree}_2]) \rightarrow\\
\quad (\mathit{embed}\ \mathit{locmin}\\
\qquad (\mathit{min}\ (\mathit{project}\ \mathit{locmin}\ \mathit{tree}_1)\\
\qquad\quad (\mathit{project}\ \mathit{locmin}\ \mathit{tree}_2)),\\
\quad [\emptyset, \emptyset])
\end{array}
$$

Note that in the resulting family, both children have empty attributions. If we instead encode two rules simultaneously, say both of

$$
\begin{array}{lll}
\mathit{Tree}_0.\mathit{locmin} & = & \mathit{min}\ \mathit{Tree}_1.\mathit{locmin}\\
& & \quad \mathit{Tree}_2.\mathit{locmin}\\
\mathit{Tree}_2.\mathit{gmin} & = & \mathit{Tree}_0.\mathit{gmin}
\end{array}
$$

we would have a non-empty attribution for the second child:

$$
\begin{array}{l}
\lambda\,(\mathit{tree}_0, [\mathit{tree}_1, \mathit{tree}_2]) \rightarrow\\
\quad (\mathit{embed}\ \mathit{locmin}\\
\qquad (\mathit{min}\ (\mathit{project}\ \mathit{locmin}\ \mathit{tree}_1)\\
\qquad\quad (\mathit{project}\ \mathit{locmin}\ \mathit{tree}_2)),\\
\quad [\emptyset, \mathit{embed}\ \mathit{gmin}\ (\mathit{project}\ \mathit{gmin}\ \mathit{tree}_0)])
\end{array}
$$

Note that we have chosen suggestive identifiers in the argument family, but these are merely local names. In the rule itself, no knowledge of the nonterminals of the underlying context free grammar has been encoded. This has certain advantages, in particular that one can give rules that are independent of the precise form of the production that they will be associated to. The main disadvantage is that the notation can be a little hairy to use in practice: although we already named descendants in a production, those same names have to be repeated in each rule associated with the production.

Let us now consider some operations for manipulating rules. By lifting the corresponding operations on families, we get an empty rule (that does not define any attributes) and a join operator:

$$
\begin{array}{lll}
\emptyset & ::\ \mathit{Rule}\\
\emptyset\, f & = \emptyset
\end{array}
$$

$$
\begin{array}{lll}
(\oplus) & ::\ \mathit{Rule} \rightarrow \mathit{Rule} \rightarrow \mathit{Rule}\\
(r_1 \oplus r_2)\, f & = (r_1\, f) \oplus (r_2\, f)
\end{array}
$$

It is at this point that we can start introducing some shorthands for common vocabulary in attribute definitions. For example, here is an operator that generates a copy rule, which simply copies an inherited attribute from the parent to all the children:

$$
\begin{array}{l}
\mathit{copyRule}\ ::\ \mathit{At}\ \alpha \rightarrow \mathit{Rule}\\
\mathit{copyRule}\ (e, p)\ (\mathit{inhp}, \mathit{syncs}) =\\
\quad (\emptyset, \mathit{repeat}\ (e\ (p\ \mathit{inhp})))
\end{array}
$$

Another common design pattern is to collect synthesised attributes off all the children. Here we need a function *collect* that maps a list of attribute values to a single value:

$$
\begin{aligned}
&collectRule :: At\,\alpha \to ([\alpha] \to \alpha) \to Rule\\
&collectRule\ (e,p)\ collect\ (inhp, syncs) =\\
&\quad (e\ (collect\ (map\ p\ syncs)), repeat\ \emptyset)
\end{aligned}
$$

The function *map p syncs* applies the projection p to each of the synthesised attributions of the children. We are assuming, therefore, that each of the children does indeed possess the attribute in question.

Finally, here is a formulation of the notion of *chain rule*. It takes an attribute, and it returns a rule that threads the attribute from left to right, before defining the synthesised occurrence at the parent:

$$
\begin{aligned}
&chainRule :: At\,\alpha \to Rule\\
&chainRule\ (e,p)\ (inhp, syncs) =\\
&\quad (last\ output,\ init\ output)\\
&\quad \textbf{where}\ output = map\ (e \cdot p)\ input\\
&\qquad\qquad input\ \ = inhp : syncs
\end{aligned}
$$

First we take all the input attributions as a list, by prefixing the synthesised attributions of the children by the inherited attribution of the parent: this yields the list named *input*. We then apply the composite function $e \cdot p$ to each of the elements of *input*: this yields the list *output*. Finally we return the last element of *output* as the synthesised attribution of the parent, and all but the last element as the inherited attributions of the children. Note that this definition is completely independent of how many children there are. In particular, if there was no child at all ($syncs = []$), the attribute is copied unchanged from the inherited attribution to the synthesised attribution.

Undoubtedly some readers will prefer subtly different definitions of these common patterns: hopefully they will be encouraged by the simplicity of our choices to try and formulate their own in the present framework.

4.3 Aspects

Often we wish to group together rules that define related attributes, across multiple productions. For instance, we might wish to group together all attribute definitions that relate to type checking, or to a particular data flow analysis. Such a group of related rules is called an *aspect*, following terminology in object-oriented design [17]. Formally, we define:

$$\textbf{type}\ Aspect = ProdName \mapsto Rule$$

In words, an aspect maps production names to rules. It is not necessary for an aspect to map every production name in a grammar to a rule: it can be a partial function. We have already discussed several aspects in the introduction to the *repmin* problem, and we shall see shortly how these can be expressed as elements of the above type.

Again we can lift the empty and join operators to operate on aspects, and again we shall write $\emptyset$ for the empty aspect, and $\oplus$ for join. The empty aspect is simply the empty finite map. The join operator is a little more subtle than before, due to the fact that aspects may be partial:

$$
\begin{aligned}
(f \oplus g)\ x &= f\ x, \textbf{ if } x \notin domain\ g\\
&= g\ x, \textbf{ if } x \notin domain\ f\\
&= f\ x \oplus g\ x, \textbf{ otherwise}
\end{aligned}
$$

An aspect is often defined by giving a default rule for most productions (for instance a copy rule for an inherited attribute), supplemented by specific rules for only a handful of the productions. To build the default aspect, we have the operator:

$$
\begin{aligned}
&defaultAspect :: Rule \to [ProdName]\\
&\qquad\qquad \to Aspect\\
&defaultAspect\ r\ ls\ = \{\ (l,r)\ |\ l \leftarrow ls\}
\end{aligned}
$$

It maps each production name l (of type *ProdName*) to the same rule r. Strictly speaking the above is not valid Haskell, as we have made up the set comprehension notation for finite maps, for increased readability.

In practice it is somewhat inconvenient to specify rules and aspects directly. Therefore, to make the interface of this library for composing attribute grammars a little less forbidding, we introduce the notion of *attribute aspects*. An attribute aspect is like an ordinary aspect, but it defines values only for a single attribute of type α. Formally, it is a finite map from production names to functions of type $Fam \to \alpha$ (ordinary aspects have a result of type $Rule = Fam \to Fam$):

$$
\begin{aligned}
&\textbf{type}\ AtAspect\ \alpha =\\
&\quad ProdName \mapsto (Fam \to \alpha)
\end{aligned}
$$

An attribute aspect can be converted into a proper aspect by applying the function *inh* (for inherited attributes) or *synth* (for synthesised attributes). In the case of an inherited attribute, the attribute aspect defines a list of values, one for each descendant:

$$
\begin{aligned}
&inh :: At\,\alpha \to AtAspect\,[\alpha] \to Aspect\\
&inh\ a\ atAspect\ pname\ f\ =\\
&\quad (\emptyset,\ map\ (embed\ a)\ (atAspect\ pname\ f))
\end{aligned}
$$

$$
\begin{aligned}
&synth :: At\,\alpha \to AtAspect\,\alpha \to Aspect\\
&synth\ a\ atAspect\ pname\ f\ =\\
&\quad (embed\ a\ (atAspect\ pname\ f),\ repeat\ \emptyset)
\end{aligned}
$$

(In these definitions, we are again taking the liberty of mixing the notation of ordinary functions with that for finite maps.) Using the above operators, we can define a primitive that defines an inherited attribute that is mostly copied, except in a few productions that are specified as an attribute aspect:

$$
\begin{aligned}
&inherit :: At\,\alpha \to [ProdName]\\
&\qquad \to AtAspect\,[\alpha] \to Aspect\\
&inherit\ a\ pnames\ atAspect\ =\\
&\quad inh\ a\ atAspect \oplus\\
&\quad defaultAspect\ (copyRule\ a)\ pnames
\end{aligned}
$$

Astute readers will recognise this as a desugared version of the *inherit* construct that was introduced earlier in this paper:

inherit ⟨attribute a⟩
copy at ⟨list of production names *pnames*⟩
define at ⟨attribute aspect *atAspect*⟩

The only difference is that the list of values was more conveniently specified in the introduction, by listing symbol occurrences in the relevant production. Of course such syntactic sugar is easily added by a simple preprocessor.

Similarly, one obtains the semantic counterpart for the *synthesise* construct in the introduction. There a synthesised attribute is collected in a specified list of productions, and defined elsewhere through an attribute aspect:

$$
\begin{aligned}
&\mathit{synthesise} :: \mathit{At}\,\alpha \to ([\alpha] \to \alpha) \\
&\qquad\qquad \to [\mathit{ProdName}] \\
&\qquad\qquad \to \mathit{AtAspect}\,\alpha \to \mathit{Aspect} \\
&\mathit{synthesise}\ a\ \mathit{coll}\ \mathit{pnames}\ \mathit{atAspect} = \\
&\quad \mathit{synth}\ a\ \mathit{atAspect} \\
&\quad \oplus \\
&\quad \mathit{defaultAspect}\ (\mathit{collectRule}\ a\ \mathit{coll})\ \mathit{pnames}
\end{aligned}
$$

Finally, a chained attribute is defined by specifying two attribute aspects. The first gives the initialisations (which are inherited) and the second gives the update rules (which are synthesised):

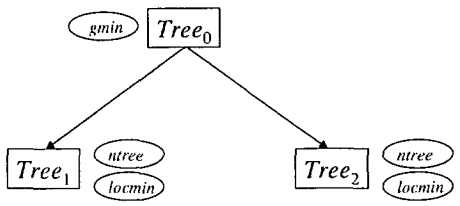
$$
\begin{aligned}
&\mathit{chain} :: \mathit{At}\,\alpha \to [\mathit{ProdName}] \\
&\qquad \to \mathit{AtAspect}\,[\alpha] \to \mathit{AtAspect}\,\alpha \\
&\qquad \to \mathit{Aspect} \\
&\mathit{chain}\ a\ \mathit{pnames}\ \mathit{atAspect}_0\ \mathit{atAspect}_1 = \\
&\quad \mathit{inh}\ a\ \mathit{atAspect}_0 \\
&\quad \oplus \\
&\quad \mathit{synth}\ a\ \mathit{atAspect}_1 \\
&\quad \oplus \\
&\quad \mathit{defaultAspect}\ (\mathit{chainRule}\ a)\ \mathit{pnames}
\end{aligned}
$$

The definition of commonly occurring patterns such as *inherit*, *synthesise* and *chain* as first-class values was our original motivation for introducing the notion of aspects. For reasons of exposition, we have chosen the simplest possible definitions of these aspects, and not the most general ones. It should furthermore be noted that aspects do not necessarily define a single attribute, and so one can also define more complex patterns involving multiple attributes. Kastens and Waite [16] discuss techniques for encoding common attribution patterns in much greater detail. Many of our examples (in particular the *chain* and *synthesise* functions) were borrowed from their paper.

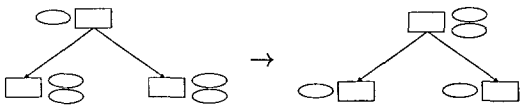
5 The *repmin* example revisited

Using the definitions of the previous section, we can return to the example introduced at the beginning of this paper. This will illustrate the use of families, rules and aspects in practice. The reader may find it helpful to refer to Figure 3, which summarises the definitions of the previous section.

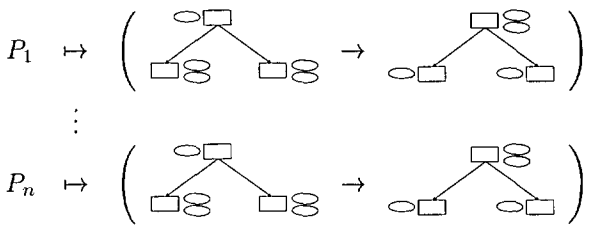
The first aspect is that of the global minimum. The global minimum is an inherited attribute that is broadcast



type $\mathit{Fam} = (\mathit{Attrs}, [\mathit{Attrs}])$



type $\mathit{Rule} = \mathit{Fam} \to \mathit{Fam}$



type $\mathit{Aspect} = \mathit{ProdName} \mapsto \mathit{Rule}$

Figure 3: A summary of Section 4

to all nodes through a copy rule. It follows that we only have to define its value at the root. There it equals the local minimum of the immediate descendant:

$$
\begin{aligned}
&\mathit{gmins} :: \mathit{Aspect} \\
&\mathit{gmins} = \mathit{inherit}\ \mathit{gmin}\ [\mathit{Node}] \\
&\quad \{\, (\mathit{Root},\ \lambda\,(\mathit{start}, [\mathit{tree}]) \to \\
&\qquad\qquad [\mathit{project}\ \mathit{locmin}\ \mathit{tree}]\,)\,\}
\end{aligned}
$$

By contrast, the local minimum is a synthesised attribute that is collected from all descendants using the function *minlist* that returns the minimum of a list of integers. For leaves, the local minimum is defined to be the value of the single child:

$$
\begin{aligned}
&\mathit{locmins} :: \mathit{Aspect} \\
&\mathit{locmins} = \mathit{synthesise}\ \mathit{locmin}\ \mathit{minlist}\ [\mathit{Node}] \\
&\quad \{\, (\mathit{Leaf},\ \lambda\,(\mathit{leaf}, [\mathit{val}]) \to \\
&\qquad\qquad \mathit{project}\ \mathit{value}\ \mathit{val})\,\}
\end{aligned}
$$

It remains to define the aspect that produces new trees. Recall that we had two versions of the example problem, which differed in the value that had to be substituted for leaves. To cater for that difference, we parameterise the aspect by the attribute that is the value to substitute at leaves. At the root, and at ordinary nodes, we collect new trees of the descendants using the *node* constructor. At the leaves, we substitute the argument attribute:

```
ntrees   :: At Int → Aspect
ntrees a = synthesise ntree node [Node]
  { (Leaf, λ (leaf, [val]) →
            leaf (project a leaf)),
    (Root, λ (start, [tree]) →
            project ntree tree) }
```

The first and simplest version of the example can now be assembled into a compiler, by passing the global minimum attribute to the *ntrees* aspect: (The Haskell function *compiler* that we use here is explained in the next section.)

```
repmin0 :: Tree ProdName Attrs
           → Tree ProdName Attrs
repmin0 = compiler [gmins,
                    locmins,
                    ntrees gmin] ntree
```

The more complicated version of the example required that we replace each leaf L by the number of times the global minimum occurs to the left of L. To program that variant, we first introduce an aspect for the counter:

```
counts :: Aspect
counts = chain count [Node]
  { (Root, λ (start, [tree]) → [0]) }
  { (Leaf, λ (leaf, [val]) →
            let v1 = project value val in
            let v2 = project gmin leaf in
             if v1 == v2 then
               project count leaf + 1
             else
               project count leaf ) }
```

The new compiler is similar to the old one, except that we now weave in the counter aspect, and we pass the *count* attribute to the *ntrees* aspect:

```
repmin1 :: Tree ProdName Attrs
           → Tree ProdName Attrs
repmin1 = compiler [gmins,
                    locmins,
                    ntrees count,
                    counts] ntree
```

6 Mapping Aspects to Compilers

In this section we shall explain how *families*, *rules* and *aspects* can be mapped to an executable implementation. We shall use the well known method of encoding attribute grammars as lazy functional programs [14, 18]. We shall give a brief introduction to this encoding, but the reader will benefit from an acquaintance with the work of Johnsson [14] and Swierstra [18]. A more recent paper by Swierstra [25] approaches the problem from a wider perspective and gives some non-trivial examples.

6.1 The Encoding

The method of encoding attribute grammars as lazy functional programs is based on the following observation: the semantics of a tree can be modelled as a *function*. This function is parameterised by the inherited attributes of the root of the tree and computes the synthesised attributes of the root. In other words, it is a function of type $Attrs \rightarrow Attrs$. This observation is valid for the following reason: if we instantiate the inherited attributes of the root of the tree, then the attribution rules tell us how to fully decorate the tree. Therefore, the synthesised attributes of the root depend functionally on the inherited attributes. Figure 4 gives an illustration of this.

In this section we shall frequently be manipulating functions of type $Attrs \rightarrow Attrs$. These functions represent the semantics of a tree, so we shall define the following shorthand:

```
type SemTree = Attrs → Attrs
```

A production is a tree constructor: it takes a list of trees (the children) and constructs a new tree. We said above that each of the children is modelled by a function of type *SemTree*. Therefore, the semantics of the production can be modelled by a function with the following type:

```
type SemProd = [SemTree] → SemTree
```

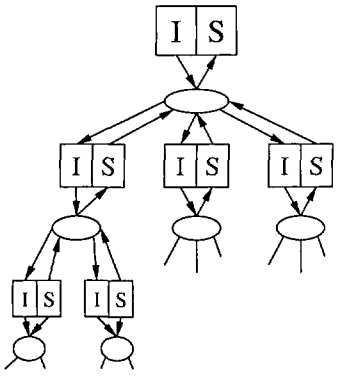
Once we have modelled the productions of an attribute grammar with functions of type *SemProd*, an evaluator for the grammar is constructed as follows: the evaluator recursively applies the semantic productions to the input tree. The result is a function of type *SemTree*, which represents the semantics of the tree. Below, we shall explain how rules can be mapped to semantic productions. Then we shall explain how aspects can be mapped to evaluators.

6.2 Mapping Rules to Semantic Productions

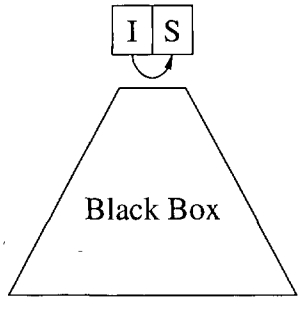
The conversion of rules to semantic productions is performed by the operation *knit*. Given a rule r and the semantics of the children fs, it should map the inherited attribution of the root to its synthesised attribution. To obtain the synthesised attributes of the root, as well as the inherited attributes of the children, we can simply apply the rule r. It remains to compute the synthesised attributes of the children: this we do by applying, for each child, the semantics to the inherited attributes. In sum, the definition of *knit* reads:

```
knit                :: Rule → SemProd
knit r fs inhroot   = synroot
  where
    (synroot, inhcs) = r (inhroot, syncs)
    syncs            = applyList fs inhcs
```

Note the cyclic definition of *synRoot*. Here we are relying on the lazy semantics of Haskell, so a similar definition



The semantics of a tree



A functional view of the semantics

Figure 4: Modelling tree semantics as a function

would not work directly in a strict language such as ML. The function *applyList* takes a list of functions and applies them pointwise to a list of values. Its definition is as follows:[1]

$$
\begin{aligned}
&applyList\ [\,]\ xs\ =\ [\,]\\
&applyList\ (f : fs)\ \sim(x : xs)\ =\\
&\quad f\ x\ :\ applyList\ fs\ xs
\end{aligned}
$$

The use of laziness is crucial to the success of this implementation. Without it, we would need to analyse each rule and determine an evaluation order for the attributes. This would prevent us from defining a single *knit* function that can be applied to any rule. Using laziness, the evaluation order is determined at runtime [24].

6.3 Mapping Aspects to Evaluators

We define an *attribute grammar* to be a finite map from production names to production semantics:

$$\textbf{type}\ AG\ =\ ProdName\ \mapsto\ SemProd$$

To convert an aspect to an attribute grammar, all that needs to be done is to knit each rule in its range. This is achieved by composing the aspect with the knit function:

$$
\begin{aligned}
&knitAspect\ \ \ \ ::\ Aspect\ \rightarrow\ AG\\
&knitAspect\ as\ =\ knit\ \cdot\ as
\end{aligned}
$$

The evaluator is a function that recursively applies the attribute grammar to the input tree.

Attribute grammars define translators, which take a tree and an inherited attribution, and which produce a synthesised attribution. To translate a *Fork* node, we translate its descendants, and apply the semantics of the relevant production. To translate a *Val* node, we return its attribution:

$$
\begin{aligned}
&trans\ ::\ AG\ \rightarrow\ Tree\ ProdName\ Attrs\\
&\qquad\rightarrow\ SemTree\\
&trans\ ag\ (Val\ a)\ inh\ =\ a\\
&trans\ ag\ (Fork\ l\ ts)\ inh\ =\\
&\quad ag\ l\ (map\ (trans\ ag)\ ts)\ inh
\end{aligned}
$$

Naturally we are usually interested only in the value of a single attribute; furthermore the compiler is often specified as a set of aspects. That common vocabulary is captured by the definition:

$$
\begin{aligned}
&compiler\ ::\ [Aspect]\ \rightarrow\ At\ \alpha\\
&\qquad\rightarrow\ Tree\ ProdName\ Attrs\ \rightarrow\ \alpha\\
&compiler\ ass\ a\ t\ =\\
&\quad project\ a\ (trans\ (knitAspect\ as)\ t\ \emptyset)\\
&\quad \textbf{where}\ as\ =\ foldr\ (\oplus)\ \emptyset\ ass
\end{aligned}
$$

In words, we take a list of aspects ass, an attribute a, and a tree t. The aim is to produce the synthesised value of attribute a at the root of t. To that end, we first join all the aspects in $ass\ =\ [as_0, as_1, \ldots, as_{k-1}]$ to obtain a single aspect $as\ =\ as_0\ \oplus\ (as_1\ \oplus\ \ldots\ (as_{k-1}\ \oplus\ \emptyset))$. We then apply the corresponding translator to the tree t, giving it the empty attribution to start with. That produces the synthesised attribution of of the root; projecting on a gives the desired result.

It is sometimes handy to decorate the tree with all its attributions, both inherited and synthesised. Doing so is in fact no more difficult than the above compiler. Afficionadoes of the traditional encoding of attribute grammars in the functional paradigm (which foregoes the notion of an aspect) may wish to contemplate whether this operation can be written with the same efficiency as the one below:

$$
\begin{aligned}
&scan\ ::\ Aspect\ \rightarrow\ Tree\ ProdName\ Attrs\\
&\qquad\rightarrow\ Attrs\ \rightarrow\\
&\qquad Tree\ (ProdName, Attrs, Attrs)\ Attrs\\
&scan\ as\ (Val\ a)\ i\ =\ Val\ a\\
&scan\ as\ (Fork\ l\ ts)\ i\ =\ Fork\ (l, i, s)\ ts'\\
&\quad \textbf{where}\\
&\qquad (s, ics)\ =\ as\ l\ (i, map\ syn\ ts')\\
&\qquad ts'\ =\ applyList\ (map\ (scan\ as)\ ts)\ ics\\
&\qquad syn\ (Fork\ (l, inh, s)\ ts)\ =\ s
\end{aligned}
$$

[1] *(For Haskell connoisseurs)* This definition contains a strictness annotation, which makes the function strict only in its first argument. This slightly simplifies the use of *knit* in practice, as it relieves users of the duty to be careful about strictness in defining attribution rules.

7 Conclusion

We have presented a semantic view of attribute grammars, embedded as first-class values in the lazy functional programming language Haskell. Naturally we regard it as a benefit that our definitions are executable, but perhaps the more important contribution is the compositional semantics that we have given to the attribute grammar paradigm. It is hoped that this compositional semantics will yield further insight into making attribute grammars more flexible, encouraging the reuse of existing code where possible.

The utility of the semantics as an executable prototype is severely marred by the absence of static checks such as closure (each attribute that is used is also defined), and the circularity check (definitions do not depend on each other in a cyclic way). It is not difficult to add these checks, however, namely by providing an abstract interpretation of attribute values, and of the semantic functions. One can use this approach to compute the dependencies for each production separately, or even to generate the text of the composed attribute grammar, which could then be presented to a traditional attribute evaluator. Full details can be found in the literate Haskell program that accompanies this paper [7].

In earlier work we presented some of the same ideas via an encoding in a Rémy-style record calculus [8]. That encoding has the advantage that one can check for closure of the definitions through type inference. We found, however, that the approach was too restrictive, and made the definition of a number of important operations (such as a combinator for introducing chained attributes) exceedingly cumbersome. It is conceivable, however, that a more appropriate type system can be found, which offers the same guarantees (in particular that each attribute is defined precisely once), without the restrictions. We are however pessimistic that such a type system will allow full type inference, and that the types will be of manageable size. Very recently, Azero and Swierstra have succeeded in simplifying our original approach through the use of novel mechanisms for resolving overloading in Haskell — but the basic drawbacks of the approach remain. While preparing the present paper, we learned that the idea to model the semantics of attribute grammars through record calculus is not new: it was first suggested by Gondow and Katayama in the Japanese literature [10].

The examples of aspects given in this paper do not demonstrate the full potential of *production names* being first-class values. Every production that an aspect annotates is explicitly listed. For example, the *locmins* aspect individually lists the *Node* and *Leaf* productions. In larger grammars it would often be useful to work with sets of productions. For example, we could compute the set of productions that might appear on a path from nonterminal X to nonterminal Y. We could then annotate every production in that set with a default computation. Production names are first-class values, so we can easily define functions that manipulate them in this way.

Acknowledgements

We have much benefited from discussions with Pablo Azero on the topic of this paper. Tony Hoare's comments on an earlier paper [8] inspired some of the improvements presented here. We would also like to thank Atze Dijkstra, Jeremy Gibbons and João Saraiva for their helpful comments on the paper. We are grateful to the Programming Tools Group at Oxford for many enjoyable research meetings where these ideas took shape. Kevin Backhouse is supported by a studentship from Microsoft Research.

References

[1] S. Adams. *Modular Attribute Grammars for Programming Language Prototyping*. Ph.D. thesis, University of Southampton, 1991.

[2] A. Augusteijn. *Functional programming, program transformations and compiler construction*. PhD thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands, 1993. See also: `http://www.research.philips.com/generalinfo/special/elegant/elegant.html`.

[3] R. S Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[4] R. S Bird. *Introduction to Functional Programming in Haskell*. International Series in Computer Science. Prentice Hall, 1998.

[5] L. Correnson, E. Duris, and D. Parigot. Declarative program transformation: a deforestation case-study. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 353–369. Springer Verlag, 1999.

[6] L. Correnson, E. Duris, and D. Parigot. Equational semantics. In A. Cortesi and G. Filé, editors, *Symposium on Static Analysis SAS '99*, volume 1694 of *Lecture Notes in Computer Science*, pages 264–283. Springer Verlag, 1999.

[7] O. De Moor, K. Backhouse, and Swierstra S. D. First-class attribute grammars: Haskell code. Available from: `http://web.comlab.ox.ac.uk/oucl/research/areas/progtools/waga.zip`, 2000.

[8] O. De Moor, S. Peyton-Jones, and E. Van Wyk. Aspect-oriented compilers. In *First International Symposium on Generative and Component-based Software Engineering*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[9] C. Elliott. Functional implementations of continuous modeled animation. In *Proceedings of PLILP/ALP '98*, volume 1490 of *Lecture Notes in Computer Science*, pages 284–299, 1998.

[10] K. Gondow and T. Katayama. Attribute grammars as record calculus. Technical report 93TR-0047, Tokyo Institute of Technology, 1993.

[11] 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*, 35:121–131, 1992.

[12] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation — an algebra of music. *Journal of Functional Programming*, 6:465–484, 1996.

[13] R. J. M. Hughes. The design of a pretty-printing library. In *Advanced Functional Programming*, volume 1995 of *Lecture Notes in Computer Science*, pages 53–96, 1995.

[14] T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer-Verlag, 1987.

[15] M. Jourdan, D. Parigot, Julié, O. Durin, and C. Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Conference on Programming Languages Design and Implementation*, pages 209–222, 1990. Published as *ACM SIGPLAN Notices*, 25(6).

[16] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.

[17] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28A(4), 1996.

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

[19] J. Paakki. Attribute grammar paradigms – a high-level methodology in language implementations. *ACM Computing Surveys*, 27(2):226–255, 1995.

[20] M. Pennings. *Generating incremental evaluators*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 1994. Available from: `ftp://ftp.cs.ruu.nl/pub/RUU/CS/phdtheses/Pennings/`.

[21] C. Ramming (editor). *Proceedings of the Usenix conference on Domain-Specific Languages '97*. USENIX, 1997.

[22] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A system for constructing language-based editors*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[23] J. Saraiva. *Purely functional implementation of attribute grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 1999. Available from: `ftp://ftp.cs.ruu.nl/pub/RUU/CS/phdtheses/Saraiva/`.

[24] J. Saraiva and S. D. Swierstra. Data Structure Free Compilation. In Stefan Jähnichen, editor, *8th International Conference on Compiler Construction - ETAPS'99*, volume 1575 of *LNCS*, pages 1–16. Springer-Verlag, March 1999.

[25] S. D. Swierstra, P. Azero Alcocer, and P. Saraiva. Designing and implementing combinator languages. In S. D. Swierstra, editor, *Third International Summer School on Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 150–206. Springer-Verlag, 1998.

[26] H. Vogt. *Higher order attribute grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 1989.

Equational Semantics

Loïc Correnson
INRIA - Rocquencourt
Loic.Correnson@polytechnique.org

Keywords: program transformation, partial evaluation, deforestation.

Edited by: Marjan Mernik
Received: June 22, 1999 **Revised:** September 5, 2000 **Accepted:** September 11, 2000

Attribute grammars are well-designed to construct complex algorithms by composing several ones together. Actually, there exists a powerful transformation called descriptional composition which highly simplifies the composition of two attribute grammars by removing useless intermediate constructions.
However, most of non-linear algorithms can not be expressed with attribute grammars. Thus, many compositions can not be simplified by the decriptional composition. In this paper, we present Equational Semantics, a formalism largely inspired by attribute grammars but where non-linear algorithms can be encoded. More precisely, instead of being restricted to one input static tree as it is the case for attribute grammars, an algorithm encoded with Equational Semantics may use dynamically constructed trees.
This formalism consists in an very poor abstract syntax. We present its semantics and some of its transformations such as partial evaluation and decriptionnal composition (also called deforestation). In some sense, Equational Semantics is a kind of lambda-calculus dedicated to program transformations.

1 Introduction

For many years, we try to promote our approach for generic programming and software reuse. It consists in composing different basic components together in order to produce more complex ones. Each basic component must be robust and general, so using them in particular cases may be costly because of some translation components or unspecialized algorithms.

Attribute grammars seems to be an interesting model to deal with this kind of generic programming since there is an algorithm, the descriptional composition [7, 8, 14], which simplifies a composition and produces a new and more efficient attribute grammar. However, this descriptional composition may fail: for instance, it may produce multiple definitions for an attribute, or it may introduce a circularity into attribute dependences.

More generally, an attribute grammar can only encode an algorithm which is linear in the number of nodes of its input tree. A syntactic reason for this is the impossibility to dynamically compute over attributes that are not linked to the input tree of the attribute grammar. The key point of our approach consists in removing this impossibility.

Let us consider the following example written with a straightforward notation. It defines an attribute grammar which computes the length of a list and its reversed list (with an accumulator). The first part of the attribute grammar introduces type definitions:

```
type list, int
constructors
  cons : int * list → list
  nil : → list
synthesized(list) = rev : list length : int
inherited(list) = accu : list
```

Then the core of the attribute grammar comes up:

```
cons x1 x2 :
  rev = x2.rev
  x2.accu = (cons x1 accu)
  length = (+ 1 x2.length)
nil :
  rev = accu
  length = 0
```

In this example, there is a functional dependency between the inherited attribute *accu* and the synthesized one *rev*: the expression $x2.rev$ can be seen as a call to some function (or procedure, or visit, or whichever is appropriate) which computes on the sub-tree $x2$ the synthesized attribute *rev* with respect to the value of its inherited attribute *accu*.

Actually, in classical attribute grammars, it is only possible to use these "function calls" on a sub-tree of the (static) input tree of the program. With such a restriction, it is impossible to consider calls on dynamically-constructed trees or multiple calls on one sub-tree with different values for its inherited attributes. This is why an attribute grammar can only encode linear algorithms. Then the key point of

our approach consists in introducing local definition, such as:

$$L1 = (cons\ x1\ x2.rev)$$

Then, we allow to use expressions like $L1.rev$ and to define a value for $L1.accu$. Thus, it becomes possible to define the reverse of the reverse of a list:

```
type unit
constructors
  reverse : list → unit
synthesized(unit) = r : list
reverse x1 :
  r = L1.rev
  L1.accu = (nil)
  L1 = x1.rev
  x1.accu = (nil)
```

Here, to compute the attribute r of the tree $(reverse\ l)$, the list l is reversed, and this dynamically constructed list is also reversed. This algorithm is still linear, but such dynamic constructions allow to encode non-linear algorithm. See section 4 for more examples.

But introducing such syntactic features merely modify the semantics of attribute grammars. Actually, we must completely redefine it. This is why we propose a new formalism, where we only kept the essential of attribute grammars to deal with program transformations, namely the notion of constructors and attributes.

The result is a kind of lambda-calculus, with a notation closed to the one of attribute grammars, especially dedicated to program transformations. We called this formalism Equational Semantic and it is presented in the section 3 of this paper. Section 4 provides examples. Section 5 is a short presentation of how to generate evaluators which compute the attributes of a tree. In section 6, we define what should be a *correct* transformation. Section 7 describes transformations, especially partial evaluation and deforestation.

2 Related Works

There exists a lot of extensions to attribute grammars. A common goal for them is to enlarge the expressiveness of standard attribute grammars. We want to mention here higher-order attribute grammars [16], tree-transducers [10], and dynamic attribute grammars [13]. Since all of them are able to encode $\lambda-$calculus, we will not expose in this article why and how their equivalence holds. We are interested here in showing an extension of an attribute grammar transformation method, the descriptional composition, which applies to non-linear programs thanks to Equational Semantics.

With another point of vue, Equational Semantics not only aims at improving attribute grammar transformations. We also use this formalism to unify both functional-programming and attribute-grammars semantics and transformations, as described in [1] and [2].

3 Equational Semantics

This section defines notions and vocabulary for the equational semantics formalism.

Terms: Terms are built using *constructors* or *primitives* which take variables or sub-terms as parameters. There is no function call.

Variables: They name or represent terms. A variable can have several forms:

- $x.k$ (k is an integer) represents the k-th sub-term of (the term represented by) the variable x.
- $x.a$ (a is an attribute name) represents the attribute a attached to the variable x.
- $x.L_k$ (k is an integer) represents a local variable associated to the variable x.

The special variable α is used as a root variable.

Attributes: An attribute a represents a computation and the variable $x.a$ represents the result of this computation on the term represented by the variable x.

Equation Systems: The considered equations are of the form $x = t$, where the left-hand-side is restricted to be a variable. A system Σ is a set of equations.

Properties and Program: A program is defined by a set of properties that rely on attributes. For instance, incrementing an integer is represented by the following property about the attribute inc:

$$(\forall x) \quad x.inc = (+\ x\ 1)$$

We will only consider properties which depend on the constructor appearing at the head of a term. For instance, the *length* attribute defining the length of a list verifies the two following properties:

$$(\forall x) \begin{cases} x = (cons\ \ldots) \Rightarrow \\ \quad x.length = (+\ 1\ x.2.length) \\ x = (nil) \Rightarrow \\ \quad x.length = 0 \end{cases}$$

To simplify notations, the universally quantified variable x is denoted by the special variable α. This yields the following specification, which is (a piece of) a program in equational semantics:

$$\begin{cases} cons \rightarrow \\ \quad \alpha.length = (+\ 1\ \alpha.2.length) \\ nil \rightarrow \\ \quad \alpha.length = 0 \end{cases}$$

The complete syntactic definition of a program in equational semantics is given below:

$\mathcal{N}$, Att, $Cons$ and $Prim$ are respectively the sets of integers, attributes, constructors and primitives.

$$\begin{array}{lcll} \mathcal{P} & ::= & (c \rightarrow p^*)^* & \\ p & ::= & x = t & \\ x & ::= & \alpha & \\ & | & x.a & a \in Att \\ & | & x.k & k \in \mathcal{N} \\ & | & x.L_i & i \in \mathcal{N} \\ t & ::= & x & \\ & | & (c\ t^*) & c \in Cons \\ & | & (\pi\ t^*) & \pi \in Prim \end{array}$$

Deduction Rule: A deduction rule φ is a function which takes a system and generates new equations according to it. The basic deduction rules are described below.

$$\varphi_{sub}(\Sigma) = \{x.k = t_k \mid x = (c\ t_1 \ldots t_n) \in \Sigma\}$$

$$\varphi_{subst}(\Sigma) = \{x = t[y := t'] \mid x = t \in \Sigma, y = t' \in \Sigma\}$$

$$\varphi_{prim}(\Sigma) = \{x = t' \mid x = t \in \Sigma, t \triangleright t'\}$$

$$\varphi_{prog}(\mathcal{P})(\Sigma) = \{p[x]\ ,\ \forall p \in A \mid x = (c \ldots) \in \Sigma, (c \rightarrow A) \in \mathcal{P}\}$$

$\triangleright$ is a rewriting rule over terms. The substitution $[x := t]$ replaces the *full* occurrences of variable x by t (i.e. x is not substituted in $x.a$, $x.k$ or $x.L_i$). The substitution $p[x]$ replaces each text occurrence of α in the property p by the variable x.

The deduction rule φ_{sub} is used to have access to subterms; for instance, if $x = (c\ t_1\ t_2)$ then the variable $x.1$ represents the sub-term t_1. The deduction rule φ_{subst} substitutes a variable by a term. The deduction rule φ_{prim} handles primitive computations; for instance $x = (+\ 1\ 1)$ gives $x = 2$. The deduction rule φ_{prog} depends on the program $\mathcal{P}$ and applies its *properties* (this notion is defined below). The program transformations described section 7 will be carried out by adding more deduction rules to this basic kernel.

Execution: While a program is defined by a set of properties, its execution[1] is a system of equations. This system is constructed by applying deduction rules to an initial system which represents the input data of the program.

The execution of a program involves the following definitions:

$$\psi ::= \varphi^*$$
$$\psi(\Sigma) = \Sigma \cup \bigcup_{\varphi \in \psi} \varphi(\Sigma)$$
$$\Sigma_\psi = \bigcup_{n \in \mathcal{N}} \psi^n(\Sigma)$$

The result of executing the program $\mathcal{P}$ with the initial system Σ and the set of deduction rules ψ is the system Σ_ψ. The basic kernel of deduction rules is:

$$\psi_{basic} = \{\varphi_{sub}, \varphi_{subst}, \varphi_{prim}, \varphi_{prog}(\mathcal{P})\}$$

The semantics of a program according to the set of deduction-rule ψ is the function which associates, to an input system Σ, the system Σ_ψ.

Such a semantics can be computed, and with a large amount of technical improvement[2], it can be computed efficiently. We have implemented a prototype, called EQS, which performs such computations, and more generally, manipulates and transforms programs in Equational Semantics.

4 Examples

This section intuitively presents how to encode various kind of algorithms with equational semantics. The example of executions come from the ones automatically computed by our implemented prototype EQS.

4.1 Attribute Grammars

As an example of encoding attribute grammars, we choose the example of reversing a list with an accumulator. The attribute *accu* is used to accumulate the elements and the final result is returned through the attribute *rev*. In the beginning, *accu* must be set to the empty list (nil). This program is specified in equational semantics as follows:

$$\begin{array}{l} cons \rightarrow \\ \quad \alpha.rev = \alpha.2.rev \\ \quad \alpha.2.accu = (cons\ \alpha.1\ \alpha.accu) \\ nil \rightarrow \\ \quad \alpha.rev = \alpha.accu \end{array}$$

Actually, a program which never use local variables looks like an attribute grammar. Now, let us consider the following initial system:

$$\begin{cases} x = (cons\ 1\ (cons\ 2\ (nil))) \\ x.accu = (nil) \end{cases}$$

[1] More precisely, it is the *trace* of an execution of the program.

[2] We do not describe them in this paper

The application of the basic kernel deduction rules yields the following execution. However, the entire execution is too large to be reported here, so we only report some new equations. For each of them, the deduction rule which produced it is noticed inside brackets.

$x.rev = x.2.rev$	$[prog]$
$x.2.accu = (cons\ x.1\ x.accu)$	$[prog]$
$x.1 = 1$	$[sub]$
$x.2 = (cons\ 2\ (nil))$	$[sub]$
$x.2.rev = x.2.2.rev$	$[prog]$
$x.2.2.accu = (cons\ x.2.1\ x.2.accu)$	$[prog]$
$x.2.1 = 2$	$[sub]$
$x.2.2 = (nil)$	$[sub]$
$x.2.2.rev = x.2.2.accu$	$[prog]$
$x.2.accu = (cons\ 1\ (nil))$	$[subst]$
$x.2.2.accu = (cons\ 2\ (cons\ 1\ (nil)))$	$[subst]$
$(\ldots)$	$[subst]$
$x.rev = (cons\ 2\ (cons\ 1\ (nil)))$	$[subst]$

To define a function that reverses a list, the constructor *reverse* is introduced. It stands for the call of this function while the attribute r is defined to catch the result of this call.

$$
\begin{array}{l}
reverse \rightarrow \\
\quad \alpha.r = \alpha.1.rev \\
\quad \alpha.1.accu = (nil)
\end{array}
$$

Now, given a list l and the equation $x = (reverse\ l)$, the reversed list is represented by the variable $x.r$.

4.2 Dynamic Trees

In the previous example, the recursion is driven by the constructors *cons* and *nil*. For functions like factorial, the recursion is only driven by a conditional expression. First, as like as in the previous example, a constructor *factorial* and an attribute r are used to represent a call to factorial. Second, for all variable x such that $x = (factorial\ t)$ the new local variable $x.L_1$ represents the result of the comparison $(<\ 1\ t)$ which drives the recursion. The computation is then continued on the constructor *true* or *false* through the attribute *fact*.

$$
\begin{array}{l}
factorial \rightarrow \\
\quad \alpha.r = \alpha.L_1.fact \\
\quad \alpha.L_1.n = \alpha.1 \\
\quad \alpha.L_1 = (<\ 1\ \alpha.1) \\
true \rightarrow \\
\quad \alpha.fact = (*\ \alpha.n\ \alpha.L_2.r) \\
\quad \alpha.L_2 = (factorial\ (-\ \alpha.n\ 1)) \\
false \rightarrow \\
\quad \alpha.fact = 1
\end{array}
$$

To illustrate how conditional recursions work with the local variable $\alpha.L_1$, we present now the execution from the initial system $\{x = (factorial\ 2)\}$:

$x.r = x.L_1.fact$	$[prog]$
$x.L_1.n = x.1$	$[prog]$
$x.L_1 = (<\ 1\ x.1)$	$[prog]$
$x.L_1.n = 2$	$[sub, subst]$
$x.L_1 = (<\ 1\ 2)$	$[sub, subst]$
$x.L_1 = (true)$	$[prim]$
$x.L_1.fact = (*\ x.L_1.n\ x.L_1.L_2.r)$	$[prog]$
$x.L_1.L_2 = \quad (factorial\ (-\ x.L_1.n\ 1))$	$[prog]$
$(\ldots)$	$[\ldots]$
$x.L_1.L_2.r = 1$	$[\ldots]$
$x.L_1.fact = (*\ 2\ 1)$	$[subst]$
$x.r = 2$	$[prim, subst]$

4.3 Composition

The example we present here does not belong to the scope of classical attribute grammars. More precisely, it can be encoded with two attribute grammars composed together, but the composition itself can not. Let n be a Peano integer, we build with the attribute *bin* a balanced binary tree of depth n with a first attribute grammar. Then a second one counts the leaves of this constructed tree with the attributes s and h, producing a new Peano integer m. Thus, we have $m = 2^n$. The composition is computed in the attribute r of the constructor *exp*.

The first attribute grammar is:

$$
\begin{array}{l}
succ \rightarrow \\
\quad \alpha.bin = (node\ \alpha.1.bin\ \alpha.1.bin) \\
zero \rightarrow \\
\quad \alpha.bin = (leaf)
\end{array}
$$

The second one is:

$$
\begin{array}{ll}
node \rightarrow & leaf \rightarrow \\
\quad \alpha.s = \alpha.1.s & \quad \alpha.s = (succ\ \alpha.h) \\
\quad \alpha.1.h = \alpha.2.s & \\
\quad \alpha.2.h = \alpha.h &
\end{array}
$$

The composition is defined by:

$$
\begin{array}{l}
exp \rightarrow \\
\quad \alpha.r = \alpha.L_3.s \\
\quad \alpha.L_3.h = (zero) \\
\quad \alpha.L_3 = \alpha.1.bin
\end{array}
$$

Thus, if n and m are Peano integers such that $m = 2^n$, then the initial system $\{x = (exp\ n)\}$ produces the equation $x.r = m$. Both "attribute grammars" are linear algorithms, but the size of the tree produced by the first one is an exponential of the size of the input tree. Thus the composition of these two attribute grammars produces an exponential algorithm. The composition itself can not be encoded with one attribute grammar.

Notice that the previous specification is transformed by our deforestation method into:

$succ \rightarrow$
  $\alpha.s' = \alpha.L_1.s'$
  $\alpha.L_1 = \alpha.1$
  $\alpha.L_1.h' = \alpha.1.s'$
  $\alpha.1.h' = \alpha.h'$
$zero \rightarrow$
  $\alpha.s' = (succ\ \alpha.h')$

$exp \rightarrow$
  $\alpha.r = \alpha.1.s'$
  $\alpha.1.h' = (zero)$

This result could not be encoded with classical attribute grammar since the visit which computes s' from h' is called twice with two different values for the attribute h' (look at the constructor *succ*). Here, the local variable $\alpha.L_1$ is identical to $\alpha.1$, but $\alpha.L_1.h'$ and $\alpha.1.h'$ represent different values. Notice that the classical descriptional composition composition failed in composing these two attribute grammars.

5 Evaluators

In this section, we show how to construct an *evaluator* for an equational semantics specification. An *evaluator* is a set of recursives *visits* that computes, for any tree t, the values of some attributes associated to t. By definition, the visit-call denoted by $[h_1 \ldots h_m \rightarrow s_1 \ldots s_n]\ (t)$ computes all the attributes s_i of t if and only if all the attributes h_j of t have been already computed. A visit is defined for each constructor by an ordered list of actions. An action could be either a call to a visit or the evaluation of an equation.

For instance, the following evaluator reverses a list:

$[accu \rightarrow rev]$
  $cons \rightarrow$
    eval $\alpha.2.accu = (cons\ \alpha.1\ \alpha.accu)$
    visit $[accu \rightarrow rev]\ (\alpha.2)$
    eval $\alpha.rev = \alpha.2.rev$
  $nil \rightarrow$
    eval $\alpha.rev = \alpha.accu$
$[\rightarrow r]$
  $reverse \rightarrow$
    eval $\alpha.1.accu = (nil)$
    visit $[accu \rightarrow rev]\ (\alpha.1)$
    eval $\alpha.r = \alpha.1.rev$

The construction of the evaluators is performed by a fix-point algorithm. The main idea is to compute step by step a pool of available visits. We first define the following operations:

- $Vcons(P, c)$: it finds all the visits that computes attributes on a constructor c. To make these visits, all the visits in pool P are assumed to be available on the sub-terms of c and on its local variables.
- $Vall(P)$: it computes (P', T) where P' is new pool of visits, and T is a table which associates each constructor to its visits. The result of *Vall* is such that for all constructor c, $T(c) = Vcons(P, c)$ and $P' = \bigcup T(c)$
- $Vverify(v, T)$: for the visit $v = [H \rightarrow S]$, it verifies that for each constructor c such that at least one attributes of S is defined on c, there exists a visit $[H' \rightarrow S]$ in $T(c)$ and $H' \subset H$. A visit that verifies this property is called "verified". If it is not the case, then the visit v may be undefined on a constructor and should be eliminated.

With such basic components, the fix-point algorithm is defined as follows:

$$\begin{array}{lcl} P_0 & = & \{[\rightarrow a] \mid a \in Att\} \\ P_{n+1} & = & F(P_n) \end{array}$$

where F is defined by:

$$F(P) = \{v \mid \ v \in P',\ Vverify(v, T),\ (T, P') = Vall(P)\}$$

When the fix-point is reached, the remaining visits correctly compute the values of the attributes. As an example, here is the first iteration to compute the visits to reverse a list:

$$P_0 = \{[\rightarrow rev],\ [\rightarrow r]\}$$

The computations of *Vcons* lead to:

$Vcons(P_0, cons) =$
  $[\rightarrow rev]$
    visit $[\rightarrow rev]\ (\alpha.2)$
    eval $\alpha.rev = \alpha.2.rev$
$Vcons(P_0, nil) =$
  $[accu \rightarrow rev]$
    eval $\alpha.rev = \alpha.accu$
$Vcons(P_0, reverse) =$
  $[\rightarrow r]$
    visit $[\rightarrow rev]\ (\alpha.1)$
    eval $\alpha.r = \alpha.1.rev$

Thus, after the first computation of *Vall* the visit $[\rightarrow rev]$ must be removed since it is not "verified" for *cons*. However, the new visit $[accu \rightarrow rev]$ is "verified" by *cons* and *nil*. Of course, since the fix point has not been reached, the evaluators found are not correct. Thus we have:

$$P_1 = \{[accu \rightarrow rev],\ [\rightarrow r]\}$$

Then, the second step produces the right evaluators and the fix point is reached.

Of course, this simple algorithm have to be improved to be efficient. The critical point is the computation of *Vcons* which seams to be highly exponential. However, a large amount of the constructed visits are identical (modulo permutation), and it is possible to compute them together. In practice, with our implemented prototype EQS, the complexity of the entire algorithm remains reasonable.

6 Safe Transformations

Intuitively, a transformation is correct if the transformed program produces the same results as the original one. In section 3 we define the execution of a program according to an input system Σ_i.

However, this execution is a system which contains many intermediate computations mixed with the expected result. Thus, we have to define which equations of the execution belong to the output system. For instance, consider the input system:

$$\Sigma_i \begin{cases} \alpha = (cons\ 1\ (cons\ 2\ (nil))) \\ \alpha.accu = (nil) \end{cases}$$

If we suppose that the interesting attributes are *rev* and *length*, the interesting output system is:

$$\Sigma_o \begin{cases} \alpha.rev = (cons\ 2\ (cons\ 1\ (nil))) \\ \alpha.length = 2 \end{cases}$$

Let R be a given set of the interesting attributes. The output system of an execution is the set of equations of the form: $\alpha.a = t$, where t is a term with no variable, and $a \in R$.

With such a definition, a program transformation is safe (or correct) if and only if, for all input system, the output systems of the original program and of the transformed one are equal. Thus, additional computations may exist and internal computations may change, but the final results have to remain identical.

7 Transformations

7.1 Partial Evaluation

Applying deduction rules and collecting the new equations produced stands for a kind of partial evaluation. For instance, suppose that we have the following program:

$$\begin{array}{l} test \rightarrow \\ \quad \alpha.r = (+\ \alpha.1\ \alpha.L_9.result) \\ \quad \alpha.L_9 = (factorial\ 3) \end{array}$$

Then from the initial system $x = (test\ x.1)$ it is possible to obtain the following equation:

$$x.r = (+\ x.1\ 6)$$

This equation can be generalized on the variable x since we only use the fact that $x = (test \ldots)$. Thus, a new property on the constructor *test* can be added, and finally we obtain the new program:

$$\begin{array}{l} test \rightarrow \\ \quad \alpha.r = (+\ \alpha.1\ \alpha.L_9.r) \\ \quad \alpha.r = (+\ \alpha.1\ 6) \\ \quad \alpha.L_9 = (factorial\ 3) \end{array}$$

Now, there exists two properties associated to the variable $\alpha.r$ for the constructor *test*. The two properties are correct according to section 6. The proof of such a correction comes from two ideas. Firstly, the property $\alpha.r = (+\ \alpha.1\ 6)$ only comes from the original program. Secondly, adding this new equation does not modify the execution of the original program, but some equations will be deduced with less applications of ψ.

Actually, partial evaluation is the real kernel of the other transformations we define in this paper.

7.2 Reduction

In a program, there are often several properties for a unique variable. In the previous example *test*, there are two properties for the variable $\alpha.r$ (the original and the generated one). In this case, it is interesting to eliminate the first one which involves too much other equations to be computed. To get benefit from a program transformation, many properties must be eliminated.

It is not always possible to eliminate a property. More precisely, an elimination will be safe if and only if it never produces undefined variables during an execution.

In most cases, many solutions exist and we have to choose an efficient one. Reaching optimality is a very difficult problem. However there are simple and intuitive heuristics (which were implemented in our prototype) to obtain reasonable results. In the previous example *test* the reduction leads to:

$$\begin{array}{l} test \rightarrow \\ \quad \alpha.r = (+\ \alpha.1\ 6) \end{array}$$

7.3 Specialization

With functional notations, this transformation is defined as follows: suppose that f is a function of n parameters $x_1 \ldots x_n$, the specialization of f when the parameter x_1 is equal to the constant K is the new function h defined by:

$$(h\ x_2 \ldots x_n) = (f\ K\ x_2 \ldots x_n)$$

This is the first step of the transformation, where a new definition is introduced. The second step of the transformation consists in recognizing where f can be replaced by h. More precisely, it consists in the following term-replacement everywhere in the program:

$$(f\ K\ t_1 \ldots t_{n-1}) \Rightarrow (h\ t_1 \ldots t_{n-1})$$

These two steps can be translated into equational semantics in a systematic way. For the first step, a new attribute is introduced for the computation of h and new attributes are introduced for its parameters. Additional properties are automatically generated in order to link the new attributes to the old ones. For the second step, a new deduction rule is added to the basic kernel, which simply translates the old attributes into the new ones whenever it is possible.

For instance, consider the example of mapping the function factorial to a list. Let *mapf* be the new attribute that

computes this specialization of *map*. Since the attribute *map* is defined on the constructors *cons* and *nil*, the properties verified by *mapf* must be reported on these two constructors. The additional program corresponding to the **first step** is then:

$$\begin{array}{l}(\forall c \in \{cons, nil\})\\ c \rightarrow\\ \quad \alpha.mapf = \alpha.L_m.map\\ \quad \alpha.L_m.f = (fact\_ho)\\ \quad \alpha.L_m = \alpha\end{array}$$

The local variable $\alpha.L_m$ must be fresh for each additional program, that is, not already used. The **second step** automatically produces the new following deduction rule:

$$\begin{array}{l}\varphi_{spe}(\Sigma) = \{x.map = x.mapf \mid\\ \quad x.f = (fact\_ho) \in \Sigma\}\end{array}$$

At this point, the specialization of the attribute *map* in the special case where f is equal to $(fact\_ho)$ is done and safe. The interesting point is now that partial evaluation and reduction will get benefit from the introduction of these new attributes, properties and deduction rules. For instance, let us describe how simplifications occur for the constructors *cons*. We only report some equations produced by partial evaluation and related to this specialization:

$x = (cons\ x.1\ x.2)$	
$x.mapf = x.L_m.map$	$[prog]$
$x.L_m = (cons\ x.1\ x.2)$	$[prog, subst]$
$x.L_m.f = (fact\_ho)$	$[prog]$
$x.L_m.map =$	$[prog]$
$\quad (cons\ x.L_m.L_4.call\ x.L_m.2.map)$	
$x.L_m.L_4.arg = x.1$	$[prog, \ldots]$
$x.L_m.L_4 = (fact\_ho)$	$[\ldots]$
$x.L_m.L_4.call = x.L_m.L_4.L_3.r$	$[prog, \ldots]$
$x.L_m.L_4.L_3 = (factorial\ x.1)$	$[\ldots]$
$x.L_m.2.f = (fact\_ho)$	$[prog, \ldots]$
$x.L_m.2.map = x.L_m.2.mapf$	$[spe]$

The two last blocks show how the constant $fact\_ho$ is propagated, and how the *map* attribute is transformed into *mapf*. After generalization and reduction, the following properties are generated for the constructors *cons* and *nil*:

$$\begin{array}{l}cons \rightarrow\\ \quad \alpha.mapf = (cons\ \alpha.L_{10}.r\ \alpha.2.mapf)\\ \quad \alpha.L_{10} = (factorial\ x.1)\\ nil \rightarrow\\ \quad \alpha.mapf = (nil)\end{array}$$

The new local variable $\alpha.L_{10}$ has been introduced to rename (safely) the local variable $\alpha.L_m.L_4.L_3$.

7.4 Deforestation

In functional terms, this transformation occurs when functions are composed. Basically, the problem involves two functions: f with parameters $x_1 \ldots x_n$ and g with parameters $y_1 \ldots y_m$. If f and g are composed, for instance through the first parameter of f, a new function h is defined:

$$(h\ y_1 \ldots y_m\ x_2 \ldots x_n) = (f\ (g\ y_1 \ldots y_m)\ x_2 \ldots x_n)$$

This is the first step of the transformation, where a new definition is introduced. The second step of the transformation consists in recognizing when f is composed with g and then in replacing such a composition by a call to h. More precisely, it consists in the following term-replacement everywhere in the program:

$$(f\ (g\ s_1 \ldots s_m)\ t_1 \ldots t_{n-1}) \Rightarrow (h\ s_1 \ldots s_m\ t_1 \ldots t_{n-1})$$

From an equational semantics point of view, this transformation is performed in two steps as like as for specialization. In the first step, we introduce a new attribute for h and new attributes for its $(m + n - 1)$ parameters. New properties (a new program) are also automatically generated to link the new attributes to the old ones. For the second step, a new deduction rule is added to the execution kernel, which simply translates the old attributes into the new ones.

As a preliminary remark, a composition is detected in equational semantics when the variable $x.b$ is used while the equation or property $x = y.a$ holds. In such a case, the composed attributes are a and b.

However, there are actually two kinds of deforestation. In the first kind, named *upward deforestation*, the attribute a is the result of a computation. In the second kind, named *downward deforestation*, the attribute a is a parameter of a computation.

We choose an example which involves these two kinds of deforestation: the reversion of the reversion of a list. For this purpose, the following program is specified:

$$\begin{array}{l}foo \rightarrow\\ \quad \alpha.r = \alpha.L_{11}.rev\\ \quad \alpha.L_{11}.accu = (nil)\\ \quad \alpha.L_{11} = \alpha.1.rev\\ \quad \alpha.1.accu = (nil)\end{array}$$

We present now the two steps of the two kinds of the deforestation transformations.

Upward Deforestation: The composed attributes are *rev* and *rev*. We denote by r_2 the attribute for the result of the composition, and by a_1 and a_2 the two attributes needed for the two accumulators of *rev* and *rev*. The **first step** defining these new attributes corresponds to the following program:

$$
\begin{array}{l}
[\text{for c} = \mathit{cons} \text{ and c} = \mathit{nil}] \\
c \rightarrow \\
\quad \alpha.r_2 = \alpha.L_p.\mathit{rev} \\
\quad \alpha.L_p.\mathit{accu} = \alpha.a_1 \\
\quad \alpha.L_p = \alpha.L_q.\mathit{rev} \\
\quad \alpha.L_q.\mathit{accu} = \alpha.a_2 \\
\quad \alpha.L_q = \alpha
\end{array}
$$

where L_p and L_q are fresh. This requirement is important to safely add these properties to the original program. The **second step** produces automatically the new following deduction rule which detects where r_2 could replace a composition:

$$
\begin{array}{l}
\varphi_{\mathit{defo\_up}}(\Sigma) = \{ \\
\quad x.\mathit{rev} = x.L_m.r_2 \\
\quad x.L_m.a_1 = x.\mathit{accu} \\
\quad x.L_m.a_2 = y.\mathit{accu} \\
\quad x.L_m = y \\
\quad \mid \quad x = y.\mathit{rev} \in \Sigma \\
\}
\end{array}
$$

where L_m is a fresh variable for each application of the deduction rule. The deforestation definition is done and safe. Now, partial evaluation and reduction will perform the expected simplifications. For instance, for the constructor *foo*, the following equations are deduced:

$x = (\mathit{foo}\ x.1)$	
$x.r = x.L_{11}.\mathit{rev}$	$[\mathit{prog}]$
$x.L_{11} = x.2.\mathit{rev}$	$[\mathit{prog}]$
$x.L_{11}.\mathit{accu} = (\mathit{nil})$	$[\mathit{prog}]$
$x.2.\mathit{accu} = (\mathit{nil})$	$[\mathit{prog}]$
$x.L_{11}.\mathit{rev} = x.L_{11}.L_m.r_2$	$[\mathit{defo\_up}]$
$x.L_{11}.L_m.a_1 = x.L_{11}.\mathit{accu}$	$[\mathit{defo\_up}]$
$x.L_{11}.L_m.a_2 = x.2.\mathit{accu}$	$[\mathit{defo\_up}]$
$x.L_{11}.L_m = x.2$	$[\mathit{defo\_up}]$

After generalization and reduction, the following properties are obtained:

$$
\begin{array}{l}
\mathit{foo} \rightarrow \\
\quad \alpha.r = \alpha.1.r_2 \\
\quad \alpha.1.a_1 = (\mathit{nil}) \\
\quad \alpha.1.a_2 = (\mathit{nil})
\end{array}
$$

In the same way, for the constructors *cons* and *nil* we obtain:

$$
\begin{array}{l}
\mathit{cons} \rightarrow \\
\quad \alpha.r_2 = \alpha.2.r_2 \\
\quad \alpha.2.a_1 = \alpha.a_1 \\
\quad \alpha.2.a_2 = (\mathit{cons}\ \alpha.1\ \alpha.a_2) \\
\mathit{nil} \rightarrow \\
\quad \alpha.r_2 = \alpha.L_{12}.\mathit{rev} \\
\quad \alpha.L_{12}.\mathit{accu} = \alpha.a_1 \\
\quad \alpha.L_{12} = \alpha.a_2
\end{array}
$$

Downward Deforestation: After the deforestation above, the second kind of deforestation appears on the constructor *nil*. The composed attributes are a_2 and *rev*, where a_2 is a parameter-attribute instead of a result-attribute. Such a deforestation through accumulative parameters is known to be difficult [4], but is naturally handled in equational semantics.

Let r_3 be the new attribute introduced for the result of the composition, and a_3 the new attribute introduced for the related accumulative parameter. The **first step** still consists in the automatic generation of the program which defines these attributes: everywhere the attribute a_2 is computed, the attribute r_3 must be equal to *rev* on a_2 with *accu* being equal to a_3. In the example, a_2 is computed on $\alpha.2$ for the constructor *cons*, and on $\alpha.1$ for the constructor *foo*. So the first step corresponds to the following additional program:

$$
\begin{array}{l}
\mathit{cons} \rightarrow \\
\quad \alpha.2.r_3 = \alpha.L_m.\mathit{rev} \\
\quad \alpha.L_m.\mathit{accu} = \alpha.2.a_3 \\
\quad \alpha.L_m = \alpha.2.a_2 \\
\mathit{foo} \rightarrow \\
\quad \alpha.1.r_3 = \alpha.L_p.\mathit{rev} \\
\quad \alpha.L_p.\mathit{accu} = \alpha.1.a_3 \\
\quad \alpha.L_p = \alpha.1.a_2
\end{array}
$$

where L_m and L_p are fresh local variables. The **second step** of the transformation is the automatic generation of the following deduction rule which detects where r_3 could replace a composition:

$$
\begin{array}{l}
\varphi_{\mathit{defo\_down}}(\Sigma) = \{ \\
\quad x.\mathit{rev} = y.r_3 \\
\quad y.a_3 = x.\mathit{accu} \\
\quad \mid x = y.a_2 \\
\}
\end{array}
$$

Multiple applications of this deduction rule on the same variable y is not allowed. This technical point is not explained here since it is too specific to this kind of deforestation. After partial evaluation and reduction, the following program is obtained:

$$
\begin{array}{l}
\mathit{foo} \rightarrow \\
\quad \alpha.r = \alpha.1.r_2 \\
\quad \alpha.1.r_3 = \alpha.1.a_3 \\
\quad \alpha.1.a_1 = (\mathit{nil}) \\
\mathit{cons} \rightarrow \\
\quad \alpha.r_2 = \alpha.2.r_2 \\
\quad \alpha.2.r_3 = \alpha.r_3 \\
\quad \alpha.a_3 = (\mathit{cons}\ \alpha.1\ \alpha.2.a_3) \\
\quad \alpha.2.a_1 = \alpha.a_1 \\
\mathit{nil} \rightarrow \\
\quad \alpha.r_2 = \alpha.r_3 \\
\quad \alpha.a_3 = \alpha.a_1
\end{array}
$$

Notice that the deforestation really succeed since only one list is constructed. Moreover, the result is a copy of the first list, as it is expected to: in fact r_2 is always equal to

r_3, and a_3 appends a_1 (initialized to *nil*) to the end of the list.

7.5 Elimination of Identity

Consider the properties about r_2 and a_3 on the constructor *nil*. They are both equalities. The elimination of identity try to prove whether these equalities are verified for all constructors or not. The transformation is performed in two steps. First, the equality is automatically proved or refuted by induction. Second, for the proved equalities, a new deduction rule is automatically defined.

In the example below, the induction proof on the constructor *cons* consists in assuming the properties on variable $\alpha.2$, and prove them on variable α. The proof will be automatically performed by partial evaluation. Assuming the induction hypothesis on $\alpha.2$ corresponds to the following system:

$$\begin{aligned} x &= (cons\ x.1\ x.2) \\ x.2.r_2 &= x.2.r_3 \\ x.2.a_3 &= x.2.a_1 \end{aligned}$$

The partial evaluation produces the following execution:

$$\begin{array}{ll} x.r_2 = x.2.r_2 & [prog] \\ x.2.r_3 = x.r_3 & [prog] \\ x.a_3 = (cons\ x.1\ x.2.a_3) & [prog] \\ x.2.a_1 = x.a_1 & [prog] \\ \hline x.r_2 = x.2.r_3 & [subst] \\ x.a_3 = (cons\ x.1\ x.2.a_1) & [subst] \\ \hline x.r_2 = x.r_3 & [subst] \\ x.a_3 = (cons\ x.1\ x.a_1) & [subst] \end{array}$$

The inductive hypothesis is verified for the equality $\alpha.r_2 = \alpha.r_3$, but the other equality is not verified. So, for the second step of the transformation there is only one new deduction rule defined:

$$\varphi_{id}(\Sigma) = \{x.r_2 = x.r_3 \mid x\ appears\ in\ \Sigma\}$$

After partial evaluation, we obtain the following program:

$$\begin{aligned} &foo \rightarrow \\ &\quad \alpha.r = \alpha.1.a_3 \\ &\quad \alpha.a_1 = (nil) \\ &cons \rightarrow \\ &\quad \alpha.a_3 = (cons\ \alpha.1\ \alpha.2.a_3) \\ &\quad \alpha.2.a_1 = \alpha.a_1 \\ &nil \rightarrow \\ &\quad \alpha.a_3 = \alpha.a_1 \end{aligned}$$

Now, we have succeed in proving automatically that reverse composed with itself is equal to the function copy, which duplicates its input list.

```
let flat x h = match x with
  node a b -> flat a (flat b h)
| leaf n -> cons n h
let flatten x = flat x nil

let f x = reverse (flatten x)
```

Figure 1: flatten and reverse

```
let f =
  fun t_27 -> (((fpfun_1 t_27) nil))

let fpfun_1 =
  fun t_42 -> (
    fun t_43 -> (match t_42 with
      | node t_44 t_45 ->
  ((fpfun_1 t_45) ((fp-
fun_1 t_44) t_43))
      | leaf t_51 -> ((cons t_51) t_43)
      ))
```

Figure 2: flatten and reverse deforested

```
let append x y = match x with
  cons a b -> cons a (append b y)
| nil -> y
let f x y z = (append (append x y) z)
```

Figure 3: Wrong composition with append

```
let fpfun_2 =
  fun t_38 -> (
    fun t_39 -> (match t_38 with
      | nil -> t_39
      | cons t_41 t_42 ->
  ((cons t_41) ((fpfun_2 t_42) t_39))
      ))

let f =
  fun t_16 -> (
    fun t_17 -> (
      fun t_15 -> (
  ((fpfun_2 t_16) ((append t_17) t_15))
      )))
```

Figure 4: Better composition with append

```
let revho x = match x with
  cons a b ->
    let k = (revho b) in
    (fun h -> k (cons a h))
| nil -> (fun h -> h)

let reverse x = ((revho x) nil)
```

Figure 5: reverse with higher order

```
let fpfun_1 =
  fun t_11 -> (
    fun t_12 -> (match t_11 with
      | nil -> t_12
      | cons t_14 t_15 ->
  ((fpfun_1 t_15) ((cons t_14) t_12))
    ))

let reverse =
  fun t_3 -> (((fpfun_1 t_3) nil))
```

Figure 6: reverse with h.o. deforested

8 Additional Results

In section 4 we presented few examples of equational programs. Of course, we would never claim that programming directly with equational semantics is easy. Actually, it is more interesting to translate existing programs into equational programs. We have found two methods, one for translating functional programs to equational ones, and one for the backward translation. Technical details and correction of these translations are too long to be exposed in this paper. But we would want to briefly present interesting examples to illustrate the power of our transformations. All these examples come from the implementation of our system.

Reversed flatten: the function `f` given in figure 1 takes a binary tree, flattens its leaves, and then reverses the obtained list. After four steps of deforestation, the program in figure 2 is obtained. One can observe that it is a variant of the function `flat` where the tree is flattened in the reversed direction. So, our analysis and deforestation methods are able to completely modify the control flow of a recursive function.

Inefficient composition: figure 3 presents the function `append` which appends two lists, and the function `f` which appends three lists. Actually, the expression `(append (append x y) z)` should be translated into `(append x (append y z))` to avoid one duplication of each list `x` and `y`. Deforestation performs the transformation automatically as shown in figure 4.

Removing continuations: As a last example, we transform the reverse function written with a continuation, given in figure 5. The data deforested is the continuation. The result in figure 6 is equal to the standard function `rev` with accumulator. This result shows the power of dealing with a system which does not include function calls. In equational semantics, functional values are encoded like other values, and thus, they could be treated in a same way. Here , the elimination of the continuation is performed by the *standard* deforestation for equational programs.

9 Conclusion

This work comes from a large comparative study of various existing methods to perform deforestation and partial evaluation in various programming paradigm. Historically, we compared [5, 4, 3] the deforestation of attribute grammars [7, 8, 14], the Wadler deforestation [18, 15, 9] in functional programming, many works about folds [6, 11] and hylomorphisms [12, 17]. In each of these formalisms, there were many interesting ideas. But they were sometimes restricted to one particular class of algorithms but sometimes more powerful than another method on the same class. However, attribute grammars seems to provide a kind of declarative notation able to gather all of them in an homogeneous way.

Actually, we think that the key of our approach is to define a program only by the set of the properties it verifies. Functions, procedures, data types, control statements of real programming languages are here considered as syntactic sugar to define properties as equations. In this context, Equational Semantics is a minimal but powerful framework to manipulate these properties and translate them back into a more efficient program.

References

[1] Loic Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Equational Semantics In SAS '99, Static Analysis, volume 1694 of LNCS. Springer Verlag, 1999.

[2] Loic Correnson. Sémantique Equationnelle PhD Thesis, Ecole Polytechnique, Avril 2000.

[3] Loic Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Symbolic composition. Technical Report 3348, INRIA, January 1998.

[4] Etienne Duris. *Contribution aux relations entre les grammaires attribuées et la programmation fonctionnelle.* PhD thesis, Université d'Orléans, 1998.

[5] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Structure-directed genericity in functional programming and attribute grammars. Rapport de Recherche 3105, INRIA, February 1997.

[6] Leonidas Fegaras, Tim Sheard, and Tong Zhou. Improving programs which recurse over multiple inductive structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'94)*, pages 21–32, Orlando, Florida, June 1994.

[7] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 157–170, Montréal, June 1984. ACM press. Published as *ACM SIGPLAN Notices*, 19(6).

[8] Robert Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25:355–423, 1988.

[9] G. W. Hamilton. Higher order deforestation. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume 1140 of *Lect. Notes in Comp. Sci.*, pages 122–136, Aachen, September 1996. Springer-Verlag.

[10] A. Kühnemann. *Berechnungsstärken von Teilklassen primitiv-rekursiver Programmschemata.* PhD thesis, Technical University of Dresden, 1997. Shaker Verlag, Aachen.

[11] John Launchbury and Tim Sheard. Warm fusion: Deriving build-cata's from recursive definitions. In *Conf. on Func. Prog. Languages and Computer Architecture*, pages 314–323, La Jolla, CA, USA, 1995. ACM Press.

[12] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conf. on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lect. Notes in Comp. Sci.*, pages 124–144, Cambridge, September 1991. Springer-Verlag.

[13] Didier Parigot, Gilles Roussel, Martin Jourdan, and Etienne Duris. Dynamic Attribute Grammars. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume 1140 of *Lect. Notes in Comp. Sci.*, pages 122–136, Aachen, September 1996. Springer-Verlag.

[14] Gilles Roussel. *Algorithmes de base pour la modularité et la réutilisabilité des grammaires attribuées.* PhD thesis, Département d'Informatique, Université de Paris 6, March 1994.

[15] Morten Heine Sørensen. A grammar-based data-flow analysis to stop deforestation. In S. Tison, editor, *Colloquium on Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*, pages 335–351. Springer-Verlag, 1994.

[16] S. Doaitse Swierstra and Harald H. Vogt. Higher Order Attribute Grammars. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lect. Notes in Comp. Sci.*, pages 256–296, New York–Heidelberg–Berlin, June 1991. Springer-Verlag.

[17] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conf. on Func. Prog. Languages and Computer Architecture*, pages 306–313, La Jolla, CA, USA, 1995. ACM Press.

[18] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. In Harald Ganzinger, editor, *European Symposium on Programming (ESOP '88)*, volume 300 of *Lect. Notes in Comp. Sci.*, pages 344–358, Nancy, March 1988. Springer-Verlag.

Two-dimensional Approximation Coverage

Jörg Harm
Universität Rostock, Fachbereich Informatik
18051 Rostock, Germany
Phone: +49 381 498 34 32, Fax: +49 381 498 34 26
E-mail: jh@informatik.uni-rostock.de
AND
Ralf Lämmel
CWI, P.O. Box 94079
1090 GB Amsterdam, The Netherlands
Phone: +31 20 592 40 90, Fax: +31 20 592 41 99
E-mail: ralf@cwi.nl

Keywords: testing, coverage, attribute grammars, declarative programming

Edited by: Marjan Mernik and Didier Parigot
Received: August 11, 2000 **Revised:** September 3, 2000 **Accepted:** September 11, 2000

The notion of approximation coverage is developed. It is applicable to first-order declarative programs (e.g., logic programs, constructive algebraic specifications, and attribute grammars) in two dimensions in a natural way. For an attribute grammar, for example, there is a syntactic dimension corresponding to the underlying context-free grammar, and there is also a semantic dimension corresponding to the attributes, conditions, and computations. The coverage notion is based on an abstract interpretation scheme. The paper also develops a generator algorithm for test sets achieving coverage. The coverage notion facilitates testing of declarative programs, and assessment of test sets. The test set generator facilitates automated testing. A language definition based on an attribute grammar specification is used as an illustrative example.

1 Introduction

Testing declarative programs Testing is useful to gain confidence about the correctness of a specification or a program. The paper provides concepts for testing first-order declarative programs. The presentation is tuned towards attribute grammars (AGs), but the concepts are also applicable to other formalisms and languages, e.g., constructive algebraic specifications, and logic programs. Let us motivate the necessity of testing in terms of a major application area for AGs, that is language definition, prototyping and implementation. Developing, extending and tuning real-world AG specifications are non-trivial tasks. Thus, testing and verification should form a standard activity in the corresponding software engineering processes. Verification has been addressed in the literature to a certain extent (cf. [12, 21]). By contrast, testing is poorly developed for AGs. Standard testing technology (cf. [23, 3]) is not applicable to a large extent.

Two-dimensional approximation coverage An important observation is that the declarative programs, which we want to test, usually exhibit two dimensions. For an attribute grammar, for example, there is a syntactic dimension corresponding to the underlying context-free grammar, and there is a semantic dimension corresponding to the attributes. As another example, one dimension in a logic program corresponds to proof tree skeletons, and another one corresponds to the parameters of the literals in proof trees. The coverage notion we are going to develop applies to the two dimensions in a uniform way in separation. Coverage in both dimensions can be lifted in a sensible way to a two-dimensional coverage. Approximation coverage is defined in terms of the structure underlying the dimensions, say context-free grammars and attribute type definitions for AGs. For the syntactic dimension of AGs, for example, approximation coverage separates the various occurrences of nonterminals, and it is also sensitive regarding the recursion involved in the underlying grammar. We will also demonstrate that negative test cases can be accomplished in this setting. Approximation coverage is useful to assess test sets or existing test suites (cf. [5]). Other applications relying on test set generation will be pointed out below.

Research context For attribute grammars, essentially only the syntactic dimension has been explored in the sense of rule coverage for context-free grammars (cf. [26]). Approximation coverage goes very much beyond rule coverage—even if it is restricted to the syntactic dimension. Coverage of the semantic dimension, and the combination of both dimensions has not been investigated at all in the literature. We will argue that two-dimensional approximation coverage covers the aspects of a given declara-

tive program in a more exhaustive manner than such simple notions as rule coverage. Previous research addressed the generation of correct derivation trees (cf. [19, 16, 15]), i.e., trees which can be decorated in accordance to the computations and the conditions. Randomized test set generation has been suggested by several authors. There is no guarantee for randomized test sets to cover all aspects of the corresponding program. For logic programs, Jack (cf. [17]) introduced a sophisticated coverage notion based on anti-unification. We will comment on its relation to our two-dimensional approximation coverage in the conclusions.

Test set generation Our coverage notion is effective in the sense that test set generation is feasible. A corresponding algorithm is developed in the paper. Generated test sets are useful in automated testing. Consider, for example, a language definition specified by an AG. By generating and applying test cases for the language definition, the developer can check if his intuitions regarding the language are met, and if the behaviour of the executable language definition is as expected. This process can be conceived as a kind of white-box debugging of language definitions. Test sets generated from an AG are also useful for a kind of black-box testing of language implementations where the AG is considered as the reference. In this case, the actual implementation does not need to be based on AGs at all. Even if it is (partially) based on compiler compiler (say AG) technology, the actual specification used for the implementation might not be accessible, or it might deviate from the reference specification for practical reasons. In any case, if a reference and an implementation need to be compared, at least testing is desirable if verification is not considered as an option. Here, generated test sets are indeed useful since they obviously automate testing in such a comparison scenario. The coverage notion is configurable to put the focus on a certain aspect, and thereby, the generation of test sets can be controlled accordingly. Test set generation is not straightforward because full coverage might be infeasible, and it is in general not decidable if a current coverage can be improved. This may cause the generation not to terminate. We will discuss techniques to recover feasibility of coverage such as more precise attribute types.

Structure of the paper In Section 2, our testing approach is motivated. An AG describing syntax and (part of the) static semantics of a language with blocks and jumps serves as an example. In Section 3, the new notion of coverage is developed. In Section 4, approximation coverage is instantiated for AGs. A non-trivial test set for the motivating AG satisfying approximation coverage in one sensible configuration is given. Opportunities for the configuration of the coverage notion are explained. In Section 5, test set generation is discussed. The goal is to generate test sets achieving full coverage. In this context, termination and search space problems need to be addressed. In Section 6, the paper is concluded.

2 Motivation

In this section, we want to discuss a few aspects of testing AGs along the scenario that an AG is supposed to provide the reference specification for some implementation of an acceptor A.

We will argue that the simple coverage of all productions in the context-free grammar is not sufficient. The other dimension of AGs, that is the attributes with conditions and computations also need to be taken into account. Actually, rule coverage is not even sufficient in the syntactic dimension. We also will comment on negative test cases, especially in the semantic dimension.

2.1 Preliminaries

We assume basic knowledge of context-free grammar theory and attribute grammars as covered by surveys like [28, 1, 25, 20]. For convenience, some elementary terminology is provided in the sequel.

A context-free grammar G is a quadruple $\langle N, T, s, P\rangle$ as usual, i.e., N and T are the disjoint finite sets of nonterminals resp. terminals. $s \in N$ is called start symbol. P is a finite set of productions or (context-free) rules with $P \subset N \times (N \cup T)^*$. We resort to the common notation $l \to r$ for a production $\langle l, r\rangle \in P$ with $l \in N$ and $r \in (N \cup T)^*$. For simplicity, we assume reduced and terminated context-free grammars in the sequel.

An attribute grammar AG is a quadruple $\langle G, A, CM, CN\rangle$, where G is the underlying context-free grammar, A associates each $x \in N \cup T$ with finite sets of synthesized attributes $A_s(x)$ and inherited attributes $A_i(x)$, CM and CN associate each production p of G with finite sets of computations $CM(p)$ and conditions $CN(p)$. We assume well-formed, non-cyclic attribute grammars in normal form. Given a production $p = x_0 \to x_1 \cdots x_m \in P$, with $x_0 \in N$, $x_1, \ldots, x_m \in N \cup T$, a computation c from $CM(p)$ is of the form $r_0.a_0 := f_c(r_1.a_1, \ldots, r_k.a_k)$ where $0 \leq r_j \leq m$, and x_{r_j} carries an attribute a_j for $j = 0, \ldots, k$; similar for conditions.

2.2 Rule coverage

Let us motivate rule coverage by the following test scenario. We want to test an acceptor A which is supposed to accept some language $L(G)$ generated by a context-free grammar G. Later we generalise this scenario from context-free grammars to AGs. We take some finite set $TS \subseteq L(G)$ and check if A accepts each $w \in TS$. We want to gain confidence that the language accepted by A actually is $L(G)$. Thus, we have to ensure that TS experiences to a certain degree all aspects of $L(G)$. Actually, TS should cover G to some extent. The bare minimum of coverage is to require that every production of G is applied in the derivation of some $w \in TS$.

Figure 1 shows an excerpt of a context-free grammar for a Pascal-like programming language. In the derivation of

[prog]	Prog	→	Block .
[block]	Block	→	Decls **begin** Stms **end**
[nodecl]	Decls	→	ϵ
[decls]	Decls	→	Decl Decls
[decl]	Decl	→	**label** id ;
[onestm]	Stms	→	Stm
[stms]	Stms	→	Stm ; Stms
[skip]	Stm	→	ϵ
[goto]	Stm	→	**goto** id
[ldef]	Stm	→	id : Stm
[if]	Stm	→	**if** Exp **then** Stm
[localb]	Stm	→	Block
[true]	Exp	→	**true**
...			

Figure 1: Productions for a Pascal-like language

the program

```
label a;
begin
 a : goto a; begin if true then skip end
end.
```

all productions of the context-free grammar of Figure 1 are used.

If A is assumed to implement an AG rather than just a context-free grammar, the above scenario needs to be refined. The conditions and partial computations of an AG usually enforce that the language generated by the AG is only a subset of $L(G)$ where G is the underlying context-free grammar. Thus, we should preferably consider only semantically correct programs in the test set TS. For any decent AG, rule coverage should remain feasible. Moreover, if the aim is just to test the context-free parsing aspect of A w.r.t. the reference grammar G, we can even consider possibly semantically incorrect test programs. From a practical perspective, we only had to be able to separate syntactic and semantic errors while applying A to TS.

2.3 Beyond rule coverage

Rule coverage is by far too simple. More complex criteria than simple rule coverage are sensible to enforce certain kinds of combinations of productions. Focusing, for example, on declaration parts of blocks in the sample language from Figure 1, the following aspects were not reflected in the sample program above:

1. The declaration part of a program block may consist not only of one, but also of zero or more than one declaration.
2. A local block statement may have a non-empty declaration part.

The first problem suggests that we need a coverage notion which treats recursion in a sensible manner. The second problem is an indication that a more context-dependent notion than just rule coverage is useful. Actually, both scenarios are somewhat context-dependent. In the first case, we are concerned with program blocks, and in the second case with local block statements. Thus, two different occurrences of the nonterminal Block or the corresponding declaration part resp. are considered.

Approximation coverage suggests a layered definition of coverage for the nonterminals. The central idea is to take a configurable number of recursive unfoldings into account. As a first attempt, for a nonterminal n with $n \rightarrow w$ as one of its alternatives, we can say that this alternative is covered by a test set TS if all grammar symbols in w are covered. Contrast that with rule coverage which just enforces that the rule $n \rightarrow w$ is used once for the derivation of some program in TS. We have to explain what it means to cover grammar symbols. Terminals like **begin** are trivially covered if the alternative is used. For terminals, which actually correspond to a terminal class, e.g., id, coverage might be achieved by using one or two different representatives for the corresponding occurrence in w. A nonterminal is covered if all alternatives for this nonterminal are covered. Of course, this definition of coverage has to be refined to cope with recursively defined nonterminals in a sensible way. We say that a nonterminal n is covered at a level λ. In particular, n is covered at level 0 if it is used in some derivation; n is covered at level λ if all its alternatives are covered, at level $\lambda - 1$ as far as recursive occurrences of n are concerned. In this way, the λ restricts the number of recursive unfoldings.

Two recursive unfoldings are already quite useful. This is somewhat similar to testing loops in an imperative program, that is tests are usually required for zero, one, and more than one iterations. Indeed, for the first problem (see 1. above), which we used to illustrate the weakness of rule coverage, a test set with programs with zero, one and more than one declarations in the program block was required. Note that the kind of context-dependency of coverage needed to address both problems is achieved by defining coverage for all rules in separation, namely by decrementing λ in a context, that is for a particular occurrence of a nonterminal. In the formalisation, we will consider a different λ_n for each nonterminal n.

2.4 The semantic dimension

So far we considered syntactic aspects of an AG, as represented by its underlying context-free grammar or language. It is also conceivable to consider aspects which are formulated in the semantic dimension. They could be concerned, for example, with (attributes for) symbol tables, label tabels, and types. Therefore, we take a look at the attribute part of an attribute grammar. For our Pascal-like programming language, the attribute part is concerned with the static semantics of the language. In the attribute grammar fragment in Figure 2–Figure 3, we focus on scope rules for labels. We use lists of identifiers, that is the domain

$\mathcal{ID\_LIST}$, for the representation of sets of labels.

Type definitions:

$\mathcal{ID} = \{a, ..., z\}^+$
$\mathcal{ID\_LIST} = [] + [\mathcal{ID}|\mathcal{ID\_LIST}]$

Attributes:

$A_i(\text{Block}) = A_i(\text{Stms}) = A_i(\text{Stm}) = \{\text{TL}\}$
$A_s(\text{Stms}) = A_s(\text{Stm}) = \{\text{DL}, \text{LTL}\}$
$A_s(\text{id}) = \{\text{Name}\}$
$A_s(\text{Decl}) = \{\text{LN}\}$
$A_s(\text{Decls}) = \{\text{L}\}$

Attribute types:

Name : $\mathcal{ID}$	name of the identifier
LN : $\mathcal{ID}$	name of the declared label
L : $\mathcal{ID\_LIST}$	list of declared labels
TL : $\mathcal{ID\_LIST}$	target labels reachable from inside the block, statement list, or statement
DL : $\mathcal{ID\_LIST}$	labels with a defining occurrence inside the statements of a block
LTL : $\mathcal{ID\_LIST}$	labels with defining occurrence inside the statement list or statement which are reachable by goto statements on the same statement nesting level

Figure 2: Attributes for checking jumps

The coverage notion for context-free grammars can be lifted to attribute grammars if we assume that the base grammar can be covered with a subset of the language defined by the attribute grammar. That does not yet imply that we have a coverage notion for the semantic dimension. It just means that coverage in the syntactic dimension is regarded in a way that the semantic dimension is respected. We can go one step further by taking the structure of the attribute values into account. The idea of a layered coverage notion spelled out for context-free grammars above can be used for attribute type definitions in a similar way. Thereby, we get a coverage notion for possibly recursive domain equations.

Let us illustrate this idea with the domains from Figure 2. Suppose that the basic domain $\mathcal{ID}$ is covered by two different representatives. We can cover the domain $\mathcal{ID\_LIST}$ in the case of recursion level 2 by lists of the length zero, one, and greater than one where at each position at least two different $\mathcal{ID}$ values occur. Two sensible test sets are the following:

- {[], [a|[]], [a|[b|[]]], [b|[a|[]]]}
- {[], [a|[]], [a|[a|[]]], [b|[b|[]]]}

Now, we can say a subset TS of the language defined by an attribute grammar AG covers an attribute of a production of AG, if the values associated with this attribute in the derivation trees of the elements of TS cover the domain of the attribute.

While the coverage notion for context-free grammars forces the application of grammar rules in meaningful syntactic contexts, the domain coverage forces meaningful semantic contexts. Applied to the domain equation of Figure 2 and to the nonterminal Stm, for example, the domain coverage criterion enforces the use of the various alternatives of Stm in different contexts covering its attributes TL, DL, and LTL.

```
[prog]   Prog  →  Block .
 1.TL    :=  []
[block]  Block →  Decls begin Stms end
 3.TL    :=  (0.TL \ 1.L) ∪ 3.LTL
 3.DL    =   1.L
[nodecl] Decls →  ϵ
 0.L     :=  []
[decls]  Decls →  Decl Decls
 0.L     :=  [1.LN|2.L]
 1.LN    ∉   2.L
[decl]   Decl  →  label id ;
 0.LN    :=  2.Name
[onestm] Stms  →  Stm
 1.TL    :=  0.TL
 0.DL    :=  1.DL
 0.LTL   :=  1.LTL
[stms]   Stms  →  Stm ; Stms
 1.TL    :=  0.TL
 3.TL    :=  0.TL
 0.DL    :=  1.DL ∪ 3.DL
 0.LTL   :=  1.LTL ∪ 3.LTL
 1.DL ∩ 3.DL = []
[skip]   Stm   →  ϵ
 0.DL    :=  []
 0.LTL   :=  []
[goto]   Stm   →  goto id
 0.DL    :=  []
 0.LTL   :=  []
 2.Name  ∈   0.TL
[ldef]   Stm   →  id : Stm
 3.TL    :=  0.TL
 0.DL    :=  [1.Name|3.DL]
 0.LTL   :=  [1.Name|3.LTL]
 1.Name  ∉   3.DL
[if]     Stm   →  if ... then Stm
 4.TL    :=  0.TL ∪ 4.LTL
 0.DL    :=  4.DL
 0.LTL   :=  []
[localb] Stm   →  Block
 1.TL    :=  0.TL
 0.DL    :=  []
 0.LTL   :=  []
[true]   Exp   →  true
...
```

Figure 3: AG for checking jumps

For both the syntactic and the semantic dimension, full coverage often cannot be achieved for a given AG specification because of two related problems:

- Full coverage for the context-free grammar may not be achieved because the conditions on the attributes rule out some syntactic combinations.
- Full coverage may not be achieved for some attributes

because in every decorated derivation tree the attribute values are of a special form.

These problems are somewhat similar to the well-known problem of unexecutable paths in testing imperative programs. An example of infeasibility of coverage in the syntactic dimension is that a program block cannot just consist of a goto-statement because the target label could not be defined in this program block. An example of infeasibility of an attribute's type is the 0.TL attribute of production [ldef] in Figure 3 which is always a non-empty list because at least the label defined by the alternative itself will be in the list due to remote dependencies. Later we will explain how to relax coverage accordingly.

2.5 Negative test cases

For the discussed scenario of testing an acceptor A w.r.t. an AG, negative test cases also have to be taken into consideration. Otherwise, the incorrectness of A might not be realized. A might accept a richer language than the intended language. In general, negative test cases are quite useful in testing language processors, to see if incorrect programs are rejected, and proper error messages are produced. As for positive test cases, we would like to reason about coverage of negative test cases, and generation of negative test cases is very useful for automated testing.

For language processors implementing AGs, there are two kinds of negative test cases to be considered due to the two dimensions involved. One kind of negative test case should cause syntactic errors. The other should violate context conditions. In the syntactic dimension, we can consider an adapted context-free grammar which is meant to be incorrect w.r.t. the original one. Assuming a test set generator for positive test cases, the very same generator could be applied to the adapted grammar to generate negative test cases. It has to be defined how the incorrect grammar is obtained. One option is to use ideas from mutation testing [24, 13]. We do not discuss negative test cases for the syntactic dimension in more detail. We will explain how to accomplish negative test cases in the semantic dimension in a systematic way.

Let us assume that the computations in a given AG do not fail, although is not difficult to lift this restriction. Then, a negative test case can be conceived as a derivation tree where some associated conditions are not satisfied. Suppose we have a test set generator for positive test cases. The same generator can be used for the generation of negative test cases, if it is applied to an modified AG with negated conditions. Violations of context-conditions are indeed enforced by the negated conditions.

To avoid confusion of different violations, each negative test case should be generated from a modified AG with just one negated condition. There are different useful coverage criteria conceivable. The minimum is, of course, that the production p with the negated condition is covered at all. Approximation coverage is useful to enforce different contexts for the nonterminal on the left-hand side of p. There is one requirement, which is specific to negative test cases, that is in a given derivation tree, the affected production p should probably covered exactly once. This requirement induces a unique error location.

Instead of negating conditions, we might also remove conditions, and check afterwards that a generated derivation tree is not correct w.r.t. the original grammar.

There is another conceivable approach to generate a test set with negative test cases. It is based on the simple idea to ignore the semantic dimension in the generation phase. The algorithm for test set generation proposed in Section 5 generates in an intuitive sense smallest programs. Thus, by ignoring the semantic dimension we get smallest programs achieving syntactic coverage. Afterwards the test set could be filtered to contain only such programs which are not accepted by the attribute grammar, that is only the programs violating some context conditions remain in the final test set. The approach based on negating conditions deals explicitly with semantic coverage.

3 Approximation coverage

In the sequel a general notion of approximation coverage will be introduced. In Section 4, we will derive a coverage notion for AGs. The notion of approximation coverage is applicable to other first-order declarative programs as well. To abstract from the particular declarative language and dimension at hand, approximation coverage is defined for equational systems of a certain form. The idea is that these equational systems capture the essence of the common dimensions involved in the declarative programs to be tested. Focusing on AGs, for example, equational systems are meant to abstract from context-free grammars and attribute type definitions. For logic programs, equational systems abstract from proof tree skeletons and functor signatures.

First, we define equational systems. Afterwards, an abstraction scheme for equational systems to model coverage is developed. Finally, the scheme is instantiated to obtain the desired notion of approximation coverage based on a finite unfolding technique.

3.1 Equational systems

An equational system S over variables $X_1, \ldots, X_n$ and constants $C_1, \ldots, C_m$ consists of n equations $X_1 \equiv t_1, \ldots, X_n \equiv t_n$, where the t_i are terms over $\mathbf{1}$, $X_1, \ldots, X_n$, $C_1, \ldots, C_m$ composed with $\times$ and $+$. A set-theoretic interpretation is assumed, i.e., $C_1, \ldots, C_m$ are predefined sets, $\mathbf{1}$ is a dedicated singleton set, $\times$ and $+$ correspond to Cartesian product and disjoint union respectively. We assume a well-formedness property for S in the sense of the termination property for context-free grammars. The solutions for the X_i are denoted by $[\![X_i]\!]$.

It should be clear that these equational systems can be used to study context-free grammars and attribute types

in a unified setting. For convenience, we list a few supporting arguments. Attribute types can directly be modelled with equational systems if they are defined as products, and sums over some basic datatypes. If we assume instead (first-order and non-parameterized) algebraic datatypes, such a definition can easily be transformed into an equivalent equational system by encoding constructors as sums of products. Context-free languages can be defined in various ways, e.g., by using context-free grammars, or by resorting to (possibly extended) BNF notation, or to an algebraic interpretation. The relation between these formalisms or notations is well understood. The kind of equational systems we propose is semantically best conceived as BNF. The terminals of a context-free grammar correspond to the C_i, whereas the nonterminals are the variables X_i. A union in a BNF (the set of alternatives for a nonterminal in a context-free grammar) is represented with $+$, whereas $\times$ models concatenation. $\mathbf{1}$ models ϵ. For (the dimensions involved in) other first-order declarative programs, the above notion of equational systems can be adopted likewise.

3.2 Coverage by abstraction

The $[\![X_i]\!]$ for an equational system S are potentially infinite sets. Thus, an exhaustive test exploring all elements of $[\![X_i]\!]$ is impractical. We need to perform abstraction to derive a feasible coverage notion. Abstract interpretation concepts are used in the sequel (cf. [14, 11, 27, 18]).

The basic idea is that the X_i are associated with abstract domains $\overline{X_i}$ modelling coverage for X_i. Actually, we also need such abstract domains for the C_i and $\mathbf{1}$. Abstract domains are supposed to obey the following structure:

- They are posets of the form $\langle \overline{z}, \bot, \top, \leq \rangle$.
- $\overline{z}$ is a (not necessarily finite) set.
- $\leq$ is a partial order on $\overline{z}$.
- There is a smallest element $\bot \in \overline{z}$.
- There is a greatest element $\top \in \overline{z}$.
- For all $a, b \in \overline{z}$, the supremum $a \sqcup b$ is defined.
- There are only finite chains.

$\bot$ models no coverage whereas $\top$ models full coverage. Another requirement for the abstraction scheme will be added later. In the same way as the X_i etc. are associated with abstract domains, the $v \in [\![X_i]\!]$ are associated with abstract values $\overline{v}$. Furthermore, usually abstract counterparts $\overline{\times}$ and $\overline{+}$ for the concrete domain constructors $\times$ and $+$ are needed as well, since the abstract domains $\overline{X_i}$ are most likely to be defined according to the structure of the equations in the underlying equational system S.

The finite chain property required above ensures that a finite test is sufficient to reach $\top$. In test set generation, indeed, we go along chains. To require finite abstract domains would be unnecessarily restrictive. For the natural numbers $\mathcal{N}_0$, for example, one useful (infinite) abstract domain can be described as follows. Coverage of natural numbers is achieved by k different natural numbers. For $k = 2$, for example, we take the flat domain with $\bot < x < \top$ for all $x \in \mathcal{N}_0$.

At this point, we can define some central terms for testing. The definitions given below are based on the following intuition. The coverage of a test set is computed by taking the supremum of the corresponding abstract values. In principle, minimal test sets should be enforced. To reflect the incremental generation of a test set based on a current coverage, sequences of test values as opposed to sets might be considered. Each element in the sequence should improve coverage. This does not imply that the sequence forms a minimal test set. On the other hand, a subsequent minimalisation is always possible.

Definition 1 (full coverage)
Given a finite (test) set $TS \subseteq [\![X_i]\!]$, *TS fully covers* X_i, *if* $\bigsqcup_{v \in TS} \overline{v} = \top$.

Definition 2 (minimal test set)
$TS \subseteq [\![X_i]\!]$ *is a minimal test set (w.r.t.* X_i*) if there is no* $TS' \subset TS$ *such that* $\bigsqcup_{v \in TS} \overline{v} = \bigsqcup_{v \in TS'} \overline{v}$.

Definition 3 (increasing coverage)
A sequence $\langle v_1, \ldots, v_k \rangle$ *with* $v_1, \ldots, v_k \in [\![X_i]\!]$ *strictly increases coverage (w.r.t.* X_i*), if* $\bigsqcup_{i=1}^{l} \overline{v_i} < \bigsqcup_{i=1}^{l+1} \overline{v_i}$ *for* $l = 1, \ldots, k-1$.

We want to mention another desirable property of the abtraction scheme. Given a set z and the corresponding abstract domain $\overline{z}$, abstract values arising from concrete values in z should exactly correspond to the smallest non-bottom values in $\overline{z}$. Actually, for every smallest non-bottom value in $\overline{z}$, there should be (at least) one associated concrete value in z. For brevity, this requirement is not formalised. Without this property, the $\overline{z}$ could contain "unreachable" values, i.e., abstract values which cannot be obtained by taking the abstraction of a concrete value, or the supremum of abstractions of concrete values.

3.3 Coverage of constants

We assume that $\overline{\mathbf{1}}$ denotes the special poset $\langle \{\bot, \top\}, \bot, \top, \{\bot \leq \top\} \rangle$.

For the $\overline{C_i}$, there is no generic way to define them. Their definition is usually specific to the equational system at hand, and to the particular test scenario. However, there are some common methods to define the $\overline{C_i}$. Often $\overline{C_i} = \overline{\mathbf{1}}$ is appropriate, e.g., for terminals. For finite sets C_i, $\overline{C_i} = \overline{\mathcal{P}}([\![C_i]\!])$, is practical, e.g., for the Booleans. Here, $\overline{\mathcal{P}}(z)$ denotes the poset for the powerset of the set z, i.e., $\langle \mathcal{P}(z), \emptyset, z, \subseteq \rangle$. Methods to separate several equivalence classes are also sensible in many occassions. For integers, for example, it is sometimes useful to consider negative and positive numbers, and 0.

3.4 Finite unfolding

We will present one particular definition of the abstract domains $\overline{C_i}$. Other definitions are conceivable, but a discussion of pros and cons is beyond the scope of the paper. The preferred approach is based on abstract domains which approximate the concrete domains in the sense of a finite unfolding technique for the equational systems. This explains the name for the coverage notion: *approximation coverage*.

Before we present the ultimate definition, we want to illustrate two extremes. One extreme is the following:

- $\overline{X_i} = \overline{\mathbf{1}}$ for $i = 1, \ldots, n$
- $\overline{v} = \top$ for $v \in [\![X_i]\!]$, for $i = 1, \ldots, n$

This means that coverage can be achieved with just one concrete value. Interestingly, if we adopt that definition for a context-free grammar in the way that we require such a coverage for all nonterminals, we get a notion of coverage which is even weaker than rule coverage. It would be enforced that all nonterminals are used at least once in a derivation. There is another extreme, which is not feasible due to infinity of the $[\![X_i]\!]$:

- $\overline{X_i} = \overline{\mathcal{P}}([\![X_i]\!])$ for $i = 1, \ldots, n$
- $\overline{v} = \{v\}$ for $v \in [\![X_i]\!]$, for $i = 1, \ldots, n$

There are several ways to derive the $\overline{X_i}$ by observing somehow the structure of the definition for X_i in the underlying equational system S. Figure 4 presents the choice for approximation coverage. The definition uses a special parameter η to keep track of remaining unfolding steps for the various X_i. The initial number of unfolding steps (η_0) is configured by some natural numbers λ_{X_i}—one parameter for each X_i. As already pointed out in Section 2, $\lambda_{X_i} = 2$ is usually sufficient. We might indeed want to supply different λ_{X_i} for the different X_i. This would be useful to put the focus on certain X_i while relaxing the coverage for other X_j.

There are two ways how the X_j are handled in the definition of $\mathcal{AT}_\eta^S$. Either $\eta(X_j) = 0$, then unfolding is stopped, or $\eta(X_j) > 0$, then the equation for X_j in S is traversed with an updated η so that the counter for X_j is decremented. Decrementing is encoded as updating the function η at X_j as denoted by $\eta[\eta(X_j) - 1/X_j]$. For $\overline{+}$ resp. $\overline{\times}$ we assume an interpretation as (normal) Cartesian product resp. strict product on posets with $\perp$. Recall that the difference between a non-strict and a strict product on posets with $\perp$ is the following. In the normal Cartesian sense, $\perp$ of the product corresponds to the tuple $\langle \perp, \ldots, \perp \rangle$. In the strict case, all tuples containing at least one $\perp$ are unified. This choice is sensible, because the tuples induced by $\overline{+}$ model coverage for the several alternatives in a sum. Some of the alternatives might be covered, others not. On the other hand, the tuples induced by $\overline{\times}$ model coverage for tuples. Non-strict products as abstract domains do not make sense here because each concrete tuple will immediately cover all components to a certain extent (more than $\perp$).

Note that the defined abstract domains have the finite chain property, since they are constructed from such domains (the $\overline{C_i}$, $\overline{\mathbf{1}}$) just in terms of product operators which preserve the finite chain property. The recursion involved in the definition of $\mathcal{AT}_\eta^S$ is harmless, since the decrementation of η ensures that the number of unfoldings of equations is limited to a finite value.

In Figure 4, the definition of the abstract domains is accompanied with the abstraction function for concrete values. For $\mathcal{AV}_\eta^S(v : X_j)$ with $\eta(X_j) = 0$, coverage is trivially satisfied ($\top$). So every value is fine. For $\eta(X_j) > 0$, values experiencing more structure of X_j are enforced. Given a tuple, coverage is a tuple, too. Coverage is computed component-wisely. Given a value $\langle v, j \rangle$ from a sum domain arising from the j-th alternative, coverage for the j-th component for the abstract value is equal to the coverage of v. For all the other alternatives, no coverage is achieved ($\perp$). This treatment of sums enforces that all alternatives of a sum have to be experienced in order to achieve coverage. Finally, note also that the given definition of abstract domains and values satisfy the requirement regarding smallest non-bottom values.

4 Attribute grammar coverage

We are going to instantiate the notion of approximation coverage for the syntactic and the semantic dimension of attribute grammars. The interesting part is the actual combination of the dimensions resulting in a two-dimensional coverage. At the end of the section, it is discussed how coverage can be configured. It should be conceivable that a similar instantiation is feasible for other declarative languages, e.g., logic programs and constructive algebraic specifications.

4.1 Preliminaries

$\mathcal{T}^G$ denotes the set of context-free derivation trees. $\mathcal{T}^{AG}$ denotes the set of derivation trees with associated computations from CN and conditions from CN according to AG. It is common to rely on the Dewey-notation for attribute references within the associated computations and conditions. $\mathcal{DT}^{AG}$ denotes the set of decorated derivation trees. Attribute evaluation means to map a derivation tree $t \in \mathcal{T}^{AG}$ to a decorated derivation tree $dt \in \mathcal{DT}^{AG}$ in accordance to the computations and conditions in t. In other terms, t induces an equational system on attributes further constrained by the conditions in t. The solution of the system (if there is any) provides the decoration in dt.

Derivation trees for context-free and attribute grammars are usually rooted by the start symbol. From a practical perspective of testing, we can indeed not assume that we can directly test a certain nonterminal n by using derivations starting from n. Language processors, for example, expect a complete program. Conceptually, decoration or attribute evaluation is usually only considered for complete derivation trees. However, as far as coverage is concerned, we

$\overline{X_i} = \mathcal{AT}^S_{\eta_0}(X_i)$ for $i = 1, \ldots, n$, where
$\eta_0 : \{X_1, \ldots, X_n\} \to \mathcal{N}_0$ with $\eta_0(X_j) = \lambda_{X_j} \in \mathcal{N}_0$ for $j = 1, \ldots, n$,

$$
\begin{array}{lcl}
\mathcal{AT}^S_\eta(\mathbf{1}) & = & \overline{\mathbf{1}} \\
\mathcal{AT}^S_\eta(C_j) & = & \overline{C_j} \\
\mathcal{AT}^S_\eta(X_j) & = & \overline{\mathbf{1}} \text{ if } \eta(X_j) = 0 \\
\mathcal{AT}^S_\eta(X_j) & = & \mathcal{AT}^S_{\eta[\eta(X_j)-1/X_j]}(e) \text{ if } \eta(X_j) > 0 \text{ and } X_j \equiv e \in S \\
\mathcal{AT}^S_\eta(e_1 \times \cdots \times e_k) & = & \mathcal{AT}^S_\eta(e_1) \overline{\times} \cdots \overline{\times} \mathcal{AT}^S_\eta(e_k) \\
\mathcal{AT}^S_\eta(e_1 + \cdots + e_k) & = & \mathcal{AT}^S_\eta(e_1) \overline{+} \cdots \overline{+} \mathcal{AT}^S_\eta(e_k)
\end{array}
$$

$\overline{v} = \mathcal{AV}^S_{\eta_0}(v : X_i)$, for $v \in [\![X_i]\!], i = 1, \ldots, n$, where

$$
\begin{array}{lcl}
\mathcal{AV}^S_\eta(v : \mathbf{1}) & = & \top \\
\mathcal{AV}^S_\eta(v : C_j) & = & \overline{v} \\
\mathcal{AV}^S_\eta(v : X_j) & = & \top \text{ if } \eta(X_j) = 0 \\
\mathcal{AV}^S_\eta(v : X_j) & = & \mathcal{AV}^S_{\eta[\eta(X_j)-1/X_j]}(v : e) \text{ if } \eta(X_j) > 0 \text{ and } X_j \equiv e \in S \\
\mathcal{AV}^S_\eta(\langle v_1, \ldots, v_k \rangle : e_1 \times \cdots \times e_k) & = & \langle \mathcal{AV}^S_\eta(v_1 : e_1), \ldots, \mathcal{AV}^S_\eta(v_k : e_k) \rangle \\
\mathcal{AV}^S_\eta(\langle v, j \rangle : e_1 + \cdots + e_k) & = & \langle c_1, \ldots, c_k \rangle \text{ where} \\
 & & j \in \{1, \ldots, k\}, \\
 & & c_j = \mathcal{AV}^S_\eta(v : e_j), c_l = \bot \text{ for } l = 1, \ldots, k, l \neq j
\end{array}
$$

Figure 4: Abstract domains $\mathcal{AT}^S_\eta$ and values $\mathcal{AV}^S_\eta$

would like to reason about particular nonterminals. Therefore, we introduce a notation to access subtrees rooted by a certain nonterminal.

Let be $t \in \mathcal{T}^G$. $\mathcal{ASUB}_n(t)$ denotes the set of all subtrees rooted by n in t. Let us also consider maximum subtrees $\mathcal{MSUB}_n(t) \subseteq \mathcal{ASUB}_n(t)$. A tree $t' \in \mathcal{ASUB}_n(t)$ is a maximum subtree of t, if all its ancestor nodes in t are different from n. We can also derive subtrees of derivation trees with associated computations and conditions ($\mathcal{T}^{AG}$), and decorated derivation trees ($\mathcal{DT}^{AG}$).

4.2 Two dimensions

The notion of equational systems and approximation coverage is applicable to the two dimensions of an attribute grammar. In the dimension induced by the context-free grammar, the variables correspond to the nonterminals, whereas in the dimension induced by the attribute type definitions, the variables correspond to the names of the type definitions. Now, we will join the abstract domains for both dimensions in a sensible way. We should point out that this join is actually independent of the details of approximation coverage. It only depends on the abstraction scheme. Note also that for other first-order declarative programs, a two-dimensional coverage notion can be derived in a similar way. However, this genericity is not explored in the paper.

Let AG be an attribute grammar with the context-free base grammar G and attribute types defined by the system D of domain equations. In the sequel, we identify G with the equational system corresponding to its productions. $\overline{n}^G$ is used to describe the syntactic coverage of the nonterminal n of G. Therefore, we call it syntactic abstract domain. Similarly, $\overline{\tau}^D$ for a type τ defined in D is used to describe the semantic coverage of attributes of type τ. Thus, we call it semantic abstract domain. Now, we build the combined abstract domain for a nonterminal in three steps:

1. Construct a semantic abstract domain $\overline{p}^D$ for each production p of G.
2. Combine the semantic abstract domains for the productions defining a nonterminal n to a semantic abstract domain $\overline{n}^D$.
3. Combine semantic and syntactic abstract domains for each nonterminal n of G to an abstract domain $\overline{n}^{AG}$.

Let p be a production of G and $r_1.a_1 : \tau_1, \ldots, r_k.a_k : \tau_k$ the attributes associated with p in the attribute grammar AG together with their types. It is clear that the type p^D of the decorations of p in a decorated derivation tree is $p^D = \tau_1^D \times \cdots \times \tau_k^D$. Thus, the semantic abstract domain $\overline{p}^D$ for p is defined as $\overline{p}^D = \overline{\tau_1}^D \,\overline{\times}\, \cdots \,\overline{\times}\, \overline{\tau_k}^D$. The definition means that each attribute of p has to be covered individually. Furthermore, the choice of $\overline{\times}$ reflects that the application of a production in a derivation covers each attribute to a certain extent. If $p_1, \ldots, p_m$ are the productions of G defining the nonterminal n, then we can encode decorations of the various productions for n as a sum. This is modelled by the concrete domain is $n^D = p_1^D + \cdots + p_m^D$. The corresponding semantic abstract domain $\overline{n}^D$ for n is defined as $\overline{n}^D = \overline{p_1}^D \,\overline{+}\, \cdots \,\overline{+}\, \overline{p_m}^D$. The two-level approach to the definition of $\overline{n}^D$ enforces that semantic coverage of non-terminals separates the different occurrences of n in the various rules. A relaxed definition of $\overline{n}^D$ could also be conceived. Finally, we define the combined abstract domain $\overline{n}^{AG} = \overline{n}^G \,\overline{\times}\, \overline{n}^D$.

It remains to define the abstraction function for attribute grammars. Here the consideration of subtrees turns out to be essential. Given a decorated derivation tree $dt \in$

$\mathcal{DT}^{AG}$, and a nonterminal n, the corresponding abstract value w.r.t. n is denoted by $\overline{dt}^n$. It is a pair of values for abstract syntactic coverage and abstract semantic coverage defined as follows:

$$\begin{aligned} \overline{dt}^n &= \langle syn, sem \rangle \\ syn &= \bigsqcup_{dt' \in \mathcal{MSUB}_n(dt)} \overline{\pi_G(dt')} \\ sem &= \bigsqcup_{dt' \in \mathcal{ASUB}_n(dt)} \overline{\pi_D(dt')} \end{aligned}$$

where $\pi_G(dt')$ denotes the derivation subtree obtained from dt' by removing its decoration, and $\pi_D(dt') \in n^D$ denotes the decoration of the top-level production of dt'. This definition of abstraction for decorated trees means that syntactic coverage for n is derived by taking the fundamental approximation coverage of all maximum subtrees rooted by n. To consider other than maximum subtrees rooted by n would be in conflict with the desired treatment of recursion. By contrast, semantic coverage is derived from all subtrees rooted by n because we want to observe the decoration of all nodes with nonterminal n and their successor nodes.

There is a fundamental problem with two-dimensional coverage. In many cases, full coverage according to the above definition is not feasible. In the syntactic dimension, coverage has sometimes to be relaxed due to semantic constraints. Dually, through syntactic dependencies, full attribute coverage is sometimes not feasible, i.e., in all derivation trees some attributes always take values of a special form. Opportunities to relax the coverage notion are discussed later.

4.3 A test set sample

In Figure 5, a represantative test set for the example AG in Figure 3 is shown. The test set achieves greatest possible coverage according to the following criteria. In the structural dimension, $\eta_0(n) = 1$ is assumed for nonterminals n. In the semantic dimension, $\eta_0(\tau) = 2$ is assumed for attribute types τ. Thereby, all attributes of type $\mathcal{ID\_LIST}$ are enforced to appear in derivation trees in which they get the empty list, a singleton list, and a list with at least two elements as values, if possible. The programs were actually generated by the algorithm described in the next section.

For example, test programs are generated where nonlocal labels are reachable from a block, i.e., the inherited attribute TL of the nonterminal Block has to be nonempty. Note that full coverage is not feasible because there are attributes of type $\mathcal{ID\_LIST}$ which cannot be associated with the empty list, e.g., the attribute TL of the left-hand side of rule [ldef].

4.4 Configuration

Approximation coverage can be configured in various ways. This is convenient to enforce a desired precision of the approximation of the concrete domains. Configuration might be essential to recover feasibility of coverage as pointed out above.

```
begin skip end.
begin if true then skip end.
begin skip; skip end.
begin begin skip end end.
begin if true then skip; skip end.
label a; begin a : skip end.
begin begin skip end; skip end.
label a; begin a : goto a end.
begin begin skip; skip end end.
label a; begin a : skip; skip end.
label a; begin skip; a : skip end.
label a; begin if true then a : skip end.
label a; begin a : if true then skip end.
label a; begin goto a; a : skip end.
label a; begin a : skip; goto a end.
label a; begin a : begin skip end end.
begin label a; begin a : skip end end.
label b; label a; begin b : a : skip end.
label a; begin a : begin goto a end end.
label b; label a; begin a : b : skip end.
label b; label a; begin a : b : goto a end.
begin label a; begin a : skip end; skip end.
label b; label a; begin b : a : skip; skip end.
label b; label a; begin skip; b : a : skip end.
label b; label a; begin if true then b : a : skip end.
label b; label a; begin a : b : if true then skip end.
label b; label a; begin a : b : begin skip end end.
```

Figure 5: A test set for the AG from Section 2

First of all, we can use different unfolding parameters $\eta_0(X_i) = \lambda_{X_i}$ for the various variables in an equational system. In the two-dimensional setting of attribute grammars, thereby both the nonterminals and the attribute types can be controlled. There are further opportunities to configure the coverage. The way how the semantic domain $\overline{p}^D$ for the production in an attribute grammar is constructed, specific coverage can be enforced for the various attributes. One useful extreme is to assume the trivial domain $\overline{1}$ rather than $\overline{\tau_i}^D$ for a certain attribute $r_i.a_i : \tau_i$ when constructing the product defining $\overline{p}^D$. Thereby, we express that the testing scenario is not concerned with $r_i.a_i : \tau_i$.

To recover feasibility of coverage, further techniques are needed. In some way or another, full coverage needs to be relaxed by excluding certain subsets of the full coverage set. In the semantic dimension, we can give more precise types to the attributes, that is subtypes of the attribute types. In the syntactic dimension, we can resort to a refactored context-free base grammar which explicitly reflects the permitted structures.

5 Test set generation

In this section, we give an algorithm for the generation of test sets providing coverage. The presentation is tuned towards AGs, that is we are concerned with (decorated) derivation trees, and words generated by an AG. In principle, the technology is also applicable to other first-order declarative programs. We presume that the coverage criterion is fulfilled by rather a larger set of small test cases than a smaller set of large test cases. Since small test cases

tend to test the aspects of the described language more separately, such test cases are more suitable for debugging purposes. This assumption is illustrated by the generated test set in Figure 5.

First, we set up some terminology. Then, we present the heart of the test set generator, that is an algorithm for completion of partial derivation trees. The algorithm derives—in some sense—smallest completions, and only correct trees regarding the AG at hand. Afterwards, we describe how coverage of the abstract domains for an AG can be achieved. Finally, the two concerns of rule completion vs. abstract domain coverage are intertwined, that is we describe how the search algorithm can be guided relying on the coverage notion. Otherwise, test set generation would be too inefficient. Also, we comment on termination problems resulting from the potential infeasibility of full coverage.

5.1 Preliminaries

We adapt the common algebraic interpretation of context-free grammars to be able to cope with partial derivation trees. Test set generation relies on the stepwise completion of partial derivation trees.

Given a context-free grammar $G = \langle N, T, s, P\rangle$, a signature Σ is derived as follows. $N \cup T$ provide the sorts of Σ. We need to include T because we want to represent terminals in the decorated derivation trees, since they may carry (synthesized) attributes. The productions in P are considered as function symbols in Σ, i.e., given a $p = x_0 \to x_1 \cdots x_m \in P$, there is a function symbol $p : x_1 \times \cdots \times x_m \to x_0 \in \Sigma$. Since we included terminals in Σ, we need to declare a term representation for terminals. We assume a special constant symbol $leaf_x : \to x$ for each $x \in T$ in Σ. Without further effort, the set of complete derivation trees $\mathcal{T}^G$ can be regarded as the set $T_s(\Sigma)$ of terms of sort s.

The representation of partial derivation trees is not so straightforward. One option to cope with "holes" in derivation trees is to consider terms with variables. Then, a partial derivation tree derived from a nonterminal n would be a term from $T_n(\Sigma, X)$ over some N-sorted family of variables X. This option is not quite usable. Variables are rather non-intuitive representations of holes in a partial derivation tree, since there is no natural interpretation for multiple occurrences of one variable in a context-free derivation tree. We resort to another option, that is we assume a constant symbol $leaf_x : \to x$ for each $x \in N$ in Σ. Recall that for terminals, $leaf_x$ denotes a leaf in the sense of a complete derivation tree. By contrast, for nonterminals, $leaf_x$ denotes a hole in the derivation tree. Extension of partial trees is meant to replace such holes.

Derivation trees might be obtained essentially in two ways. First, a rule can be represented as a partial derivation tree. Second, partial derivation trees can be extended by replacing holes by further derivation trees. We will present these two fundamental concepts. For convenience, we also assume that $leaf_x$ for each $x \in N$ is an elementary partial derivation tree.

Given a production $p = x_0 \to x_1 \cdots x_m \in P$, the corresponding partial derivation tree is represented by the term $p(leaf_{x_1}, \ldots, leaf_{x_m})$. We assume a notation for selecting or addressing subterms in the spirit of VDL (cf. [29]). Given a term $t \in T_n(\Sigma)$ and a selector sequence $q^\star \in \mathcal{N}_0^\star$, selection of the subterm in t addressed by $q^\star$ is denoted by $\mathcal{S}(t, q^\star)$. In a similar way, the extension of t by $t' \in T_{n'}(\Sigma)$ at the leaf addressed by $q^\star$ is denoted by $\mathcal{E}(t, q^\star, t')$. Selection and extension are defined in Figure 6.

It is easy to cope with derivation trees with associated computations and conditions. Using the standard Dewey notation as for selection, attributes are made unique.

5.2 Completion of derivation trees

We want to develop that part of the test set generator which completes a given incomplete derivation tree aiming at—in some sense—smallest completions. Later we also explain how to take the current coverage into account to guide the underlying search algorithm.

Let AG be an attribute grammar with $G = \langle N, T, s, P\rangle$ as its context-free base grammar. Starting from elementary derivation trees of the form $t_p = p(leaf_{x_1}, \ldots, leaf_{x_m}) \in T_{x_0}(\Sigma)$ for a production $p = x_0 \to x_1, \cdots, x_m \in P$, we can construct all derivation trees using the extension function $\mathcal{E}$. The derivation tree t corresponding to a derivation $s \overset{p_1}{\Rightarrow}_G w_1 \overset{p_2}{\Rightarrow}_G w_2 \overset{p_3}{\Rightarrow}_G \cdots \overset{p_n}{\Rightarrow}_G w_n$ can be constructed in top-down manner as follows:

$$\mathcal{E}(\mathcal{E}(\cdots \mathcal{E}(\mathcal{E}(t_{p_1}, q_1^\star, t_{p_2}), q_2^\star, t_{p_3}) \cdots), q_{n-1}^\star, t_{p_n})$$

for appropriate $q_1^\star, \cdots, q_{n-1}^\star$. Of course, we can also construct t in bottom-up manner:

$$\mathcal{E}(t_{p_1}, q'^\star_1, \mathcal{E}(t_{p_2}, q'^\star_2, \mathcal{E}(\cdots \mathcal{E}(t_{p_{n-1}}, q'^\star_{n-1}, t_{p_n}) \cdots)))$$

for appropriate $q'^\star_1, \cdots, q'^\star_{n-1}$. Actually, t can be constructed in any order of derivation steps. Every partial derivation tree $t \in T_n(\Sigma)$ can be completed by

- a tree $t_s \in T_s(\Sigma)$, where the only nonterminal leaf in t_s is of sort n, and
- a tree $t_{n'} \in T_{n'}(\Sigma)$ without nonterminal leaves for every node $leaf_{n'}$ in t.

For the search algorithm used below, we need a measure for the size of a derivation tree t that is strictly increasing with the length of a derivation yielding t. Let us review possible options:

1. the number of nodes of t not of the form $leaf_x$ where $x \in N \cup T$ (this corresponds directly to the length of the derivation yielding t),
2. the number of terminal leaves of t, i.e. nodes of t of the form $leaf_x$ where $x \in T$ (for complete derivation trees this corresponds to the length of the derived word),

$$\mathcal{S}(t, q^\star) = \begin{cases} t, & \text{if } q^\star = \langle\rangle \\ \mathcal{S}(t_i, \langle q_2, \ldots, q_l\rangle), & \text{if } t \text{ is of the form } p(t_1, \ldots, t_k), \\ & q^\star = \langle q_1, \ldots, q_l\rangle, q_1 = i \in \{1, \ldots, k\} \\ \text{undefined}, & \text{otherwise} \end{cases}$$

$$\mathcal{E}(t, q^\star, t') = \begin{cases} t', & \text{if } t = leaf_{n'}, q^\star = \langle\rangle \\ p(t_1, \ldots, t_{i-1}, t'_i, t_{i+1}, \ldots, t_k), & \text{if } t \text{ is of the form } p(t_1, \ldots, t_k), \\ & q^\star = \langle q_1, \ldots, q_l\rangle, q_1 = i \in \{1, \ldots, k\}, \\ & t'_i = \mathcal{E}(t_i, \langle q_2, \ldots, q_l\rangle, t') \\ \text{undefined}, & \text{otherwise} \end{cases}$$

Figure 6: Derivation trees: Selection and Extension

3. the total number of nodes in t, and
4. the sum of 1 and 2, i.e. the nodes of t not of the form $leaf_n$ where $n \in N$.

The second choice is not strictly increasing because the extension with elementary derivation trees corresponding to rules without terminals on the right hand side do not introduce terminal leaves. The third choice is not strictly increasing because the extension with elementary derivation trees corresponding to rules of the form $n \rightarrow \epsilon$ only replaces one node by a new one. Thus, in the sequel, the size $|t|$ of a derivation tree t refers to 1 or 4.

Definition 4 (smallest term. deriv. tree)
A derivation tree $t \in T_n(\Sigma)$ without nonterminal leaves is called a smallest terminal derivation tree of sort n, if for all derivation trees $t' \in T_n(\Sigma)$ holds that $|t| \leq |t'|$.

Smallest terminal derivation trees can be calculated searching $T_n(\Sigma)$ starting from $leaf_n$ with a best first search algorithm (cf. [30]) using the extension with elementary derivation trees representing productions as successor relation.

Definition 5 (context tree)
A derivation tree $t \in T_s(\Sigma)$ is called a context tree of sort n, if t has at least one nonterminal leaf of sort n. t is called a terminal context tree, if t has exactly one nonterminal leaf.

Definition 6 (smallest context tree)
A (terminal) context tree t of sort n is called a smallest (terminal) context tree, if for all (terminal) context trees t' of sort n holds that $|t| \leq |t'|$.

As for smallest terminal derivation trees we can use best first search for the calculation of smallest context trees. Combining smallest context trees and smallest terminal derivation trees in an appropriate manner we can construct smallest terminal context trees.

Definition 7 (smallest completion)
A complete derivation tree t is called a completion of a derivation tree t', if there is a subtree t'' of t which can be obtained from t' by replacing the nonterminal leaves from t' by appropriate derivation trees of the corresponding sort. t is called a smallest completion of t', if for all completions t'' of t' holds that $|t| \leq |t''|$. $|t'|_{min}$ is the size of a smallest completion of t'.

This definition also makes clear that a smallest completion t of a derivation tree t' can be constructed from t' itself, a smallest terminal context tree, and smallest terminal derivation trees for all the nonterminal leaves. It is interesting to notice that we could achieve rule coverage by taking words represented by smallest completions of the elementary trees for the various productions as test set. However, for an attribute grammar, we also need to take the semantic dimension into account. We are interested in proper words generated by the AG at hand, and not just by its base grammar. Thus, the notion of a smallest completion is not directly useful as is. In our motivating example, if we start from the production for the goto statement, a smallest completion is a derivation tree encoding the following program:

```
begin goto a end.
```

This is not a valid program, since the label a is neither declared nor defined.

Definition 8 (correct compl. deriv. tree)
A complete derivation tree t is called correct, if the equational system on the attributes of t induced by the computations of t has a solution where the conditions of t hold.

Thus, we have to search for not necessarily smallest, correct completions in a systematic manner. Obviously, the enumeration of all completions, and testing of correctness until a correct completion is found suffices. To conclude the example above, a correct completion of the production for the goto statement is the following:

```
label a; begin a : goto a end.
```

We are now in the position to discuss details of the algorithm for completion. Since we have a lower bound for the size of a completion $|.|_{min}$, we can use a heuristic search algorithm like A* (cf. [6, 30]). The iterative deepening variant IDA* is more practical regarding space requirements. The algorithm is sketched in Figure 7.

```
d_new := |t|_min
repeat
  l := [t]; d := d_new; d_new := ∞
  repeat
    ct := head(l); l := tail(l)
    if ct is complete then
      if ct is correct then
        return ct
      endif
    else
      select a position p of ct for extension
      build the |.|_min-sorted list l'
       of all extensions of ct at p
      l := append(l'_≤d, l);
      d_new := min(d_new, |l'_>d|_min)
    endif
  until l = []
until d_new = ∞
return fail
```

t	derivation tree to be completed
l	list of candidate trees
d	actual tree size limit
d_{new}	limit for the next iteration (the size of the smallest trees not considered so far)
ct	derivation tree currently selected for extension
$l_{\leq d}$	prefix of l (all t with $\|t\|_{min} \leq d$)
$l_{>d}$	postfix of l (all t with $\|t\|_{min} > d$)
$\|l\|_{min}$	minimum of the $\|.\|_{min}$ of the elements of the list l. If l is the empty list then $\|l\|_{min} = \infty$

Figure 7: Completion of derivation trees

For each step of the search algorithm we have two ways to extend the current derivation tree t by an elementary derivation tree t':

- if t has a leaf of sort n, t can be extended at this leaf by a derivation tree t' of sort n through $\mathcal{E}(t, q^\star, t')$ for an appropriate $q^\star$;
- if t is of sort n, t can be extended at the root by a derivation tree t' with a leaf of sort n through $\mathcal{E}(t', q^\star, t)$ for an appropriate $q^\star$.

The freedom to choose where to extend the current derivation tree can be used to tune the search algorithm. Since we can evaluate computations and conditions in a stepwise manner, derivation trees can be rejected as soon as it becomes clear that a condition fails, or the result of a computation is undefined in the case of partial functions. A possible heuristic for the selection of the extension position (root vs. one of the nonterminal leaves) is the most-constraint heuristic, i.e., the position with the greatest number of computations and conditions is selected for extension. This heuristic tries to build the tree in a direction that as many as possible computations and conditions become evaluable as soon as possible.

5.3 Abstract domain coverage

In this section, we assume some definition of the abstract domains $\overline{n}^{AG}$ for (decorated) derivation trees of sort n according to an AG. Basically, a test set achieving coverage is generated by repeated application of the search algorithm from above, where the derived program is only added to the test set if it improves coverage. This is done until full coverage is achieved. Here, we do not rely on approximation coverage and the kind of combination of the two dimensions as proposed in Section 3–Section 4. In practice, we rely on a specific coverage notion in order to guide the search in an efficient manner.

In Figure 8, we present the procedure to construct a test set achieving full coverage (provided it exists). Suppose we want to cover the nonterminal n according to $\overline{n}^{AG}$. Thus, a test set $TS = \{t_1, \ldots, t_l\}$ has to be derived which achieves coverage for n, i.e., $\overline{t_1}^n \sqcup \cdots \sqcup \overline{t_l}^n = \top$. The t_i are derived as follows. Starting from the empty test set, and $\bot$ as the current coverage C, the search algorithm is applied to generate a correct completion t for $leaf_n$ (which is of course a derivation tree of sort s), and we require that t has to increase the coverage C, i.e., $\overline{t}^n \not\leq C$. The generation step is iterated with $C := C \sqcup \overline{t}^n$ while $C < \top$. For subsequent nonterminals, we start with the coverage reached by the test set generated so far. The generation forces the current coverage to strictly increase. Recall that this does not imply that the generated test set is minimal.

- $TS = \{\}$
- For all n:
  - $C := \bigsqcup_{t \in TS} \overline{t}^n$
  - While $C < \top$
    - Generate a correct completion t from $leaf_n$ where $\overline{t}^n \not\leq C$
    - $TS := TS \cup \{t\}$, $C := C \sqcup \overline{t}^n$

Figure 8: Procedure for test set generation

5.4 Guidance of search and termination problems

Due to the structure of the abstract domains as defined by approximation coverage we can use the current coverage to guide the search algorithm. As soon as it becomes clear that no completion of a derivation tree will increase the coverage, we remove it from the candidate list. Actually, this situation arises, if the tree is already covered by the syntactic part of the current coverage, and the attributes contributing to the semantic part of the coverage criterion are already fully covered or bound to values not increasing the coverage. This way, the search space is more and more reduced with the coverage increasing.

Additionally, we can select as position for extension the one with the greatest number of attributes, which contribute to the coverage criterion, depending on its attributes. The selection of the position might also be driven by incomplete

coverage in the syntactic sense. This way, the search algorithm is guided into a direction increasing the coverage as soon as possible. If the current derivation tree is known to increase the coverage but it is not yet complete, the usual search algorithm can be used to find a correct completion.

If full coverage is possible, the search algorithm will terminate, since it essentially enumerates the derivation trees. If full coverage is impossible, the search algorithm may not terminate. A guided search as proposed above may reduce the search space to become finite, and thus make the search algorithm terminate. Due to the expressiveness of AGs, it is in general not decidable, if the search may lead to correct derivation trees improving coverage. Thus, termination cannot be guaranteed. The means of configuration discussed in Section 4.4 need to be used to recover feasibility of coverage. Ultimately, we can enforce termination by restricting the search depth. For most decent attribute grammars it should be possible to limit the search depth, e.g., by restricting the number of recursive unfoldings.

6 Concluding remarks

Results The first contribution of the paper is a general and intuitive notion of coverage for attribute grammars and other kinds of declarative programs. The notion of approximation coverage takes the context-free part and the attributes in an attribute grammar into account, and it goes strictly beyond syntactic rule coverage. It covers more aspects of the program to be tested in an intuitive manner. A certain complexity of derivation trees and attribute values is enforced relying on an unfolding technique to cope with recursion.

The second contribution of the paper is an algorithm for test set generation complementing the coverage notion. Only correct decorated derivation trees are generated. The generator algorithm relies on some kind of breadth-first search, and it starts from an elementary derivation tree which is then completed. The generated test cases are in some sense as small as possible. Therefore, redundancy is reduced. Termination cannot be ensured in general, but means to recover termination have been indicated. Also, non-termination can be observed to a certain extent, if a certain search depth is considered as harmful.

Related work In [26], Purdom gives an algorithm to generate a small set of short words from a context-free grammar where each production of the grammar is used in the derivation of at least one word. There are some attempts to extend Purdom's approach in different ways in order to take context conditions into account so that correct programs are generated. However, more sophisticated coverage notions than rule coverage for context-free grammars or attribute grammars do not exist in the literature.

In [9], a special grammar formalism with actions working on the internal data structures of a generator is used. If during the generation process a violation of the context conditions is encountered, the actions cause that text is inserted in or deleted from the generated word at dedicated positions. Thereby, it should be guaranteed that the resulting word respects the context conditions.

A more declarative approach is pursued in [2] using context-free parametric grammars. The parameters associated with the grammar symbols can take values from context-free languages. When a parameter value is needed and not yet defined during the generation process, a value has to be generated or provided by the user. Only words derivable with respect to the generated or user-provided parameter values can be generated. A survey of further approaches used in test set generation for compiler testing is given in [7].

Some authors resort to randomized test set generation (cf. [4, 22, 8]) hoping that the resulting test sets—if large enough—will include all interesting language aspects. Note also that only generation is facilitated. Using our approach coverage measurement of test suites or manually provided test cases can also be performed. There are other approaches to coverage measurement for test programs which are not based on a reference AG. In [10, 5], program instrumentation is used to gather coverage information from prototypical language implementations.

In [17], Jack gives a coverage notion and an algorithm for test set generation for logic programs based on anti-unification. The main problem with the given form of anti-unification is that sums are not treated in a sufficiently precise manner. Two terms with different functors suffice to get full coverage even if there are further functors of the same sort. In that sense, approximation coverage provides a useful notion for testing logic programming. Jack also assumes that the test sets are generated on a per predicate basis. Translated to the AG context, this means that each nonterminal would have to be tested separately. This is not compatible with the concept of a start symbol. From a practical perspective, for attribute grammar implementations, e.g., compilers and interpreters, usually only words derived from the start symbol can be used for testing.

Future work We have reasonable experience in generating test sets for simple rule coverage while aiming at correct derivation trees (cf. [15, 16]). The technique is feasible for non-trivial language definitions. Using the more general notion of approximation coverage blindly, the generated test sets tend to get too complex to be helpful in actual testing. Thus, a primary subject for future work is a feasibility study to apply the technique to a Pascal-like language, and to work out some pragmatic properties of the technique back-to-back. We mentioned some techniques to focus on rules, nonterminals or attributes. The ultimate goal in this respect is a test case generation language.

Another subject for future work concerns the termination problems discussed in the paper. A substantial part of the termination problems can be resolved if the grammar and/or the attribute types are refined. We believe that most decent AGs can be completed in this way to achieve

normal termination. However, such refinements put a burden on the programmer. Also, it had to be ensured that the grammar refinements preserve the generated language, and that type refinements are sound. One possible direction for improvements is to derive the refinements (e.g., in the form of annotations) largely automatically by a kind of type inference.

There is a related problem, that is to say, the complexity of test set generation. Especially, in the semantic dimension, too little information is used to guide the search discussed in this paper. The conditions and computations are regarded as black boxes. Also, attribute dependencies are not yet used for guidance. A white-box approach, where conditions and computations are specified, for example, as recursive functions, and attribute dependencies are taken into account, could be used to guide the generator algorithm.

The type inference proposed above also relies on the white-box setting. We want to infer, for example, that a function implementing a computation from an AG is only defined on certain parameter patterns. A particular way to implement the white-box approach and to take attribute dependencies into account is based on constraint-logic programming. The computations and conditions are implemented as constraints. We have done very promising experiments in this direction. The constraint system cuts off many parts of the search space, and cases, where full coverage is not possible, are often identified.

References

[1] H. Alblas. Introduction to Attribute Grammars. In H. Alblas and B. Melichar, editors, *Proceedings of International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, June 1991.

[2] F. Bazzichi and I. Spadafora. An Automatic Generator for Compiler Testing. *IEEE Transactions on Software Engineering*, 8(4):343–353, July 1982.

[3] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.

[4] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[5] A. Boujarwah and K. Saleh. Compiler test suite: evaluation and use in an automated test environment. *Information and Software Technology*, 36(10):607–614, 1994.

[6] A. Bundy, editor. *Artificial Intelligence Techniques : A Comprehensive Catalogue*. Springer-Verlag, 4th rev. edition, 1997.

[7] C. J. Burgess. The Automated Generation of Test Cases for Compilers. *Software Testing, Verification and Reliability*, 4(2):81–99, jun 1994.

[8] C. J. Burgess and M. Saidi. The automatic generation of test cases for optimizing Fortran compilers. *Information and Software Technology*, 38(2):111–119, Feb. 1996.

[9] A. Celentano, S. Crespi Reghezzi, P. Della Vigna, C. Ghezzi, G. Granata, and F. Savoretti. Compiler Testing using a Sentence Generator. *Software – Practice and Experience*, 10:897–918, 1980.

[10] Z. J. Cichanowicz and A. C. De Weever. The 'Completeness' of the Pascal Test Suite. *Software – Practice and Experience*, 14(5):463–471, May 1984.

[11] M.-M. Corsini and K. Musumbu. Failure Analysis based on Abstract Interpretation. In J. Darlington and R. Dietrich, editors, *PHOENIX Seminar and workshop on declarative programming*, Workshops in Computing, pages 295–309, London, Nov. 18–22 1992. Springer-Verlag.

[12] B. Courcelle and P. Deransart. Proofs of partial correctness for attribute grammars and recursive procedures. Technical Report RR-0322, INRIA Rocquencourt, July 1984.

[13] R. A. DeMillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, Sept. 1991.

[14] M. J. Foster. Software validation using abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 2, pages 32–44. Ellis Horwood, 1987.

[15] J. Harm. Automatic Test Program Generation from Formal Language Specifications. *Rostocker Informatik-Berichte*, 20:33–56, 1997.

[16] J. Harm, R. Lämmel, and G. Riedewald. The Language Development Laboratory ($^{\Lambda}\Delta_{\Lambda}$). In M. Haveraaen and O. Owe, editors, *Selected papers from the 8th Nordic Workshop on Programming Theory, December 4–6, Oslo, Norway, Research Report 248*, pages 77–86. University of Oslo, May 1997.

[17] O. Jack. *Software Testing for Conventional and Logic Programming*. Number 10 in Programming Complex Systems. Walter de Gruyter, Berlin, 1996.

[18] N. D. Jones and F. Nielson. Abstract Interpretation. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4. Clarendon Press, Oxford, 1995.

[19] U. Kastens. Studie zur Erzeugung von Testprogrammen für Übersetzer. Bericht 12/80, Institut für Informatik II, University Karlsruhe, 1980.

[20] A. Kühnemann and H. Vogler. *Attributgrammatiken*. vieweg Verlag, 1997.

[21] R. Lämmel and G. Riedewald. Provable Correctness of Prototype Interpreters in LDL. In P. A. Fritzson, editor, *Proceedings of Compiler Construction CC'94, 5th International Conference, CC'94, Edinburgh, U.K.*, volume 786 of *Lecture Notes in Computer Science*, pages 218–232. Springer-Verlag, 1994.

[22] V. Murali and R. Shyamasundar. A Sentence Generator for a Compiler for PT, a Pascal Subset. *Software – Practice and Experience*, 13(9):857–869, Sept. 1983.

[23] G. J. Myers. *The Art of Software Testing*. Wiley - Interscience, New York, 1979.

[24] A. J. Offutt and S. D. Lee. An Empirical Evaluation of Weak Mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.

[25] J. Paakki. Attribute Grammar Formalisms — A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[26] P. Purdom. A sentence generator for testing parsers. *Behaviour and Information Technology*, 12(3):366–375, July 1972.

[27] M. Rosendahl. Strictness Analysis for Attribute Grammars. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming (PLILP '92)*, volume 631 of *Lecture Notes in Computer Science*, pages 145–157, Berlin, 1992. Springer-Verlag.

[28] T. A. Sudkamp. *Languages and Machines*. Addison-Wesley, Reading, Massachusetts, 1988.

[29] P. Wegner. The Vienna Definition Language. *ACM Computing Surveys*, 4(1):5–63, Mar. 1972.

[30] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1992.

A Multi-phase Parallel Algorithm for the Eigenelements Problem

Abdelhamid Benaini and David Laiymani
Faculté des Sciences du Havre, 25, Rue Philippe Lebon, 76058 Le Havre Cedex. France
Phone: (00) 02 32 74 43 02, Fax: (00) 02 32 74 43 02
E-mail: benaini@univ-lehavre.fr

Keywords: eigenelements, sub-spaces method, parallel reconfigurable machine, multi-phase model.

Edited by: Rudi Murn
Received: October 28, 1999 **Revised:** April 8, 2000 **Accepted:** May 15, 2000

This paper presents a parallel implementation of a variant of the QR method for the eigenelements problem. This method consists in factorizing a symmetric matrix A in the form $A = QXQ^T$ where Q is an orthonormal matrix and X has nonzeros components only on main and cross diagonals. We present a multi-phase parallel implementation of this method onto a reconfigurable machine. We decompose this method into a series of standard parallel computations and for each of them we choose the best interconnection topology in order to speed up the communication time. Numerical tests corroborate nicely a theoretical evaluation of our parallel algorithm.

1 Introduction

The numerical solutions of the eigenelements of a large matrix arise in numerous scientific applications. The most popular methods used to solve this problem are the Jacobi's algorithm, the QR method or the Housholder's transformation and the methods based on projection techniques onto appropriate sub-spaces such as Lanczoz's and Davidson's methods. To speed up the associated computations, many parallel algorithms have been presented and their implementations present some particularities depending on the target parallel architecture [18, 13, 2, 10, 12, 15, 9].

This paper focuses on the method presented in [4] for computing the eigenvalues and the corresponding eigenvectors of a symmetric matrix A. It consists in factorizing A into the form $A = QXQ^T$ where Q is orthonormal and X is a symmetric matrix having nonzero components only on main and cross diagonals. The method takes ideas from the generalized WZ factorization [5, 6, 8] in which the associated sequence of computational operations is more suitable for parallel processors than the classical methods.

Our second aim is to expose a parallel implementation of this algorithm on a dynamically reconfigurable machine. A parallel machine is called dynamically reconfigurable if its interconnection network can be altered during the execution of the same application [5, 3]. This yields a variety of possible topologies for the network and allows a program to exploit this topological variety in order to speed up the computation. A possible network topology is any one in which the number of connections per processor is less than or equal to a constant d. The underlying algorithmic model for reconfigurable machines is called *multi-phase* model [19, 1]. This model relies on the idea that a parallel algorithm can be decomposed into series of elementary data movements. So, programs are designed so as to execute a series of phases. Each phase uses its own topology which suits in the best way communication requirements and phases are assumed to be separate from one another, by synchronization-reconfiguration points [1]. In this way the communication cost of an application is reduced and designing a multi-phase algorithm consists in finding the best phase decomposition and the best topology for each phase.

This paper is organised as follows. We first present the sequential method for the eigenelements computation. Next we show how this method can be divided into a series of parallel phases, each of them corresponding to standard parallel computations. We also analyse how to choose the best topologies (i.e. the topologies which minimize the communication cost) to perform these computations. Section 4 presents a theoretical evaluation of the multi-phase algorithm and some numerical experiments and a comparison with the sub-spaces.

2 The sequential algorithm

Let A be a symmetric matrix of order n with n real eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$. We assume that the multiplicity of each λ_i is ≤ 2 and that $|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_n|$. Let $< u, v >$ denote the scalar product of u and v and $||u||$ the Euclidean norm of u. The method, introduced in [4] for computing the eigenelements of the matrix A, consists in computing an orthogonal matrix Q_1 such that

$A^{(1)} = Q_1 A Q_1^T$ with $A^{(1)}$ symmetric of the form :

$$A^{(1)} = \begin{pmatrix} a_{11}^{(1)} & & 0 & & a_{1n}^{(1)} \\ & a_{22}^{(1)} & \cdots & a_{2,n-1}^{(1)} & \\ 0 & \vdots & a_{ij}^{(1)} & \vdots & 0 \\ & a_{n-1,2}^{(1)} & \cdots & a_{n-1,n-1}^{(1)} & \\ a_{n1}^{(1)} & & 0 & & a_{nn}^{(1)} \end{pmatrix}$$

Let $(a_{ij}^{(1)})$ denote the elements of $A^{(1)}$, then $a_{1j}^{(1)} = a_{j1}^{(1)} = a_{nj}^{(1)} = a_{jn}^{(1)} = 0$ for $2 \leq j \leq n-1$. Let $Q_1 = (q_1, \ldots, q_n)$ where $q_i \in R^n$, denotes the i-th column of Q_1. In this way, $A = Q_1 A^{(1)} Q_1^T$ implies the following two systems

$$(S_1) \begin{cases} Aq_1 = a_{11}^{(1)} q_1 + a_{1n}^{(1)} q_n \\ Aq_n = a_{1n}^{(1)} q_1 + a_{nn}^{(1)} q_n \\ < q_1, q_n > = 0 \\ \|q_1\| = \|q_n\| = 1 \end{cases}$$

$$(S_2) \begin{cases} Aq_j = \sum_{k=2}^{n-1} a_{kj}^{(1)} q_k \\ \forall\ 2 \leq j \leq n-1 \end{cases}$$

Thus (S_1) is a non-linear system of $2n+3$ equations and $2n+3$ unknowns $(a_{11}^{(1)}, a_{1n}^{(1)}, a_{nn}^{(1)}$ and the $2n$ components of q_1 and $q_n)$. The following algorithm (derived from the subspace method) is proposed in [4] to get a solutions q_1 and q_n of system (S_1). Others solutions for (S_1) are obtained by applying a rotation to vectors u and v. Further we show how to deduce λ_1, λ_2, and the corresponding eigenvectors from the solutions of (S_1).

Algorithm for solving (S_1)

Let $u^{(0)}$ and $v^{(0)}$ two orthonormal vectors of R^n;
For k=0,1,... until convergence do

$\gamma_{1,k} = \|Au^{(k)}\|$;
$\gamma_{2,k} = \|Av^{(k)}\|$;
$\gamma_{3,k} =< Au^{(k)}, Av^{(k)} >$;
$\gamma_k = \sqrt{\gamma_{1,k}^2 \gamma_{2,k}^2 - \gamma_{3,k}^2}$;
$\tan\theta_k = \frac{\gamma_{3,k}}{\gamma_{1,k}^2 + \gamma_k}$;
$\cos\theta_k = \frac{1}{\sqrt{1+\tan^2\theta_k}}$;
$\sin\theta_k = \frac{\tan\theta_k}{\sqrt{1+\tan^2\theta_k}}$;

$$B_k = \begin{pmatrix} \cos\theta_k & -\sin\theta_k \\ \sin\theta_k & \cos\theta_k \end{pmatrix} \begin{pmatrix} \frac{1}{\gamma_{1,k}} & 0 \\ -\frac{\gamma_{3,k}}{\gamma_{1,k}\gamma_k} & \frac{\gamma_{1,k}}{\gamma_k} \end{pmatrix};$$

$$\begin{pmatrix} u^{(k+1)} \\ v^{(k+1)} \end{pmatrix} = B_k \begin{pmatrix} Au^{(k)} \\ Av^{(k)} \end{pmatrix}$$

Done

According to the definition of θ_k^s, B_k is symmetric, inversible and

$$\lim_{k\to\infty} B_k^{-1} = B^{-1} = \begin{pmatrix} a_{11}^{(1)} & a_{1n}^{(1)} \\ a_{1n}^{(1)} & a_{nn}^{(1)} \end{pmatrix}$$

The eigenvalues of B^{-1} are λ_1 and λ_2 and

$$q_1 = \lim_{k\to\infty} u^{(k)}, \quad q_n = \lim_{k\to\infty} v^{(k)}$$

are a solutions of (S_1). On the other hand,

$$e_1 = q_1 + r_1 q_2, \quad e_2 = q_1 + r_2 q_2 \tag{1}$$

where

$$r_1 = \frac{a_{nn}^{(1)} - a_{11}^{(1)} + (\lambda_1 - \lambda_2)}{2a_{1n}^{(1)}}$$

$$r_2 = \frac{a_{nn}^{(1)} - a_{11}^{(1)} - (\lambda_1 - \lambda_2)}{2a_{1n}^{(1)}}$$

are two orthogonal eigenvectors respectively associated with λ_1 and λ_2.

The angle θ_k of the algorithm is defined in such a way that the B_k matrix is symmetric. Nevertheless, to determine this angle, other choices are discussed in [4]. If we take $\theta_k = 0, \forall k > 0$ in the algorithm, then we find the sub-spaces method for computing the two dominant eigenvalues of A. The introduction of the rotation $\begin{pmatrix} \cos\theta_k & -\sin\theta_k \\ \sin\theta_k & \cos\theta_k \end{pmatrix}$ which is, as it was, a relaxation factor of the sub-spaces method allows an acceleration of the algorithm's convergence (see [4] for details).

Having computed $(q_1, q_n, a_{11}^{(1)}, a_{1n}^{(1)}, a_{nn}^{(1)})$, the other vectors $q_j, 2 \leq j \leq n-1$ can be determined using the Gram-Schmidt method. Next, we get from $(S_2))$

$a_{kj}^{(1)} =< q_k, Aq_j >, \forall\ 2 \leq j,k \leq n-1$
and

$a_{nj} = a_{jn} = a_{1j} = a_{j1} = 0, \forall 2 \leq j \leq n-1$

Similarly the symmetric matrix $A_1 = (a_{ij}^{(1)})_{2\leq i,j\leq n-1}$ of order $n-2$ can be decomposed into the form $A_1 = Q_2 A^{(2)} Q_2^T$ and so on. This process results after $q = \lfloor \frac{n-1}{2} \rfloor$ steps in an orthonormal matrix Q and a matrix X having nonzero elements only on main and cross diagonals such that $A = QXQ^T$.

Note that a decomposition $A = JXJ^T$ can be achieved, using a variant of the Jacobi method [17]. In this case the problem size remains unchanged, equal to n, during the execution of the algorithm.

Our aim is to compute the eigenelements of A. So, to avoid the use of Gram-Schmidt method, we use the deflation technique to implement this algorithm in this way : after the execution of the first step, i.e. the computation of $a_{11}^{(1)}, a_{1n}^{(1)}, a_{nn}^{(1)}, q_1, q_2, \lambda_1, \lambda_2$, we consider the matrix

$A_1 = A - \lambda_1 e_1 e_1^t - \lambda_2 e_2 e_2^t$

The eigenvalues of A_1 are $0, 0, \lambda_3 \ldots \lambda_n$, each of multiplicity ≤ 2 and the two dominant eigenvalues of A_1 are λ_3 and λ_4. So A_1 satisfies the same hypothesis as A. Therefore the method can be applied to A_1 in order to compute λ_3 and λ_4 and the corresponding eigenvectors. These two steps (computing two eigenelements and updating A by deflation) are repeated $q = \lfloor \frac{n-1}{2} \rfloor$ times. Note that in the Gram-Schmidt method, the size of the A matrix does not decrease.

3 The multi-phase parallel algorithm

In this section, we present a multi-phase parallel analysis for the method previously exposed. First we present the computational model and next we detail the different phases.

3.1 Computational model

The computational model used throughout this work is the *multi-phase model*. in this model an algorithm is implemented as a series of phases, so that, each phase is efficiently executed on the processor graph (of degree d) that exactly reflects the need of the current data transfer pattern [6]. Phases are assumed to be separated from one another, by synchronization-reconfiguration points. This model assumes a reconfigurable machine where physical interconnections are set before the beginning of a phase. Let a reconfigurable machine with p identical processors, each of them own d bidirectional communication links. We assume that it is possible to perform in parallel on the same processor, bidirectional data transfers on each link. In order to refer a node, the p processors are viewed as a ring. A processor located at the i-th position ($0 \leq i \leq p-1$) is labelled P_i.

>From an algorithmic point of view, the *multi-phase* model provides two main advantages:

- Improvement in the performance of an application. Indeed, for a parallel distributed memory machine, communications are often a restrictive factor. Then, the performance of a parallel algorithm depends on how well its communication graph matchs the interconnection network of the target parallel machine. In this way, parallel systems with static interconnections require to adapt algorithms to the architecture. The conception of such algorithms is often difficult because there is no ideal topology for a set of algorithm and because of the intractable problem of mapping an algorithm onto a parallel system. The use of a router can remove the designing problems but in any case, the effects of the architecture limitation involve an increase of the communication costs. Reconfigurable machines overcome this problem because they allow to adapt the topology of the interconnexion network to the needs of the specific application.
- An informal approch (like the sequential top-down analysis) allows to conceive efficient multi-phase parallel algorithms. This approch was presented in [6, ?] and it consists in decomposing a problem by successive refinements in order to get a sequence of elementary sub-routines which solves the problem. In the multi-phase model we propose to apply, to the communication scheme of a parallel algorithm, a succession of refinements steps in order to get elementary topologies (of degree d at most) and sub-problems corresponding to phases. In the remainder of this paper we propose to illustrate this methodology.

3.2 Principles

This section presents a phases decomposition of the sequential method. This decomposition is based on the computation needs of the algorithm. In the next sections we discuss the choice of an adequate topology for each phase.

The inner loop (*for* $k = 0, 1 \ldots$ *until convergence*) of the sequential algorithm allows to compute two eigenelements. At step k we begin by computing the two matrix-vector products $Au^{(k)}$ and $Av^{(k)}$. Next we compute the different values $\gamma_{1,k}, \gamma_{2,k}, \gamma_{3,k}, \gamma_k, \tan\theta_k^s, \cos\theta_k^s$ and $\sin\theta_k^s$.

After convergence we update the $A^{(k)}$ matrix by deflation and we start the computation of the next two eigenelements (step $k+1$).

Hence, for each k, the parallel multi-phase algorithm is naturally composed of the following phases :

- computing $Au^{(k)}$ and $Av^{(k)}$,
- computing $\gamma_{1,k}$, $\gamma_{2,k}$ and $\gamma_{3,k}$,
- updating $A^{(k)}$ by deflation.

It is clear that the most time consuming process is the matrix-vector products of phase 1. So, the initial data distribution must be performed in order to get the best parallelization of this process. We have choosen the *contiguous row decomposition* scheme in which each row is stored entirely in one processor and, starting with the first row, every contiguous $r = \frac{n}{p}$ rows are stored in the same processor. This data distribution is illustrated in figure 1.

Note that at step k of the outer loop, the size of the problem is $n-2(k-1)$. Thus, it is necessary to include another phase which performs a load balancing of the remaining data.

3.3 $Au^{(k)}$ and $Av^{(k)}$ computation

For the sake of simplicity, we let $u^{(k)} = u = (u_i)_{0 \leq i < n}$ and $v^{(k)} = v = (v_i)_{0 \leq i < n}$, $x = (x_i)_{0 \leq i < n}$, $y = (y_i)_{0 \leq i < n}$. This first phase computes the product $(x, y) = A.(u, v)$ of a $n \times n$ matrix A by two vectors u and v of size n. The parallelization of this problem has been extensively studied in the literature [14].

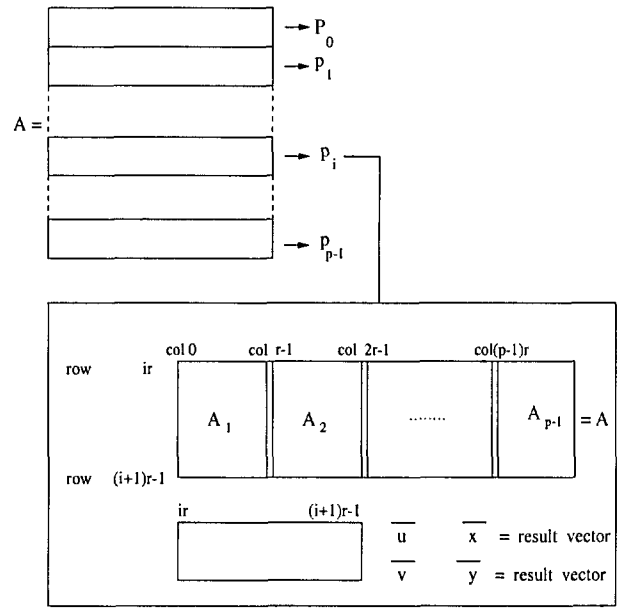


Figure 1: Row data decomposition

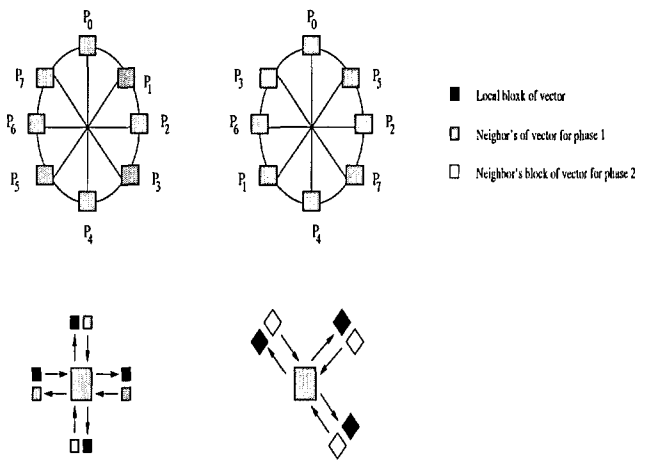


Figure 2: The multi-phase matrix-vector product for $p = 8$ and $d = 4$

We propose a new algorithm for the matrix-vector product on a dynamically reconfigurable machine. This algorithm assumes the data decomposition of figure 1 and runs in $\lceil \frac{p-1}{d} \rceil$ phases. The topology of each phase is a *chordal ring* of degree d (see figure 2). First, each processor P_i, $0 \leq i \leq p-1$ computes $A_i\bar{u}$ and $A_i\bar{v}$ while it sends its blocks $\bar{u}$ and $\bar{v}$ to the set S of its neighbors processors. Then each processor can compute $A_j\bar{u}$ and $A_j\bar{v}$ for $j \in S$. Clearly the algorithm ends after $\lceil \frac{p-1}{d} \rceil$ steps which correspond to the different phases. Figure 4 illustrates this algorithm for $p = 8$ and $d = 4$.

We point out that communications and computations can be overlaped during the entire execution of the algorithm and that the number of step is d times smaller than those of the algorithm presented in [14]. We do not focus on the reconfiguration cost because it is negligible and furthermore it can be overlaped by computations too. The optimal packet size for which communications are entirely overlaped by computations, can be computed using techniques presented in [11].

3.4 $\gamma_{1,k}, \gamma_{2,k}$ and $\gamma_{3,k}$ computation

As shown in section 3.2, the computation of $\gamma_{1,k}, \gamma_{2,k}$ and $\gamma_{3,k}$ allows to compute $u^{(k)}$ and $v^{(k)}$. This phase consists in computing three scalar products. In fact, $\gamma_{1,k} =< x, x >$, $\gamma_{2,k} =< y, y >$ and $\gamma_k =< x, y >$ where $x = Au^{(k)}$ and $y = Av^{(k)}$ have been processed in phase 1 and are distributed among the processors. Furthermore, these scalar products must be broadcasted to all the network in order to compute $u^{(k+1)}$ and $v^{(k+1)}$. So this phase can be summarized as follows :

- All the processors P_i for $0 \leq i \leq p-1$, compute in parallel

$$\gamma_i^{(1)} = \sum_{j=ir}^{(i+1)r-1} \bar{x}_j^2,$$
$$\gamma_i^{(2)} = \sum_{j=ir}^{(i+1)r-1} \bar{y}_j^2,$$
$$\gamma_i^{(3)} = \sum_{j=ir}^{(i+1)r-1} \bar{x}_j\bar{y}_j.$$

- Perform a reduction operation in order to lead the processor P_0 to compute $\gamma_{1,k}, \gamma_{2,k}$ and γ_k
- P_0 broadcast $\gamma_{1,k}, \gamma_{2,k}$ and $\gamma_{3,k}$ in the network.

The reduction operation is defined as follows. Each processor P_i, $0 \leq i \leq p-1$, holds three data item $\gamma_i^{(1)}, \gamma_i^{(2)}$ and $\gamma_i^{(3)}$ and processor P_0 has to compute

$$\gamma_{1,k} = \gamma_0^{(1)} + \ldots + \gamma_{p-1}^{(1)}$$
$$\gamma_{2,k} = \gamma_0^{(2)} + \ldots + \gamma_{p-1}^{(2)}$$
$$\gamma_{3,k} = \gamma_0^{(3)} + \ldots + \gamma_{p-1}^{(3)}$$

Recall that computation and communication can be overlaped and that communications can occur in parallel on all links. A simple strategy for performing reduction operations uses a tree-based interconnection network [16]. Each node sends the result of the reduction operation of its own subtree to its father. Intuitively, given p, the larger the degree of the tree, the smaller the levels in the tree, hence the less costly the communication.Moreover, the larger the degree of the tree, the smaller the number of processor that perform arithmetic operations in parallel (leaf processors do not perform any arithmetic) [16]. There is a tradeoff to be found that depends upon the ratio between the communication time and the computation time. It is shown in [16] how to determine the best tree-based topology of degree d as a function of p and of the communication and computation times. Particularly, when the ratio communication/arithmetic is high (which is the case of our test

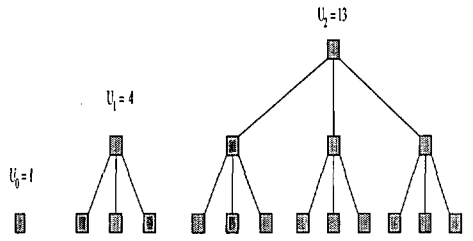


Figure 3: A communication tree: $d = 4$, $p = 13$

machine), they defined a tree-based topology, called *communication trees* as follows. For $p \in \{w_t : t \geq 0\}$ where $w_0 = 1$, $w_{t+1} = (d-1)w_t + 1$ for $t \geq 0$, the construction consists in concatening $(d-1)$ communication trees of height t to get a communication tree of height $t+1$ (see figure 3). Clearly, the reduction operation can be optimaly performed on a communication tree of degree d, with p processors, so that P_0 computes $\gamma_{1,k}, \gamma_{2,k}$ and $\gamma_{3,k}$. The broadcast of these values can be performed on the same tree-based topology in an optimal way.

Remark that in [7], we propose a multi-phase broadcast procedure better than the tree topology, for large messages. Here we broadcast messages of small size, so this procedure is not suitable.

The phases 1 and 2 are repeated until $||x - u^{(k)}|| + ||y - v^{(k)}|| \leq \varepsilon$ where ε is the accuracy. After this step, each node computes the two dominant eigenvalues γ_1 and γ_2 and the associated eigenvectors according to Section 2.

3.5 The deflation phase

In this phase the matrix A is updated in this way : $A = A - \lambda_1 e_1 e_1^t - \lambda_2 e_2 e_2^t$ where e_1 and e_2 are defined by (1), (2) and are distributed among the processors. Using the strategy presented for the phase 1 the *outer products* $e_1 e_1^t$ and $e_2 e_2^t$ can be easily computed on a chordal ring topology of degree d. Then, in parallel, all nodes are able to update the matrix A.

We have exposed the multi-phase algorithm for the first step of the sequential method. The same analysis holds for steps $2, 3, \ldots \lfloor \frac{n-1}{2} \rfloor$. At step 1 the problem size is equal to n and a step 2 it becomes equal to $n-2$ and at step k it becomes equal to $n - 2(k-1)$ inducing that processors P_0 and P_{p-1} will become inactive. Thus, it is necessary to balance the data distribution. This could be done on a ring topology with a shifting of rows as shown in figure 4.

The three phases are iterated $\lfloor \frac{n-1}{2} \rfloor$ times. The multi-phase algorithm is summarized in figure 4.

3.6 Complexity analysis

Let $q = \lfloor \frac{n-1}{2} \rfloor$ and I_ϵ^k be the number of iterations required to get the convergence of the algorithm with an accuracy ϵ

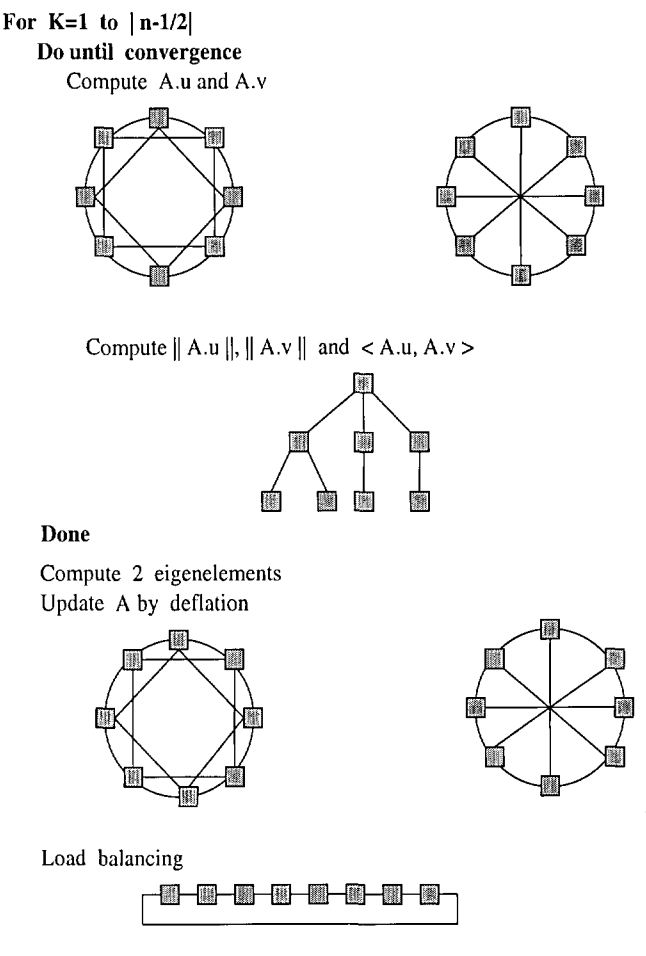


Figure 4: The multi-phase algorithm for $d = 4$ and $p = 8$

at step k. Let $m = n-2(k-1)$ and $t(L)$ be the time needed to transfer a message of L floating numbers between two physically connected processors.

Let t_{rec} be the time needed to reconfigure the interconnection network between two consecutive phases. As phases 1 and 2 are repeated until convergence of the total reconfiguration cost is $(\lceil \frac{p-1}{d} \rceil + 1)I_\epsilon^k + 2)qt_{rec}$.

As exposed in section 3, during the first phase of the multi-phase algorithm (matrix-vector products) communications are overlaped by computations and thus, we can assert that communication costs of phase 1 are null. In phase 2, both reduction and broadcast operations are performed with a communication cost of $\log pt(3)I_\epsilon^k q$. Phase 3 consists in the same strategy as phase one and consequently this phase presents no communication cost. Moreover, the load balancing process has a communication cost of $O(m)$. Therefore, the total communication cost of the multi-phase algorithm is $O(n \log p)$.

The number of floating operations for the matrix-vector product phase is $O(\frac{m^2}{p})$. For computing $\gamma_{1,k}, \gamma_{2,k}$ and $\gamma_{3,k}$ each processor executes $O(\frac{m}{p})$ floating operations (3 local scalar products to compute). During the data reduction $4 \log pI_\epsilon$ additions are necessary. So, the number of floating operations for the norms computation phase is $O(\frac{m}{p} + \log p)$.

For the deflation phase, the number of floating operations required is $O(\frac{m^2}{p})$. Finally the computation of two eigenelements and the updating of the A matrix has a cost of $O(\frac{m^2}{p})$.

4 Numerical tests and concluding remarks

We report in this section, numerical experiments for $\theta_k = \theta_k^s$ and $\theta_k = 0$, on a SuperNode machine with $p = 16$ and $d = 4$, running with the C_Net programming environment [1]. Because of memory limitation (1 Mo per processor) n is limited to 256. The test matrix is $A = (a_{ij})_{0 \leq i,j \leq n-1}$ where $a_{ij} = n - i$ if $j \leq i$ and $a_{ij} = n - j$ if $i > j$.

Figure 5 shows the speed up of the multi-phase algorithm for different values of p and n. Recall that the communication time grows as $O(n \log p)$ while the execution time grows as $O(\frac{n^2}{p})$. So, for a given value of p (respectively of n), the larger n is (respectively the smaller p is), the more the communication time becomes insignificant. These remarks explain the shape of the different curves of figure 5 and explain why the speed up is not optimal for $p = 16$ and $n = 256$ and optimal for $p = 4$ and $n = 256$. According to this discussion, one can say that for larger values of n and p for which the communication cost $O(n \log p)$ is negligible compared to the execution time $O(\frac{n^2}{p})$ the speed up becomes optimal. As expected the shape of the different curves shows a polynomial growth in $O(\frac{n^2}{p})$ of the speed-Up of this parallel program. In the same way, the communication time illustrated in figure 6 grows as $O(n \log p)$

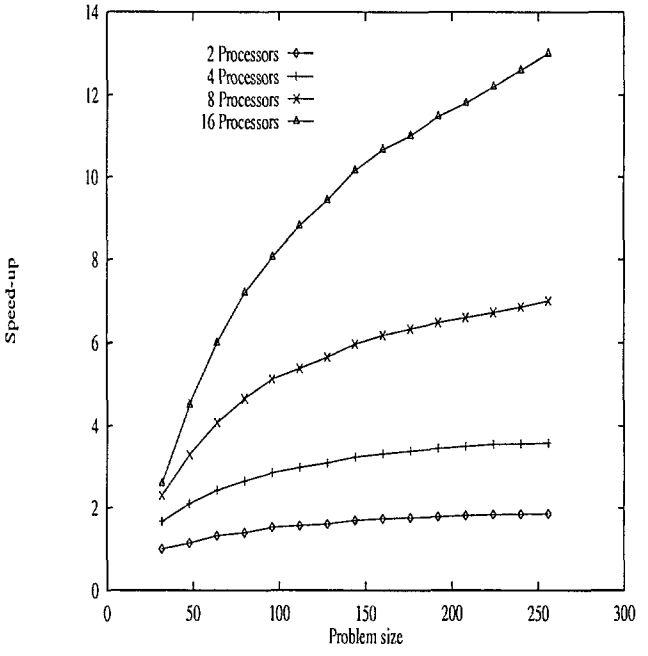


Figure 5: Speed-Up of the multi-phase algorithm for $\theta_k = \theta_k^s$ and $\varepsilon = 10^{-3}$

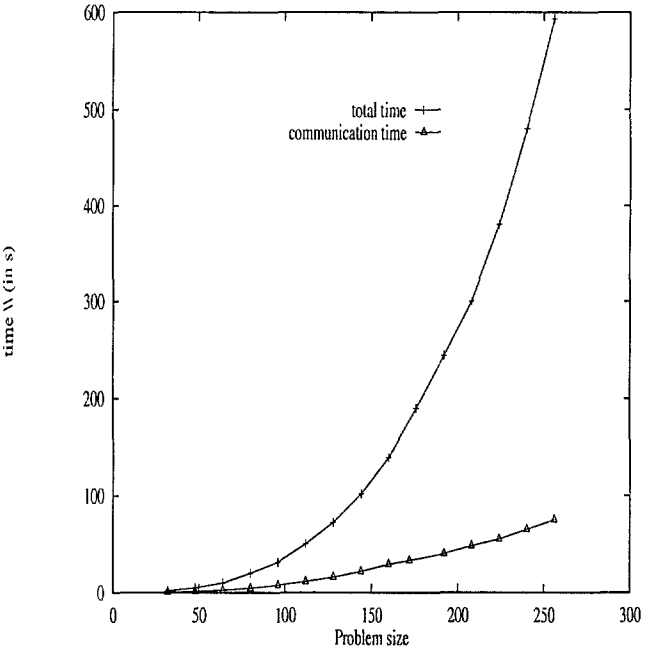


Figure 6: Computation time and communication time for $\theta_k = \theta_k^s$ $\varepsilon = 10^{-3}$ and $p = 16$

and corroborate the theoretical predictions of the previous section.

Finally, we underline that the experiment tests reported in [4] show that the case $\theta_k = 0$ requires, for n large, approximatively the double number of iterations and the double execution time than the case $\theta_k = \theta_k^s$. We do not know, theoretically, the relation between the convergence factors in the two cases. On the other hand, the case $\theta_k = 0$, requires a very fine accuracy ϵ. For instance, for $n \geq 64$ only precisions superior to 10^{-6} give correct results. On the contrary, for $\theta_k = \theta_k^s$, $\epsilon < 10^{-1}$ is sufficient. Also in that case, the problem of the numerical stability of the algorithm has to be studied. Nevertheless, theses numerical experiments show that with a fine accuracy ($\epsilon = 10^{-8}$), the algorithm is stable and converges more rapidly than the sub-spaces method.

References

[1] J-M. Adamo and L. Trejo (1994) Programming Environment for Phase-reconfigurable Parallel Programming on SuperNode *JPDC*, 23, 278-292.

[2] A. Basermann and P. Weidner (1992) A Parallel Algorithm for Determining all Eigenvalues of Large Real Symmetric Tridiagonal Matrices. *Parallel Computing*, 18, p. 1129-1141.

[3] Y. Ben-Asher et al. (1991) The Power of Reconfiguration. *JPDC*, 13, p. 139-153.

[4] A. Benaini and M. Drissi (1995) Generalization of a subspace method for the symmetric eigenelements problem. *CWI Quaterly*, 8, 3, p. 257-275.

[5] A. Benaini and D. Laiymani (1994) Generalized WZ Factorization on a Reconfigurable Machine. *J. of Parallel Algorithms and Applications*, 3, 1, p. 261-269.

[6] A. Benaini and D. Laiymani (1994) Parallel Block Generalized WZ Factorization on a Reconfigurable Machine. *Parallel and Distributed Systems, IC-PADS'94*, IEEE CS.

[7] A. Benaini and D. Laiymani (1994) A Multi-phase Gossip Procedure : Application to Matrices Factorization. *Software for Multiprocessors and Supercomputers*, p. 426-434.

[8] A. Benaini (1995) The WW^T factorization of dense and sparse matrices. *Inern. J. of Computer Maths*, 56, 1, p. 219-229.

[9] B. Lang (1999) Exploiting the symmetry in the parallelization of the Jacobi method. *Parallel Computing*, 25, 7, p. 845-860.

[10] G.J. Davis and G.A. Geist (1990) Finding Eigenvalues and Eigenvectors of Unsymmetric Matrices Using a Distributed Memory Multiprocessor. *Parallel Computing*, 13, p. 199-209.

[11] F. Desprez (1994) Procedures de Base pour le Calcul Scientifique sur Machines Paralleles a Memoire Distribuee. *PhD* Institut National Polytechnique de Grenoble.

[12] J.J. Dongarra and A. Robert (1992) Reduction to Condensed Form for the Eigenvalue Problem on Distributed Memory Architectures. *Parallel Computing*, 18, p. 973-982.

[13] W. Mackens and H. Voss (1999) General masters in parallel condensation of eigenvalue problems. *Parallel Computing*, 25 (7) p. 893-903.

[14] K. Grigg and S. Miguet and Y. Robert (1990) Symmetric Matrix-vector Product on a Ring of Processors. *Information Processing Letter*, 35, p. 239-248.

[15] S. Lo and B. Philippe and A. Sameh (1987) A Multiprocessor Algorithm for the Symmetric Tridiagonal Eigenvalue Problem. *SIAM J. Sc. Statist. Comput.* 8, p. 155-165.

[16] S. Miguet and Y. Robert (1992) Reduction Operations on a Distributed Memory Machine with a Reconfigurable Interconnection Network. *IEEE TPDS*, 3, 4, p. 500-505.

[17] J.J. Modi (1990) Parallel Algorithms and Matrix Computation *Clarendon Press - Oxford.*

[18] R. Schreiber (1986) Solving Eigenvalue and Singular Value Problems on an Undersized Systolic Array. *SIAM J. Sc. Statist. Comput*, 7, p. 441-462.

[19] L. Snyder (1982) Introduction to the Configurable, Highly Parallel Computer. *JPDC*, 15, p. 47-56.

DD-Mod: A library for distributed programming

Jesús M. Milán-Franco
Facultad de Informática. Universidad Complutense de Madrid.
Madrid 28040. Spain.
Phone: (+34) 913944350. Fax: (+34) 913944654.
E-mail: milanjm@sip.ucm.es
AND
Ricardo Jiménez-Peris and Marta Patiño-Martínez
Facultad de Informática. Universidad Politécnica de Madrid.
Boadilla del Monte. Madrid 28060. Spain.
Phone: (+34) 913367452. Fax: (+34) 913367412.
E-mail: (rjimenez,mpatino)@fi.upm.es

Keywords: distributed programming, concurrent programming, Modula-2.

Edited by: Rudi Murn
Received: May 14, 1999 **Revised:** April 22, 2000 **Accepted:** July 12, 2000

DD-Mod is a Modula-2 library for teaching concurrent and distributed programming. The use of this library, together with Modula-2, instead of the traditional socket interface and C, makes it possible to propose programming projects in a distributed programming course. Teaching tools based on high-level languages, such as DD-Mod (based on Modula-2), allow students to focus on the main topics of the course. In this way, it is avoided the waste of time with details related to the low-level interface of C and Unix.

1 Introduction

Concurrent programming courses are very common nowadays in undergraduate curricula, and not just a topic on operating system courses. There are also some experiences in distributed systems courses, but they are mainly included as a laboratory on the referred operating system courses, where students practice with the Unix socket library. Here, an extension of Modula-2 suitable for a concurrent and distributed programming course in undergraduate curricula is proposed.

In our University, Modula-2 [16] is taught as the first programming language and it is used in CS1 and CS2 courses. An extension of it, CC-Modula [13, 15], developed in our department, is used in a concurrent programming course. Finally, as part of an operating systems course, the students experiment with the Unix socket library.

Our goal is to provide a library suitable for a distributed programming course (placed between the concurrent programming and the operating systems courses). In this course, students have to face unreliable communication problems, timeouts, etc. High-level languages used for concurrent programming courses, like SR [1] or CC-Modula, are not suitable because they only provide reliable communication. On the other hand, the Unix socket library is too low level for our purposes, because of its unfriendly interface.

The best way to obtain programming skills in this area is writing network applications using a standard high-level language, which must be well known by the students. This approach implies that they do not need to learn a new programming language and they are free from the details and peculiarities of Unix system calls. DD-Mod [14] has been developed as an extension of Modula-2 because it is the language known by our students, so they can concentrate on the topics of the course and not on other details. Other libraries like XDP [2] do not suit our necessities because they imply learning a new language, or are oriented to a different kind of course. For instance, XDP is oriented towards a workstation-programming course.

The rest of the paper is structured as follows: Section 2 shows our system model; Section 3 describes the different primitives provided by DD-Mod. In Section 4, we compare our approach with other appeared in literature and Section 5 describes our experiences with this and other approaches; Section 6 gives the source code for a complete example using DD-Mod. We finally present our conclusions in Section 7.

2 System Model

DD-Mod system model is composed of several hosts connected by a communication network. All the hosts of the system run concurrently the same application program, but not all of them have the same processes running.

A distinguished host, referred to as the *main host*, is in charge of distributing the available resources among all the hosts of the running system. The rest of the hosts are called

remote hosts. The main host is identified at the beginning of the application with the `MainHost` primitive.

In order to run concurrent processes in remote hosts, on every host of the current system there is a special process called *local manager of remote processes (LMRP)*. The LMRP is the only process authorized to create and launch concurrent processes on his local host when remotely invoked.

The LMRP works as follows (Fig. 1): whenever a concurrent block is executed at a host[1] (by a parent process), it is necessary to launch the different concurrent processes that are inside the block on their corresponding destination hosts. If the destination host of a process is the master host, then the process is created as a local child process of the parent process (processes A, B and D in 1).

However, when the destination host is a remote one, the parent process sends a message with the process name to the LMRP at the destination host. The LMPR will create a concurrent process as his local child at the request (processes C and E in 1). The communication between the parent process and the LMRP is done using a special communication channel (the *inter-host channel*). An inter-host channel for every pair of hosts is automatically created at the beginning of the application in the current system.

After having launched all the processes of the concurrent block, the master host suspends the execution of the parent process until the end of all the child processes.

Whenever a local process ends its execution, it directly informs the parent process. On the other hand, when a remote process ends, it informs his local LMRP (his parent process), who in turn sends a message to the parent process at the master host informing of this event.

When all the processes (local and remote) of the concurrent block have ended, the parent process at the master host resumes its execution.

3 The DD-Mod library

As we have previously said, DD-Mod is an extension of Modula-2 implemented as a library, so to use it in a program, the student just has to import the library as he does with any Modula-2 library. It is also necessary to provide a configuration file called `alias.tab`. The library provides the necessary primitives for concurrent and distributed execution of processes and their communication and synchronization across the network.

A DD-Mod distributed program is very simple: it only consists of an executable program and a configuration file. The executable program runs in all the hosts specified in the configuration file; these hosts will be the execution system.

Processes can be started at any host of the system, just indicating the host name where we want to execute it. Two kinds of schemas for host naming are provided: a logical name schema and a physical name schema. The use of the physical name schema implies to give the real name of the host on every call, and we have to modify the code and recompile it if we want to change the configuration of the execution system.

On the other hand, the use of the logical name schema allows program reconfiguration without the need of recompiling it as the program use logical names provided by the configuration file `alias.tab`. To reconfigure the system we only have to change the alias assignment from physical to logical hosts in the configuration file.

If we are using a logical name schema, the primitive `ReadAlias` must be called to read the current configuration file. Then, whenever a physical host name is needed, it is obtained via the primitive `Alias` that translates a logical name to the corresponding physical name.

After reading the configuration file, if needed, the program has to create two tables at every host, one with the hosts where the program is going to run and the other one with the processes that can run concurrently. All the hosts that are not included in the host table, or in the process table are ignored during execution. The host table is created calling the `InitHost` primitive for each host of the execution system. The process table is created calling the `InitProc` primitive with the name and local address of the process.

The Fig. 1 shows a system with two hosts and the two tables mentioned before, the process and host tables.

The services provided by the library are comprised of two groups of primitives: process management and communication primitives (that can be plain or selective). In the following, we describe both sets of primitives.

3.1 Process management primitives

```
BEGIN
   ...
   CoBegin;
      StartPro-
cess (Proc_name_1, Host_1);
      ...
      StartPro-
cess (Proc_name_N, Host_M);
   CoEnd;
   ...
END Main_program.
```

Figure 2: DD-Mod process creation

Process execution model is based on Dijkstra's cobegin construction [9]. The `CoBegin` primitive starts the concurrent execution of processes and the `CoEnd` primitive synchronizes the flow of the parent process with the end of all the child processes and resumes the parent process. Processes are dynamically created at the chosen host calling the `StartProcess` primitive, this can only be done between a `CoBegin` and a `CoEnd` call. The `StartPro-`

[1]This host is called the *master host*, as it will direct the execution of the processes of the block.

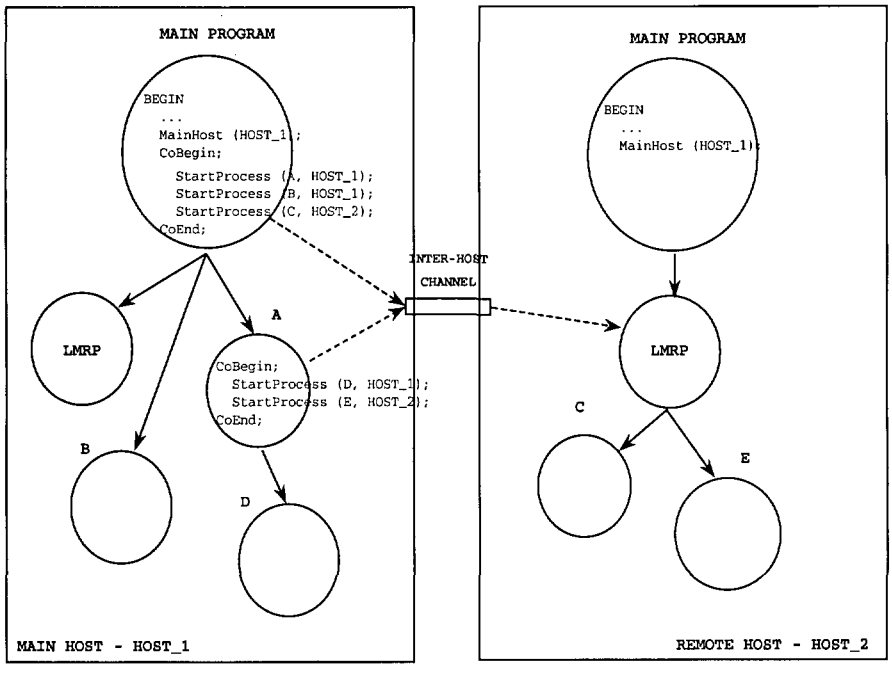


Figure 1: Process creation using LMRP

cess primitive takes two parameters: the name of the concurrent process and a host where to run the process. Fig. 2 shows the DD-Mod code needed for process creation.

DD-Mod processes are structured as parameterless procedures identified by their names, which are kept in the process table maintained on every host along with their local address.

3.2 Process interaction primitives

As we are dealing with distributed systems, we cannot use shared memory for inter-process communication purposes, so processes in our system interact by means of message passing. The messages are sent via communication channels, which are declared as variables of the abstract data type `ChannelType`. Channels always have a proprietary process, which is the only one that can receive messages from it. On the other hand, any process can use a channel to send messages to the owner process (i.e., DD-Mod provides N:1 communication).

Channels are created using the `InitChannel` primitive, this primitive takes as parameters the channel variable, a communication port and the destination host. Any process willing to send or receive messages via a channel must create a channel. In addition, a process willing to receive through a channel must call `GetChannel` with the channel variable in order to get the property of the channel.

Messages are sent using the `Send` primitive naming the channel used. DD-Mod provides two kinds of reception mechanisms: a plain reception mechanism and selective

```
BEGIN (* Sender *)
   ...
   InitChannel (chan-
nel, port, destination_host);
   ...
   Send (channel, message);
   ...
END Sender;

BEGIN (* Receiver *)
   ...
   InitChannel (chan-
nel, port, destination_host);
   GetChannel (channel);
   ...
   Receive (channel, buffer[, timeout]);
   ...
   ReleaseChannel (channel);
   ...
END Receiver;
```

Figure 3: DD-Mod communication schema

reception mechanism[2]. The plain reception is used to receive a message from a particular channel by calling the `Receive` primitive providing the channel and a buffer where to put the data received. Once the channels are no more needed they are destroyed with the primitive `ReleaseChannel`, and so their resources are released and

[2]In both cases, the receiver must have the property of the channels before receiving messages.

used in other channels. The DD-Mod code for channel creation and plain send–receive is shown in Fig. 3.

Selective reception, on the other hand, allows waiting messages on a set of channels for a given condition (this schema is explained in Section 3.3). Both communication schemas allow including a timeout.

As it can be seen, the interface used for communication and synchronization through the network is easy to use and friendlier than that of Unix sockets. Fig. 4 shows a snapshot of a running program. The figure shows the creation of local and remote processes[3] and the use of the plain communication schema where the receiver processes use their own channels.

3.3 Selective reception

The DD-Mod selective reception schema is similar to the CSP guarded command [11] or the CC-Modula `select` statement [15].

To use selective reception, the first thing to do is to build the list of channels and the guarded conditions to be used. The list is provided as an abstract data type (`GuardList-Type`) and it is initialized with the primitive `Create-GuardList` providing a guard list. Each branch of the selective reception (a pair of Boolean conditions and associated channel) is inserted in the list using the primitive `InsertGuard` indicating the list[4], the condition and the reception channel. When a list is no more needed, it is destroyed with the `DestroyGuardList` primitive.

Once the guard list is built, a call to the `Select` primitive is made. This primitive will block the process until a message is received in one of the channels whose condition is true, or the timeout is reached (if a timeout is included). The `Select` primitive returns the position in the list of the chosen branch to complete the reception of the pending message calling `ReceiveGuardChannel` with the channel and a buffer for the message. If the timeout is reached, then the `Select` primitive returns `MAXCARD` and a default action can be taken.

The `Select` primitive combined with the `CASE` statement can be used to build *Dijkstra's alternative construct* [10] and using this structure inside a `LOOP` block we get *Dijkstra's repetitive construct*. Fig. 5 shows the code for the repetitive construct using the selective reception and the `CASE` statement inside a `LOOP` block.

Although selective reception is more complex than plain reception, it is still simpler to use and less prone to errors than the socket interface. For instance, the channel list is an abstract data type and we have shown that building complex constructions using the DD-Mod primitives is very easy. What is more important, the creation and destruction of communication channels is dynamic and this improves system resources' utilization.

[3]This is done sending messages to the LMRP through the inter-host channel.

[4]This is necessary because it is possible to have several lists at the same time.

4 Comparison with other approaches

Other approaches for including distributed courses in undergraduate curricula have been presented in literature, but they do not fulfill our necessities.

The library XDP presented in [2] is oriented for a workstation programming course, rather than a "general" distributed course and faces the communication services using a low-level approach. DD-Mod, on the other side, is oriented towards general distributed courses and provides a high-level interface for communication services.

In [6] a package, ST-Threads, is introduced for its use in a concurrent programming course. The package is a simplification of the interface provided by the operating system. It does not cover unreliable communication services, which is one of the targets of our approach; and it forces students to face the peculiarities of system calls.

The Ben-Ari Concurrent Interpreter, or BACI, is the topic of [7] for concurrent programming course. BACI is based on the approach presented in [3]. As in DD-Mod, the *CoBegin–CoEnd* block is introduced to delimit concurrent blocks. As in [6], unreliable communication and distributed execution are not covered. Moreover, it implies the learning of a new language, which is one of the things that we wanted to avoid by using DD-Mod. Recently, a distributed version of BACI has appeared, Distributed BACI, which extends BACI to allow the development of distributed algorithms by adding communication primitives (send, receive and broadcast). However, it does not support selective reception as in DD-Mod.

Ben-Ari, itself, has developed a software package, DPLab [5], for supporting the teaching of distributed programming. It is an extension of Pascal with primitives and constructs for distributed programming. The package has a proper environment and compiler, this limits its capabilities as it does not support pointers, and the communication schema used is broadcast. Our approach extends Modula-2 with concurrent and distributed primitives and constructs, but does not limit the standard capabilities of the language and it provides selective reception.

An integrated course on parallel and distributed processing is presented in [8]. This course does not only include distributed systems, but also transactions and parallel systems. Many theoretical topics are included and the PVM system is used as a tool for practical sessions. Lectures cover a wide range of knowledge and not just distributed systems, which are the aim of our approach. The labs imply the learning of the PVM programming interface and the use of C as programming language. We have avoided this by extending the language the students know (Modula-2) with a library suitable for distributed programming.

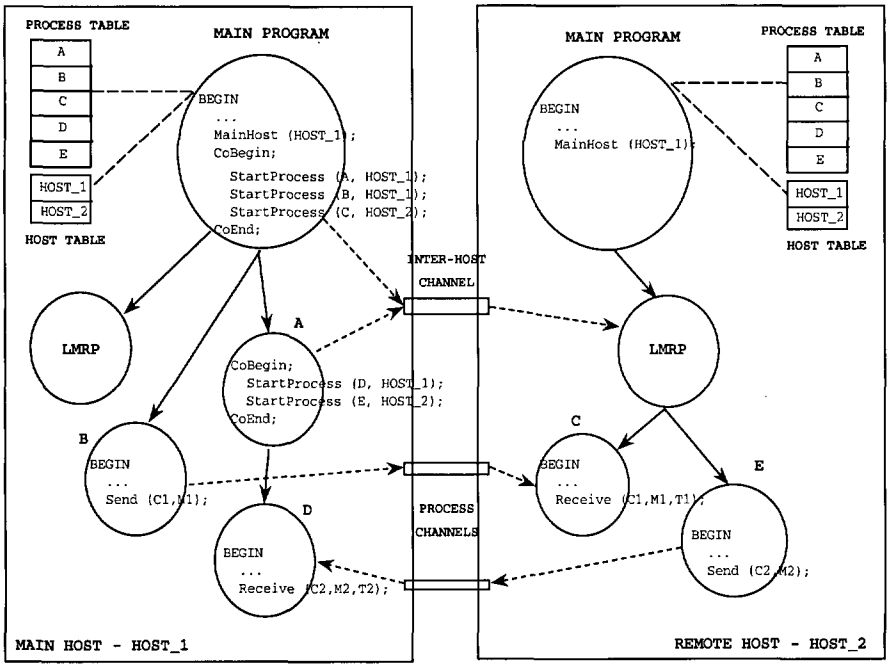


Figure 4: Snapshot of a running system

5 Experimental evaluation

Before choosing DD-Mod as a language for teaching distributed programming, we used different approaches, some of them similar to the ones described in the previous section, and evaluated the results we got.

The first thing to take into consideration is that students already know the used programming language, which avoids the need of teaching a new language for the course and the time required to obtain the necessary skill with this language. Second, by using a high level library, students do not have to face the special characteristics of the operating system calls and can concentrate on the topics of the course.

The main advantage obtained is the gain in time. It is not necessary to waste time (around one month) in teaching the language and getting the appropriate skill with it. Therefore, it is possible from the beginning to put in practice the topics of the course.

6 An example using DD-Mod

As a practical example of distributed programming using DD-Mod, we provide the code for a solution of the *Bounded Buffer* problem (see Fig. 7 - 13).

The problem is a variation of the producer-consumer problem. Two processes share a common (bounded) buffer for communication purposes. The producer process puts data items on the buffer and the consumer gets the stored data items from the buffer to consume them.

The bounded buffer is constructed as an array with two indices: one index indicates the next free slot on the array and the other indicates the slot containing the next object to be removed. As the array is of a finite size, the indices must around the array, i.e., the buffer is a *circular buffer*.

The producer-consumer problem arises when the producer tries to put a new item in the buffer when there are no free slots on it or the consumer wants to get an item from the buffer, when is empty.

The proposed solution uses three processes (Fig. 6): one for the producer, another one for the consumer, and other one to manage the *(circular)* bounded buffer. Hence, the bounded buffer is an active process and not just a passive abstract data type. As this is a distributed solution, we have placed every process at different (logical) hosts, as seen in Fig. 12.

The explanation of the example follows. The producer (Fig. 7) runs an infinite loop that is always creating new data items and putting them in the buffer (i.e., it sends the items to the buffer process, which stores them in the buffer). The consumer (Fig. 8) runs also an infinite loop that retrieves data items from the buffer (asks the buffer process for items who sends them to the consumer), to consume them.

The Bounded Buffer process (Fig. 9) also runs an infinite loop that receives items from the producer to store them in the buffer and requests from the consumer to send it the buffer items. Therefore, the buffer process has two functions:

1. If there is free space in the buffer, it has to allow the reception of data items from the producer.

```
VAR
   list : GuardListType;
   ...
BEGIN
   ...
   LOOP
      CreateGuardList(list);
      InsertGuard(list, channel_0, condition_0);
      ...
      InsertGuard(list, channel_N, condition_N);
      CASE Select(list, timeout) OF
         0: ReceiveGuardChan-
nel(channel_0, buffer_0);
            ... Action 1 ...|
         ...
         N: ReceiveGuardChan-
nel(channel_N, buffer_N);
            ... Action N ...
      ELSE
         ... Timeout action ...
      END; (* CASE *)
      DestroyGuardList(list);
   END; (* LOOP *)
   ...
END Selective_process;
```

Figure 5: DD-Mod selective communication (repetitive construct)

2. If there are data items in the buffer, it must send these items to the consumer when this is ready to receive them (i.e., it sends request to the buffer asking for items).

We provide these two functions by using the selective reception schema in a repetitive construct. The running of the selective reception in the bounded buffer process is as follows:

- If there is free space in the buffer, the reception of new data items from the producer is allowed.
- If there are data items in the buffer, the consumer is allowed to take them.
- If there is no free space in the buffer, the only admissible operation is the sending of data items to the consumer.
- If the buffer is empty, it is only possible to receive items from the producer.

Using selective reception, the accidental destruction of un-consumed items and deadlocks is prevented. The bounded buffer process cannot receive items from the producer if there is no free space, thus the items are safely stored until they are consumed. In addition, if the buffer is empty, it is not possible to receive requests from the consumer.

The main program is divided into three blocks for clarity:

Initialization block, Fig. 11, where the configuration file is read to make it possible the use of logical names instead of physical ones to designate the hosts. Afterwards, we create the host and the process tables and all the communication channels, this way they are available for all the processes. The initialization ends with the definition of the main host.

Concurrent execution block. (Fig. 12) This is the main block of the distributed program and only the main host executes it. It has the first "CoBegin–CoEnd" block and here the concurrent (or distributed) process execution begins using the `StartProcess` primitive naming the host where the process is going to be executed.

Termination block. (Fig. 13) As the concurrent execution block, only the main host executes this block. It is used to terminate the execution of the remote LMRP using the `StopHost` primitive, and so could end the execution of the program.

7 Conclusions

We have presented a library of intermediate level suitable for distributed programs courses using Modula-2. The use of this library helps the students to concentrate on the aspects of building distributed programs, as it is just an extension of Modula-2 the language they know.

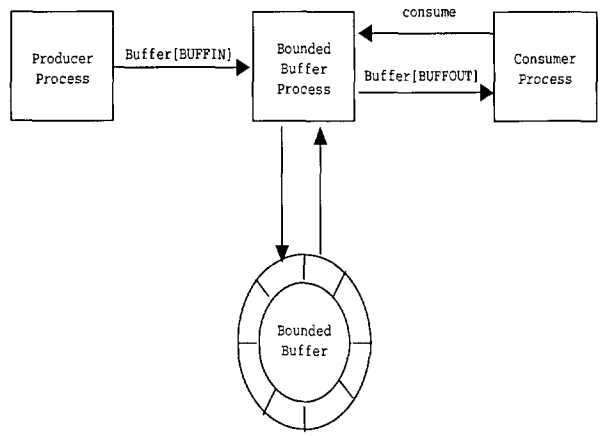


Figure 6: Bounded Buffer processes schema

```
PROCEDURE Producer;

(* The producer process is an infinite loop that creates    *)
(* data items and sends them to the bounded buffer process. *)

VAR
   item : CARDINAL;

BEGIN
   LOOP
      (* PRODUCE A DATA ITEM *)
      Send (buffer [BUFFIN], item);
   END; (* LOOP *)
END Producer;
```

Figure 7: Bounded Buffer problem: Producer processes code

The library provides a clean and typed interface, and its services include the process management, and the sending and reception of messages, either selective or plain. The hardest aspects of communication, like channel or guard list declaration, are solved providing abstract data types.

Other proposals presented in literature do not suit our necessities because they imply the learning of new languages, the use of low-level interfaces, or even C and Unix services.

References

[1] G.R. Andrews. The Distributed Programming Language SR: Mechanisms, design and implementation. Software: Practice and Experience. Vol. 12, no. 8, 1982.

[2] D.M. Arnow. XDP: A Simple Library for Teaching a Distributed Programming Module. In 26th SIGCSE Technical Symposium on Computer Science Education, pp. 82–86. Conference abstracts. Nashville, Tennessee, 1995.

[3] M. Ben-Ari. Principles of Concurrent Programming. Prentice-Hall International, 1982.

[4] M. Ben-Ari. Principles of Concurrent and Distributed Programming. Prentice-Hall International, 1990.

[5] M. Ben-Ari and S. Silverman. DPLab: An Environment for Distributed Programming. In 4^{rd} ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education. Krakow, Poland, 1999.

[6] T.S. Berk. A Simple Student Environment for Lightweight Process Concurrent Programming under SunOS. In ACM SIGSCE '96 2/96, pp. 165–169. Philadelphia, PA, 1996.

[7] B. Bynum and T. Camp. After You, Alfonse: A Mutual Exclusion Toolkit. In ACM SIGSCE '96 2/96, pp. 170–174. Philadelphia, PA, 1996.

[8] J.C. Cuhna and J. Lourenço. An Integrated Course on Parallel and Distributed Processing. In ACM SIGSCE 98, pp. 217–221. Atlanta, GA, 1998.

```
PROCEDURE Consumer;

(* The consumer process is an infinite loops that, when *)
(* ready, asks the buffer for items to consume.         *)

VAR
   item : CARDINAL;

BEGIN
   (* The process becomes proprietary of his channel: *)
   (* buffer [BUFFOUT] channel.                       *)
   GetChannel (buffer [BUFFOUT]);
   LOOP
      Send (consume, item);
      Receive (buffer [BUFFOUT], item);
      (* CONSUME THE DATA ITEM *)
   END; (* LOOP *)
   ReleaseChannel (buffer [BUFFOUT]);
END Consumer;
```

Figure 8: Bounded Buffer problem: Consumer processes code

[9] E.W. Dijkstra. Cooperating Sequential Processes. In F. Genuys (Ed.), Programming Languages. Academic Press, New York, 1968.

[10] E.W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Communications of the ACM, vol. 18, no. 8, pp. 453–457, August, 1975.

[11] C.A.R. Hoare. Communicating Sequential Processes. Communications of the ACM. Vol. 21, no. 8, pp. 666–677, August, 1978.

[12] G. Hommel. Language Constructs for Distributed Programs. In Distributed Systems: Methods and Tools for Specification. An Advanced Course. Lecture Notes in Computer Science, vol. 190, pp. 287–341, 1985.

[13] R. Jiménez. CC-Modula: An Environment for Concurrent Programming. Technical Report. Facultad de Informática, Universidad Politécnica de Madrid, 1990.

[14] J.M. Milán. A Kernel for a Distributed Language in a Unix Environment. Technical Report. Facultad de Informática, Universidad Politécnica de Madrid, 1994.

[15] R. Morales and J.J. Moreno. CC-Modula: A Modula-2 Tool to Teach Concurrent Programming. ACM SIGCSE Bulletin, vol. 21, no. 3, pp. 19–25, 1989.

[16] N. Wirth. Programming in Modula-2. Springer Verlag, New York, 1982.

```
PROCEDURE Bounded_Buffer;

(*  The bounded buffer is implemented using a repetitive    *)
(*  selective communication schema with two branches:        *)
(*  1. Receive data items from the producer and store them   *)
(*     in the buffer when there is free space.               *)
(*  2. Send items, if there are any stored in the buffer,    *)
(*     to the consumer when it request them.                 *)

VAR
   item  : ARRAY [0..MAXITEM-1] OF CARDINAL;
   head, tail, size : CARDINAL [0..MAXITEM-1];
   msg   : CARDINAL;
   list  : GuardListType;

BEGIN
   (* Initially the buffer is empty. *)
   FOR head := 0 TO MAXITEM - 1 DO
      item [head] := NONE
   END; (* FOR *)
   head := NONE;
   tail := NONE;
   size := NONE;
   (* The process becomes proprietary of his channels: *)
   (* buffer [BUFFIN] and consume.                      *)
   GetChannel (buffer [BUFFIN]);
   GetChannel (consume);
   LOOP
      CreateGuardList (list);
      InsertGuard (list, buffer [BUFFIN], size < MAXITEM - 1);
      InsertGuard (list, consume, size > NONE);
      CASE Select (list) OF
         0 : ReceiveGuardChannel (buffer [BUF-
FIN], item [tail]);
             tail := (tail + 1) MOD MAXITEM;
             INC (size); |
         1 : ReceiveGuardChannel (consume, msg);
             Send (buffer [BUFFOUT], item [head]);
             item [head] := NONE;
             head := (head + 1) MOD MAXITEM;
             DEC (size);
      END; (* CASE *)
      DestroyGuardList (list)
   END; (* LOOP *)
   ReleaseChannel (buffer [BUFFIN]);
   ReleaseChannel (consume);
END Bounded_Buffer;
```

Figure 9: Bounded Buffer problem: Bounded Buffer process code

```
MODULE Bounded_Buffer_Problem;

FROM kerneldd IMPORT (* The DD-Mod primitives are imported *)
   ReadAlias, Alias, InitHost, InitProc, Main-
Host, StopHost, CoBegin,
   StartProcess, CoEnd, ChannelType, InitChannel, GetChannel,
   Receive, ReleaseChannel, GuardListType, CreateGuardList,
   InsertGuard, Select, ReceiveGuardChannel, DestroyGuardList;

CONST (* Declaration of the different constants used     *)
   MAXITEM     = 5; (* Max number of items in the buffer *)
   NONE        = 0;
   BUFFIN      = 0;
   BUFFOUT     = 1;
   BUFFINPORT  = 5400; (* BUFFINPORT, BUFFOUTPORT and CONSUME-
PORT *)
   BUFFOUTPORT = 5411; (* are the addresses of the ports used by  *)
   CONSUMEPORT = 5420; (* the communication chan-
nels.             *)

VAR (* Declaration of communications channels and other vari-
ables *)
   buffer  : ARRAY [BUFFIN..BUFFOUT] OF ChannelType;
   consume : ChannelType;
   name    : ARRAY [0..20] OF CHAR;
```

Figure 10: Bounded Buffer problem: Declaration code

```
BEGIN (* MAIN PROGRAM *)

   (* INITIALIZATION BLOCK:                                    *)

   (* The configuration file is read to use logical names    *)
   (* instead the physical ones, the Host table is built     *)
   (* (notice the use of the primitive Alias to get the      *)
   (* physical name from the logical ones). The Process      *)
   (* table is built with three entries: Bounded_Buffer,     *)
   (* Producer and Consumer. All the communication channels  *)
   (* are created and the main host ('alpha') is declared.   *)

   ReadAlias;
   Alias ('alpha', name);
   InitHost (name);
   Alias ('beta', name);
   InitHost (name);
   Alias ('gamma', name);
   InitHost (name);
   InitProc ('Bounded_Buffer', Bounded_Buffer);
   InitProc ('Producer', Producer);
   InitProc ('Consumer', Consumer);
   Alias ('alpha', name);
   InitChannel (buffer [BUFFIN], BUFFINPORT, name);
   Alias ('gamma', name);
   InitChannel (buffer [BUFFOUT], BUFFOUTPORT, name);
   Alias ('alpha', name);
   InitChannel (consume, CONSUMEPORT, name);
   Alias ('alpha', name);
   MainHost (name);
```

Figure 11: Bounded Buffer problem: Main program, initialization block

```
   (* CONCURRENT EXECUTION BLOCK:                                    *)

   (* The distributed execution (CoBegin-CoEnd block) begins *)
   (* launching the processes in the desired destination     *)
   (* host: Bounded_Buffer at 'alpha', Producer at 'beta'    *)
   (* and Consumer at 'gamma'. Afterwards, the main host     *)
   (* waits the end of the concurrent processes (CoEnd       *)
   (* sentence).                                             *)

   CoBegin;
      Alias ('alpha', name);
      StartProcess ('Bounded_Buffer', name);
      Alias ('beta', name);
      StartProcess ('Producer', name);
      Alias ('gamma', name);
      StartProcess ('Consumer', name);
   CoEnd;
```

Figure 12: Bounded Buffer problem: Main program, concurrent execution block

```
   (* TERMINATION BLOCK:                                             *)

   (* When the main host finishes the Concur-
rent Block, it has *)
   (* to tell the remote hosts to end their LMRP daemons and *)
   (* finish the program. The LMPR daemon of the main host   *)
   (* dies automatically when the main host process ends.    *)

   Alias ('beta', name);
   StopHost (name);
   Alias ('gamma', name);
   StopHost (name)
END Bounded_Buffer_Problem.
```

Figure 13: Bounded Buffer problem: Main program, termination block

A Technique for Computing Watermarks from Digital Images

Chin-Chen Chang and Chwei-Shyong Tsai
Department of Computer Science and Information Engineering,
National Chung Cheng University, Chaiyi, Taiwan 621, R. O. C.
Email:{ ccc, tsaics}@cs.ccu.edu.tw

Keywords: Digital watermark, intellectual property protection, vector quantization

Edited by: Rudi Murn
Received: September 10, 1999 **Revised:** March 22, 2000 **Accepted:** April 4, 2000

The new watermarking scheme in this paper solves the problem of digital copyright protection in computer network society. The proposed scheme coalesces both the technique of vector quantization (VQ) and the technique of principal component analysis (PCA). Therefore, it can effectively embed and extract watermarks. The proposed scheme is an invisible, robust, secure, multiple and blind watermarking technique. After various attacks such as JPEG lossy compression, blurring, cropping, rotating, and sharpening, the proposed scheme still provides robust watermarks.

1 Introduction

With the development of digital multimedia data, a large quantity of information may exchange faster and faster through Internet. People can easily download other's data within shorter time. Thus more and more digital multimedia are illegally distributed. This leads to more tension in intellectual property protection. With the easy access to copying, modifying and forging, the fake copies not only cause the author's economical loss, but also invite legal disputes because people may clone the image and modify it for illegal use. Therefore, researchers are working hard to protect the image authentication. Digital watermarking technique is the most common method to solve this dilemma in intellectual property protection. With recent published research results [1-4,7-11,14-16], digital images have promised a brighter future with the use of digital watermarking technique.

Basically, digital watermarking techniques include watermark embedding process and watermark extracting process. First, the watermark embedding process can embed copyright information like an owner's logo or label into an original image and produces watermarked image when published. One may also use secret keys to enhance the security. Second, the watermark extracting process is used to extract the embedded watermark from the watermarked image. It helps people to recognize the ownership and to prevent plagiarism. Besides, the authentication center can be the judge whenever the dispute of ownership occurs [15,16]. This requires that the owners register their watermarked images and watermarks at the authentication center. Then the authentication center can have the secret key from the owner to distinguish the copyright through the watermark extracting process.

To protect the intellectual property in digital images, an effective digital watermarking technique must contain the five characteristics as follows:

1. Invisibility - The image shouldn't be different after being watermarked.
2. Security- When the watermark extracting process reveals the watermark for legal owner, the embedded watermark shouldn't be removed by the attacker without the secret keys and parameters.
3. Robustness -When the watermarked image goes through the image processing to enhance the image quality, or if the watermarked image has been destroyed on purpose, but its PSNR is still above 30, the watermark should be recognizable when extracted. Of course, the recovered watermark may perceptually have been lossy.
4. Blindness- The watermark can be recovered without the original image. This prevents the use of additional spaces.
5. Multiple Watermarking - It allows several cooperators to embed their respective watermarks into an image, simultaneously.

In this paper, the proposed scheme fulfills all the requirements in digital watermarking. That is the coalition of vector quantization (VQ), which is often used in image compression, and principal component analysis (PCA), which is used popularly in pattern recognition. This coalition technique can embed and extract watermarks.

In the proposed scheme, the original image is a gray-scale image and the watermark is a binary image. Every watermark pixel is possible to be 0 or 1. Before embedding the watermark of binary image into an original image, it is necessary to choose a VQ codebook and sort the codebook with the technique of PCA. Each watermark bit will

randomly match with one block in the original image. After searching the codebook, which finds the most similar codeword for the block and returns the codebook index of the codeword, the watermark bit and index may cooperate to produce a matched index for a watermark table. When all the bits in binary image are processed, the watermark is embedded. On the other hand, one may recover the watermark when using codebook and index to match the block. The experimental results have shown that the correct rate of the recovered watermark's bit exceeds 86 % after the digital images go through the process of JPEG lossy compression, blurring, rotating, cropping, and sharpening.

This paper is organized as below. Section 2 will introduce the technique of VQ and PCA. Section 3 will explain the proposed scheme of coalition technique (VQ and PCA). Section 4 shows the experimental results and discussions. Section 5 will be the conclusions.

2 Relates Works

2.1 Vector Quantization (VQ)

VQ is a technique of lossy image compression. It mainly compresses images with vectors. About original image OI, VQ includes vector encoding phase and vector decoding phase. First, OI represents as a set of vectors $O_1, O_2, \ldots, O_m$. Let each Oi's dimensionality be $V, i = 1, 2, \ldots, m$. Then it is necessary to find a proper codebook CB for OI. CB is composed of codewords. Let $CB = C_1, C_2, \ldots, C_n$. C_j is one of the vectors and its dimensionality is also $V, j = 1, 2, \ldots, n$.

When the vector encoding phase is managing every O_i, it finds a codeword C_k and makes minimum distortion between O_i and C_k. One may call that C_k the most similar codeword of O_i. It is possible to measure the distortion between O_i and O_k with Equation (1). Equation (1) is used to compute the squared Euclidean distances between two vectors.

$$d(O_i, C_k) = |O_i - C_k|^2 = \sum_{j=1}^{V} (O_{ij} - C_{kj})^2 \quad (1)$$

,where O_{ij} and C_{kj} are the j-th components of O_i and C_k, respectively.

In the following, VQ will represent O_i by index k. After all the O_i's are processed, OI is encoded as an index table. In vector decoding phase, each encoded index in the index table is used to retrieve the corresponding codeword from CB according to the index value itself. The retrieved codeword is utilized to restore a vector of the original image. When all the encoded indices have been processed, the most similar image with the original one is restored. VQ can effectively compress and decompress images by the search of codebook. Today, many published research results [5,6] may make the search of codebook faster. The way that the proposed scheme uses VQ technique will be shown in the following. One may choose a proper codebook for the original image. In watermark embedding process, every watermark bit will match with a random block in original image and searches for the codeword, which is the most similar to the block. If the watermark bit is 1, the index of the most similar codeword will be stored in the watermark table. On the contrary, the index of a dissimilar codeword will be stored in the watermark table when the watermark bit is 0. To reach a standard of coherence and reasonableness, PCA is utilized to sort the codebook beforehand. PCA softens the dilemma when a dissimilar codeword is stored in the watermark table. We will detail the procedure in next section. In the following subsection, the technique of PCA will be described.

2.2 Principal Component Analysis (PCA)

PCA [12] is a quite popular dimensionality reduction technique in Pattern Recognition area. Along the direction of maximum variance, PCA projects the data into a linear subspace with a minimum loss of information. In other words, all the projection points obtained from subspace still keep the characteristics of the original information.

For the given n information $R_1, R_2, \ldots, R_n$, every R_i is represented by an n-dimension vector. PCA can find a direction D, $D = (d_1, d_2, \ldots, d_r)$ such that $\sum_{i=1}^{r} d_i^2 = 1$. This makes the projection points keep their largest difference between each other after being projected to D as n information. The following steps will be exemplified to find D.

Algorithm: [PCA][12]

1. Normalize all $R_i, i = 1, 2, \ldots, n$. This produces the corresponding $R_i', i = 1, 2, \ldots, n$.
2. With normalized $R_i', i = 1, 2, \ldots, n$, calculate the covariance matrix M, which has dimension $n \times n$.
3. Find all eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$, if $\lambda_i \geq \lambda_{i+1}, i = 1, 2, \ldots, n-1$. And let $D_1, D_2, \ldots, D_n$ be the matched eigenvectors of $\lambda_1, \lambda_2, \ldots, \lambda_n$, respectively, where $|D_i| = 1$.
4. Let $D = D_1$ (called the first principal component direction). $D = D_2$ is called the second principal component direction, ... $D = D_n$ is called the n-th principal component direction.

With the use of PCA, the first principal component direction can keep the largest difference in data. This technique makes the similar points closer after projection, and makes the dissimilar points farther after projection. This distinct characteristic makes it possible to sort data of multidimension. The way to project data to D is to multiply the data by D.

Using the technique of PCA can sort all the codewords in the codebook. PCA finds the first principal component

direction and then projects all codewords to D. According to the order of the projection points, the matched codewords of the corresponding projection points compose a new codebook. This is the so-called sorted codebook.

3 The Proposed Scheme

The proposed scheme can satisfy all current requirements in image watermarking. The watermark embedding process can hide the watermark without modifying the original image. In initial process, it must have a codebook to represent the original image. This can be done with Linde-Buzo-Gray (LBG) [13] algorithm often seen in VQ. For images partition, and for fixed size rectangle blocks, let each of them be composed of $k \times k$ pixels. Then the process executes LBG algorithm, so that all the blocks can be trained to produce some image blocks, which can be typified. These blocks are named "codewords". When $n = k \times k$, all the codewords can compose a codebook CB. One may regard every codeword as an n-dimension vector. Finally, the process utilizes PCA to find first principal component direction D of codewords in CB, and projects every codeword to D. All the projection points on D are corresponded to their matched codewords. These codewords are operated in order and stored in another codebook CB'. The CB' is named after "sorted version of CB". Let CB have m codewords. In the proposed scheme, the original image is a gray-scale image O with $N_1 \times N_2$ pixels. Digital watermark W, which represents copyright information, is a binary image with $w \times h$ pixels. Every pixel is possible to be 0 or 1. When the process of embedding watermark or extracting watermark occurs, the codewords in CB' are used to obtain the relative information about W. The next subsection will explore more methods definitely.

3.1 Embedding Watermark

Before embedding every watermark pixel $WP_i, i = 1, 2, \ldots, w \times h$, the process chooses a matched block OB_i with a size of $k \times k$ pixels from the original image O. Using the secret key S as the seed, pseudo random number generator PRNG will generate $w \times h$ random integer pairs $(x_i, y_i), i = 1, 2, \ldots, w \times h$. This makes $1 \leq x_i \leq N_1 - k$, and $1 \leq y_i \leq N_2 - k$. The coordinate of (x_i, y_i) is also OB_i's coordinate of pixel at left upper corner. Generally, all coordinates of OB_i are $(x_i + a, y_i + b)$'s, where $a = 0, 1, \ldots, k - 1$ and $b = 0, 1, \ldots, k - 1$. Note that OB_i and OB_j can be partially overlapped, for $i \neq j, 1 \leq i, j \leq w \times h$. After obtaining OB_i, the process searches CB' for the codeword C, the most similar to OB_i, and then retrieves the index idx of C in CB'. Embedding WP_i and storing the matched index information depend on whether WP_i is 0 or 1. When WP_i is 1, the matched index idx will be stored in the i-th entry of the watermark table WT; in this process, the index of the codeword most similar to OB_i will be stored. On the other hand, when WP_i is 0, let

(a) (b)

Figure 1: (a)Original image of "Lena" (512 × 512), (b)Watermark of "National Chung Cheng University" (64× 64)

(a) (b)

Figure 2: (a)Reconstruction of JPEG compression of "Lena", (b)Recovered watermark from Figure 2(a)

(a) (b)

Figure 3: (a)Blurred image of "Lena", (b)Recovered watermark from Figure 3(a)

(a) (b)

Figure 4: (a)Rotated image of "Lena", (b)Recovered watermark from Figure 4(a)

(a) (b)

Figure 5: (a)Cropped image of "Lena", (b)Recovered watermark from Figure 5(a)

(a) (b)

Figure 6: (a)Sharpened image of "Lena", (b)Recovered watermark from Figure 6(a)

$\alpha = idx + \frac{\lfloor m \rfloor}{2}$, the process stores $(idx + \frac{\lfloor m \rfloor}{2})$ in WT if $\alpha < m$; otherwise, the process stores $(idx - \frac{\lfloor m \rfloor}{2})$ in WT. This process stores the index of the more dissimilar codeword to OB_i. For example, suppose that CB' has 256 16-dimension codewords, so that $m = 256, n = 16, k = 4$. Suppose $WP_i = 1$. When $idx = 200$, the process stores 200 in WT. When $WP_i = 0$, the process will store $(200 - \frac{\lfloor 256 \rfloor}{2}) = 72$ in WT. When all the WP_i's are processed, the watermark image W is embedded. Finally, the owner registers the codebook CB', the watermark image W, and the watermark table WT at the authentication center. The secret key S is kept secretly by the owner until the dispute of copyright happens.

Algorithm: [Embedding Watermark]

Input: Original image O with $N_1 \times N_2$ pixels, watermark image W with $w \times h$ pixels and codebook CB'.

Output: Secret key S and watermark table WT with size $w \times h$.

1. Randomly choose an integer S as the secret key of O.
2. Use S as the seed of PRNG to generate $w \times h$ random integer pairs (x_i, y_i) such that $1 \leq x_i \leq N_1 - k, 1 \leq y_i \leq N_2 - k$ and $1 \leq i \leq w \times h$.
3. For each watermark pixel WP_i, map it to a corresponding $k \times k$ block OB_i of O and OB_i use (x_i, y_i) as the position of the left upper corner.

Figure 7: Original image of "F14" (512 × 512)

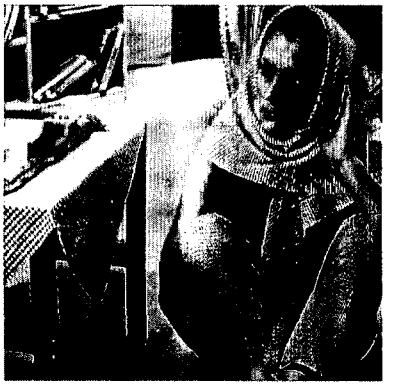


Figure 8: Original image of "Barbara" (512 × 512)

4. Search codebook CB' for OB_i to find the most similar codeword C, and let the index of C be idx.
5. Check WP_i. For $WP_i = 1$, store idx to $WT(i)$ else store $(\frac{\lfloor m \rfloor}{2} + idx)$ to $WT(i)$, if $(idx + \frac{\lfloor m \rfloor}{2}) < m$; otherwise, store $(idx - \frac{\lfloor m \rfloor}{2})$ to $WT(i)$.
6. Repeat from **Step 3** to **Step 5** until all WP_i's are processed.

Finally, the watermark is embedded.

3.2 Extracting Watermark

When one claims himself to be the legitimate owner, the secret key S' must be presented to the authentication center. The authentication center uses S' to execute watermark extracting process. At that time, $S'$is the seed of PRNG, and $w \times h$ random integer pairs $(x_i, y_i), 1 \leq i \leq w \times h$ are generated through PRNG. Then every (x_i, y_i) corresponds to a $k \times k$ block OB_i in O. The coordinate of (x_i, y_i) is at OB_i's left upper corner. Because of this, the coordinates of $k \times k$ pixels of OB_i are $(x_i + a, y_i + b)'$s, for $a = 0, 1, \ldots, k-1$ and $b = 0, 1, \ldots, k-1$. Through the search in CB for OB_i, the most similar codeword C obtains the index $midx$ of C. Furthermore, $midx$ and the corresponding i-th entry in WT cooperatively determine the recovered watermark pixel. Setting up a threshold T determines the pixel to be 1 or 0 after restoration. When $|WT(i) - midx| < T$, the recovered watermark pixel WP_i should be 1; otherwise, $WP_i = 0$. After operating all OB_i's in order, the process can generate the recovered watermark W' and take out the registered watermark W. Therefore, the authentication center may compare these two versions to determine whose copyright it is. In the following, extracting watermark algorithm will be explained.

Algorithm: [Extracting Watermark]
Input: Secret key S', original image O with $N_1 \times N_2$ pixels, watermark image W with $w \times h$ pixels, codebook CB', watermark table WT with size $w \times h$, and threshold T.
Output: "Legal owner" or "Illegal owner".

1. Use secret key S' as seed of PRNG to generate $w \times h$ random integer pairs $(x_i, y_i), 1 \leq x_i \leq N_1 - k, 1 \leq y_i \leq N_2 - k$ and $1 \leq i \leq w \times h$.
2. Map each (x_i, y_i) to a corresponding $k \times k$ block OB_i of O by using (x_i, y_i) to be the position of OB_i of the left upper corner, $1 \leq i \leq w \times h$.
3. Search CB' to find the most similar codeword C for OB_i and let the index of C be $midx$.
4. Retrieve the i-th entry $WT(i)$ from WT.
5. If $|midx - WT(i)| < T$ then let recovered pixel $WP_i = 1$ else let recovered pixel WP_i=0.
6. Repeat from **Step 2** to **Step 5** to generate the recovered Watermark W'.
7. If $W = W'$ then return ("Legal owner") else return ("Illegal owner").

4 Experimental Results

In this paper, the proposed scheme can satisfy the current requirements of robust watermarking. Some experiments have been done by attacking the image first and extracting the watermark later. The image attacks include JPEG lossy compression (compression rate 14:1), blurring (a 5×5 neighborhood median filter), rotating (one degree in clockwise direction), cropping (1/4 of the image), and sharpening. These experiments are executed by Photoshop, which is issued by Adobe. The experimental results are very positive. One can still extract the watermark after the image attacks. About the given original image gray-scale "Lena" with 512×512 pixels (shown as Figure 1(a)), LBG algorithm trains a codebook CB for "Lena." CB contains 256 codewords indexed by $0, 1, \ldots, 255$. Every codeword is a 16-dimenion vector; i.e. let $m = 256, n = 16, k = 4$. PCA will help find the first principal direction D of all the codewords and project all the codewords to D. According to the projection points' order, the codewords are stored as a sorted codebook CB'. Afterwards, the embedding watermark algorithm may embed 64×64 binary watermark W (shown as Figure 1(b)) to "Lena." Choosing a secret key may be thought as seed of PRNG, which is utilized to generate 64×64 random integer pairs (x_i, y_i). Every coordinate of (x_i, y_i) in "Lena" represents a 4×4 block at left upper corner of OB_i. Therefore, 64×64 blocks OB_i in "Lena" are obtained. When embedding watermark pixel WP_i, the process searches for the most similar codeword and the index idx with OB_i in CB'. For $WP_i = 1$, idx is stored in WT. For $WP_i = 0$, the process stores $(128+idx)$ in WT if $128 + idx < 256$; otherwise, stores $(idx - 128)$ in WT. After all the WP_i's have been processed, the work of W embedding is done.

Table 1: The bit correct rates of extracted watermark of different image under various attacks

Image	JPEG	Blurring	Rotating	Cropping	Sharpening
Lena	99.95%	99.95%	93.65%	86.40%	99.78%
F14	100.00%	99.90%	95.53%	95.60%	99.41%
Barbara	99.91%	99.65%	92.55%	93.04%	98.24%

When there is a need to identify the watermark, the authentication center must obtain secret key from the owner to execute the process of extracting watermark. By using the proposed extracting watermark algorithm, the process extracts watermark. In the experiments, the threshold is set to be 40, and secret key S' functions the same as the one in embedding watermark, and this helps obtain 64×64 of 4×4 block in OB_i. From OB_i to codebook, the process searches for the most similar codeword with OB_i, and obtains index $midx$, which is the codeword's index. Both the i-th entry $WT(i)$ in WT and $midx$ may cooperate to determine the restoration of WP_i . If $| WT(i) - midx | < T$, the recovered $WP_i = 1$; otherwise, the recovered $WP_i = 0$. After all OB_i's have been processed, the embedded watermark is recovered.

After the original image "Lena" has been through different kinds of attacks, one may still extract the watermark. The experiments have attacked the images with JPEG compression (shown as Figure 2(a)), blurring (shown as Figure 3(a)), rotating (shown as Figure 4(a)), cropping (shown as Figure 5(a)), and sharpening (shown as Figure 6(a)). The results are shown in Figures 2(b), 3(b), 4(b), 5(b), and 6(b), respectively.

Another two original images, which are 512 × 512 pixels gray-scale "F14" (shown as Figure 7) and "Barbara" (shown as Figure 8), have repeated the same experiments, respectively. Table 1 represents Bit Correct Rate (BCR) after each image attack. These results are gained through the proportionality of correct recovered watermark pixels and the original watermark image pixels. The experimental results have shown that the BCR exceeds 86% in each image. That is, the watermark can still be clearly recovered after various attacks.

The experimental results have explained that the proposed scheme is robust without modifying original image. The security depends on the secret key. With embedding multiple watermarks, different watermarks will only need their distinct secret keys for security. Besides, every watermark pixel is independent in embedding and extracting. Therefore, parallel process may be used to accelerate the pace.

5 Conclusions

In this paper, the proposed technique of digital image watermarking can satisfy the current requirements of the watermarking technique,which is invisible, secure, robust,multiple and blind.

The proposed scheme can effectively embed watermark without modifying the original image. With multiple watermarks, only different secret keys are needed. Because of the distinct characteristics, multiple watermarks can be embedded and each of them can be extracted independently. The proposed scheme is truly robust under various attacks. The experimental results have also supported the proposed scheme that BCR exceeds 86% in each attack.

References

[1] W. Bender, D. Gruhl, N. Morimoto, and A. Lu, "Technique for data hiding," *IBM System Journal*, vol. 35, no. 3,pp. 313–336, 1996.

[2] A. G. Bors and I. Pitas, "Image watermarking using DCT domain constraints," *Proceedings of 1996 IEEE International Conference on Image Processing (ICIP'96)*, vol. 3, pp. 231–234, 1996.

[3] C.C. Chang, T.S. Chen, and P.F. Chung, "A technique for copyright of digital images based upon (t, n)-threshold scheme," *to appear in Informatica.*

[4] T.S. Chen, C.C. Chang, and M.S. Hwang, "A virtual image cryptosystem based on vector quantization," *IEEE Transactions on Image processing*, vol. 7, no. 10, pp. 1485–1488, 1998.

[5] T.S. Chen and C.C. Chang, "A new image coding algorithm using variable-rate side-match finite-state vector quantization," *IEEE Transactions on Image processing*, vol. 6, no. 8, pp. 1185–1188, 1997.

[6] T.S. Chen and C.C. Chang, "Diagonal Axes Method (DAM): A fast search algorithm for vector quantization," *IEEE Transactions on Circuits and System for Video Technology*, vol. 7, no. 3, pp. 555–559, 1997.

[7] I. J. Cox, J. Kilian, F. T. Leighton, and T. Shamoon, "Secure spread spectrum watermarking for multimedia," *IEEE Transactions on Image Processing*, vol. 6, no. 12, pp. 1673–1687, 1997.

[8] I. J. Cox, and J.P.M.G. Linnartz, "Some general methods for tampering with watermarks," *IEEE Journal on Selected Areas in Communication*, vol. 16, no. 4, pp. 587–593, 1998.

[9] S. Craver, N. Memon, B. L. Yeo, and M. M. Yeung, "Resolving rightful ownerships with invisible watermarking techniques: limitations, attacks, and implications," *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 4, pp. 573–586, 1998.

[10] C.T. Hsu and J.L. Wu, "Hidden digital watermarks in images," *IEEE Transcations on Image Processing*, vol. 8, no. 1, pp. 58–68,1999.

[11] M. Kutter, F. Jordan, and F. Bossen, "Digital watermarking of color images using amplitude modulation," *Journal of Electronic Imaging*, vol. 7, no. 2, pp. 326–332, 1998.

[12] R.C.T. Lee, Y.H. Chin, and S.C. Chang, "Application of principal component analysis to multikey searching," *IEEE Transactions on Software Engineering*, vol. SE-2, No. 3, pp. 185–193, 1976.

[13] Y. Linde, A. Buzo, and R. M. Gray, "An algorithm for vector quantization design," *IEEE Transactions on Communications*, Vol. 28,pp. 84–95, 1980.

[14] M. D. Swanson, M. Kobayashi, and A. H. Tewfik, "Multimedia data-embedding and watermarking technologies," *Proceedings of IEEE*, vol. 86, no. 6, pp. 1064–1087, 1998.

[15] G. Voyatzis and I. Pitas, "Protecting digital-image copyrights: A framework," *IEEE Computer Graphics and Applications*, vol. 19, no. 1, pp. 18–24, 1999.

[16] R.B. Wolfgang and E.J. Delp, "A watermark for digital image," *Proceedings of the 1996 International Conference on Image Processing, Lausanne, Switzerland*, vol. 3,pp. 219–222, 1996.

A Program–Algebraic Approach to Eliminating Common Subexpressions

James M. Boyle
Technology Development Division
Argonne National Laboratory
Argonne, IL 60439, USA
boyle@td.anl.gov
AND
R. Daniel Resler
Dept. of Mathematical Sciences
Virginia Commonwealth University
Richmond, VA 23284–2014, USA
dresler@vcu.edu

Keywords: programming languages, optimizing

Edited by: Rudi Murn
Received: August 17, 1998 **Revised:** July 15, 1999 **Accepted:** May 18, 2000

An operation often performed by optimizing compilers for higher–level languages is common–subexpression elimination. Traditionally, common–subexpression elimination is performed on a directed, acyclic graph representing the expression or program. This paper shows how common–subexpression elimination can be expressed algebraically, using a "program algebra" incorporating the syntax of typical higher–level language expressions plus λ–expressions from the λ calculus and functional programming. This approach has two major advantages—it is intuitive and easy to understand and it uses transformations for which correctness-preservation is easy to prove.

1 Introduction and Motivation

An operation often performed by optimizing compilers for higher–level languages is *common–subexpression elimination*. In common–subexpression elimination, an arithmetic or other expression that appears to be computed more than once (and that would produce the same value each time) is computed just once and assigned to a temporary variable; this temporary variable then replaces the multiple occurrences of the expression. The goal of common–subexpression elimination, of course, is to reduce execution time; not only is the number of arithmetic operations reduced, but also the number of memory fetches to obtain operand values are reduced. Finally, in many cases the value of the common subexpression can be held in a hardware register, further reducing computation time.

Although at first thought it may seem unlikely that a programmer would write out a non-trivial expression more than once, he or she may do so for reasons of clarity. Moreover, a significant number of common subexpressions arise *implicitly*; for example, from repeated occurrences of a subscripted variable, such as `a[i,j]`, whose address computation requires the evaluation of an addressing polynomial.

Traditionally, common–subexpression elimination is performed on a directed, acyclic graph (DAG) representing the expression or program [1, Section 9.8]. The purpose of this paper is to show how common–subexpression elimination can be expressed algebraically, using a "program algebra" incorporating the syntax of typical higher–level language expressions plus λ–expressions (see Section 2) from the λ calculus and functional programming.

In contrast to the more usual approach to common–subexpression elimination based on DAGs, the algebraic approach has a number of advantages:

- It is based on familiar programming and mathematical notation, which facilitates human description and understanding of the process of common–subexpression elimination.
- The manipulations necessary to eliminate common subexpressions can be expressed in the high–level notation of program transformations, which can be applied by TAMPR [3], a fully automatic rewrite–rule based program transformation system.
- Because the meaning of a program transformation depends on the meaning of the programming language in which the expressions are written (in this case, augmented with λ–expressions), it is easy to prove that the program transformations preserve the correctness of the programs being transformed.

These advantages of the algebraic approach also apply to solving other types of problem, see [3, 5, 4].

It is the latter advantage that provides one of the principal motivations for the work described here; the development

of the common–subexpression elimination transformations is a step in the development of a *trusted compiler*—one that can be proven to correctly compile any correct program it receives [6].

Common–subexpression elimination arises in designing the transformations for the register–allocation phase of such a trusted compiler. We have found that, with the right notation, common–subexpression elimination can be performed as a natural, and almost incidental, step in register allocation.

1.1 Problem Statement

We examine the problem of identifying common subexpressions in arithmetic expressions of the sort found in typical programming languages such as Fortran or C.

To expose the essence of the common–subexpression elimination problem, we make three simplifying assumptions that eliminate unnecessary detail from the presentation without materially affecting the generality and applicability of the techniques we discuss:

1. We assume that the expressions considered do not involve side–effects on the state of the computation. (If the language defines the meaning of expressions that have side effects, transformations can be written to introduce temporary variables and break up such expressions into a sequence of assignments that captures the order of operations involved and that ensures that the side–effects behave according to the requirements of the language semantics.)

2. Further, we assume that all variables involved in the expressions are simple variables, without subscripts or structure qualifiers. (It is trivial to expand our approach to handle the commoning of expressions appearing in subscripts and the evaluation of address polynomials; there is no point in complicating the discussion by including them. We do remark, however, that the use of subscripted variables can lead to syntactic aliasing (`a[i]` is a syntactic alias for `a[j]` if `i = j`), which increases the difficulty of detecting common subexpressions, but has no effect on the correctness of commoning for those common subexpressions that are detected.)

3. Finally, we attempt to common only syntacticly identical subexpressions; we do not consider commutative and associative variants, nor variants clouded by aliasing. (Finding all possible common subexpressions based on algebraic identities for commutativity and associativity is a complex problem [1, Section 9.10]. Moreover, such identities do not hold unconditionally in program algebra; for example, even the integer expression $i - j + k$ may not be equivalent to the expression $i + k - j$ in computer arithmetic because, when i, j, and k are positive integers, the second may overflow when the first does not.) We reconsider this topic briefly in Section 5.

2 Notation and Motivation for the Algebraic Approach

The algebraic approach to common–subexpression elimination relies, both conceptually and notationally, on algebraic manipulations involving λ–expressions. In the usual mathematical notation, a typical λ–expression is $\lambda x \,.\, f(x) \,.\, e$. In this expression x is called the λ–variable, $f(x)$ the λ–body, and e the λ–argument. The λ–expression without the argument is called a λ–abstraction, which is a (nameless) function; like a function, a λ–abstraction can be *applied* to an argument (see [2], p. 6). The λ–expression gives a name (x) to the value of its argument; this name can be used to represent that value throughout the expression that is the λ–body. To perform automated program transformation using TAMPR, an ASCII representation of λ–expressions is required; for the preceding expression, the corresponding TAMPR notation is `lambda x @ f(x) end (e)` .

The basis for the algebraic approach is the observation that λ–expressions can be used conveniently to model the hardware operations required to evaluate programming-language expressions. For example, if one wishes to generate near–optimal assembly code for a RISC computer, application of a λ–abstraction can be used to represent the command to load the value of a variable into a register. In this case, the λ–variable represents a hardware register, and the argument of the λ–application is the variable whose value is to be loaded:

```
lambda r0 @                r0 := a
    ...                    ...
end (a)
```

Similarly, a RISC arithmetic operation (which computes a value and deposits it into a register) can be represented by a λ–application whose argument is a binary operation:

```
lambda r2 @                r2 := r0 + r1
    ...                    ...
end (r0 + r1)
```

Combining these two representations, the RISC evaluation of the expression

$$a + b$$

can be represented by

```
lambda r0 @                r0 := a
    lambda r1 @            r1 := b
        lambda r1 @        r1 := r0 + r1
            r1
        end (r0 + r1)
    end (b)
end (a)
```

For most hardware architectures, it is advantageous to avoid unnecessary reloads of registers from memory in

generating code for expressions. In the algebraic notation it is natural to perform this optimization by expanding the scope of the λ–abstraction representing the load of the value of a program variable to include all references to that variable (assuming sufficient hardware registers are available). Thus, for the expression

$$a + a$$

we wish to obtain

```
lambda r0 @                    r0 := a
    lambda r0 @                r0 := r0 + r0
        r0
    end (r0 + r0)
end (a) ;
```

which requires one memory access, rather than

```
lambda r0 @                    r0 := a
    lambda r1 @                r1 := a
        lambda r1 @            r1 := r0 + r1
            r1
        end (r0 + r1)
    end (a)
end (a)
```

which requires two.

Common subexpression elimination is simple to implement in the algebraic approach because only a slight generalization of the algebraic manipulations that minimize memory fetches for program variables is required to minimize the re–computation of non–trivial expressions.

3 An Algebraic Approach to Eliminating Common Subexpressions

Taking advantage of the correspondence between λ–expressions and RISC instructions, our objective is to transform any arithmetic expression into a λ–expression in a canonical form that is the precursor of near–optimal code for a RISC machine. As discussed in the preceding section, each λ–expression in our final canonical form, which we call a "λ–nest", can then be transliterated into a machine instruction in the target assembly–language program. More specifically, each λ–expression represents a load of a register from memory, an arithmetic operation, or a store of a register value to memory.

Consider the expression

$$(b + (a - c)) * (a - c) \qquad (1)$$

Our goal is to transform this expression into the λ–expression shown in Figure 1(a); In this form, new temporaries (λ–variables) have been introduced for each variable reference and arithmetic operation in the expression, and common subexpressions have been eliminated. Such an expression greatly simplifies register allocation—careful allocation of registers to each λ–variable and assignment of λ–arguments to them results in the three–address machine instructions shown in Figure 1(b).

The creation of this canonical form, from which RISC code can easily be generated, depends on transforming an expression through a sequence of intermediate canonical forms, each of which captures a part of the compilation process. (The role of canonical forms in program transformation is also discussed in [4].) The following sections discuss the transformations and intermediate canonical forms necessary for elimination of common subexpressions in complicated arithmetic expressions.

3.1 Introducing Identity λ–expressions into Expressions

For common subexpressions to be eliminated, the value of each program variable, constant, and result of an operation in an expression must be associated with a temporary variable name that represents that value. The following definition simplifies discussion of the introduction of these names.

Definition 1 *Given an expression meeting the restrictions discussed in Section 1.1, the* set of subexpressions *of that expression consists of the expression itself, plus all operands of operators in that expression.*

A consequence of this definition is that each variable or constant in an expression is considered a subexpression, as is the entire expression.

A simple and correctness-preserving way to associate temporary variable names with subexpressions is to replace each subexpression of the original arithmetic expression by an *identity* λ–abstraction applied to that subexpression as argument. These applied identity λ–abstractions have the form

```
lambda tempID @
   tempID
end ( <expression> )
```

As discussed later, transformation is simplified if each introduced identity λ–expression has a unique λ–variable name (`tempID`).

The syntactic properties of the expression in this form, which we call *canonical–λ–form–1*, can be expressed more formally by the BNF (Backus-Naur Form) rules:

canonical–λ–form–1 → *identity–λ–op–expression*
identity–λ–op–expression →
simple–identity–λ–expression
| lambda <ident> @
<ident>
end (*identity–λ–op–expression*
<op>
identity–λ–op–expression)

```
lambda a_2 @
  lambda c_3 @
    lambda t_4 @
      lambda b_1 @
        lambda t_5 @
          lambda t_9 @
            t_9
          end ( t_5 * t_4 )
        end ( b_1 + t_4 )
      end ( b )
    end ( a_2 - c_3 )
  end ( c )
end ( a )
```

(a)

```
r0 := a
r1 := c
r1 := r0 - r1
r0 := b
r0 := r0 + r1
r1 := r0 * r1
```

(b)

Figure 1: Final forms for Expression (1)

simple–identity–λ–expression →

```
lambda <ident> @
  <ident>
end ( <ident> )
| lambda <ident> @
  <ident>
  end ( <const> )
```

with the restriction that the identifier in the body of an *identity–λ–op–expression* or *simple–identity–λ–expression* be the same as the identifier of the λ–variable of that λ–expression. (See Appendix A for a listing of portions of the TAMPR subject–language grammar relevant to these examples.)

The TAMPR transformation list that converts Fortran or C expressions to this first canonical form is shown in Figure 2. TAMPR applies transformations to the *parse tree* of a program or expression, constructed according to the grammar in Appendix A. This parse tree consists of *nodes* labeled with the appropriate terminal and non-terminal symbols from the grammar. Each transformation consists of a *pattern*—the part between `.sd.` and arrow (==>)—and a *replacement*—the part between the arrow and `.sc.` TAMPR applies the transformations by visiting each node in the parse tree for the expression in post–order (bottom–up, left–to–right), attempting to match each transformation in a transformation list in turn at each node. When the pattern of one of the transformations matches the part of an expression at that node, the matching transformation is said to *apply*, and its replacement describes how to assemble a syntactically legal expression parse tree to substitute for the matched non-terminal.

The first three of these transformations describe how to convert a Fortran or C binary–expression, identifier, or constant, respectively, into an identity λ–expression. The fourth transformation describes removal of unneeded parentheses from the resulting expression. (Parentheses may have been necessary in the original expression to obtain the required order of operation. Once λ–expressions have been introduced, the order of operations has been fixed and the parentheses are no longer needed; removing them simplifies later transformations.) Again, the productions from the TAMPR subject–language grammar needed to understand these transformations are shown in Appendix A.

Consider the first transformation in the list in Figure 2. The pattern matches any subexpression that involves a binary operator, represented by `<op>`, in an `<op expr>`. (Syntactically, what we refer to informally as expressions are called `<op expr>`s in the grammar of Appendix A.) The replacement of this transformation assembles an identity λ–expression defining a new temporary λ–variable (`<var>"1"`); this λ–expression has the original subexpression (`<op expr>"1"`) as its argument. (The `.generate.` clause of the transformation asks TAMPR to create a new identifier by suffixing the identifier `t` with an integer to make the new name unique.)

The pattern of the second transformation matches any identifier in a `<primary>`, which represents either operand in an `<op expr>`. Its replacement assembles an identity λ–expression as in the first transformation, with the Fortran or C identifier (`<ident>"1"`) as its argument. The third transformation behaves similarly for a constant.

Figure 3 shows the results of applying these transformations to Expression (1). This form of the expressions conforms to the BNF grammar for *canonical–λ–form–1*.

Of course, one usually intends that a program transformation produce a result that is not only syntactically legal but also semantically valid. That is, one wants to show that the transformation is *correctness–preserving*—that the replacement subexpression generated by the application of the transformation is a *refinement* [7] of the subexpression matched by the pattern of the transformation. (One fragment of a program, p_2, is a refinement of another, p_1, if they produce the same result, are both undefined, or if p_1 is undefined and p_2 is defined.) In the case of the transformations that introduce identity λ–expressions, the proof that they preserve correctness is based on an algebraic law for

```
.transform 1. {
<op expr> {
.sd.
 <op expr>"1" {<op expr> <op> <primary>}
.generate. <var>"1" .like. <var> { t }
==> .stop.
 lambda <var>"1" @
   <var>"1"
 end ( <op expr>"1" )
.sc.
}

<primary> {
.sd.
 <ident>"1"
.generate. <ident>"2" .like. <ident>"1"
==> .stop.
 lambda <ident>"2" @
   <ident>"2"
 end ( <ident>"1" )
.sc.

.sd.
 <const>"1"
.generate. <var>"1" .like. <var> { const }
==> .stop.
 lambda <var>"1" @
   <var>"1"
 end ( <const>"1" )
.sc.

.sd.
 (<lambda abstrac-
tion"1" <pending args>"1")
==>
 <lambda abstraction"1" <pending args>"1"
.sc.
}
}
```

Figure 2: Transformations for the first canonical form

identity λ–expressions from the λ calculus:

$$\lambda x.x.(e) \equiv e$$

Normally, TAMPR applies transformations *to exhaustion*; that is, after a transformation applies, the nodes of the *replacement* parse tree are recursively visited with the transformations in another post–order traversal (see Section 3.1). Application terminates when the entire (possibly modified) parse tree has been traversed without any transformation applying. This method of application would lead to infinite re–applications for the transformations in Figure 2, because in each of these transformations, the replacement contains a copy of the symbols matched in the pattern. TAMPR therefore provides a mode for applying transformations that performs a *single* post–order traversal of the parse tree, visiting each node just once; this mode is indicated by `.transform 1.` and the `.stop.` that follows the arrows in the transformations.

```
lambda t_9 @
    t_9
end (
    lambda t_5 @
        t_5
    end (
        lambda b_1 @
            b_1
        end ( b ) +
        lambda t_4 @
            t_4
        end (
            lambda a_2 @
                a_2
            end ( a ) -
            lambda c_3 @
                c_3
            end ( c )
        )
    ) *
    lambda t_8 @
        t_8
    end (
        lambda a_6 @
            a_6
        end ( a ) -
        lambda c_7 @
            c_7
        end ( c )
    )
)
```

Figure 3: Expression (1) in canonical form 1

3.2 Commoning Subexpressions

Once the transformations have caused each subexpression of an expression to become the argument of a λ–abstraction, the next step is to eliminate syntactically–identical common subexpressions. Given that the expression is in the first canonical form, common–subexpression elimination is easy to understand and verify in the algebraic approach.

The fundamental observation is that syntactically identical common subexpressions lead to duplicate λ–expressions. We call one λ–expression a *duplicate* of another if both λ–expressions have syntactically identical arguments.

The general idea of this step is to expand the scope of each λ–abstraction in turn, one step at a time, checking whether the newly expanded scope now includes a du-

plicate λ–expression. If so, each duplicate λ–expression within the scope is removed, taking care to replace instances of the λ–variable of the deleted λ–expression with the variable of the remaining, outermost λ–expression. Or, stated in another way, common subexpressions are eliminated by systematically increasing the scope of one λ–expression "over" another in order to compare and eliminate one of the expressions if they have identical arguments. When all λ–expressions have been examined in this way, any subexpression that appeared more than once in the original expression will appear only once in the final text, the multiple instances of that subexpression having been replaced by multiple instances of the λ–variable to which it is bound.

An examination of the grammar for *canonical–λ–form–1* and Figure 3 reveals that two transformations are required, corresponding to the two operands of a binary operator. These transformations are shown in Figure 4. Each of these transformations matches a λ–expression, `lambda <var>"1"`, having an argument that is an expression with a binary operation, `<op>"2"`; call this matched λ–expression the "outer λ–expression". Each transformation then looks for a λ–expression in one operand of the binary operator and expands its scope, performing common subexpression elimination if necessary.

Transformation 1 matches a λ–expression, `lambda <var>"2"`, the "expandable λ–expression" that appears as the *first* operand in the argument of the outer λ–expression. Transformation 1 expands the scope of the expandable λ–expression to include the outer λ–expression. (The application of transformation 1 to an intermediate form that arises during the transformation of Expression 1 is shown in Figures 5 and 6.) The expansion of the scope of `lambda <var>"2"` brings all λ–expressions in the second operand, `<primary>"2"`, within the scope of the λ–variable `<var>"2"`. A λ–variable name in `<primary>"2"` cannot be the same as ("clash" with) `<var>"2"` when `<primary>"2"` is brought with the scope because there are no λ–expressions in the input program, a unique variable name is generated for each inserted λ–expression, and no λ–expression is replicated by the transformations. (In λ calculus terms, this statement means that "α–conversion" is never required to avoid name clashes.)

The bringing of λ–expressions within the scope of the variable `<var>"2"` provides an opportunity to detect common subexpressions. After the transformation increases the scope of the expandable λ–expression, a *subtransformation* (the `.transform 1. { ... }` following `<primary>"2"` in the replacement of the transformation) is applied to `<primary>"2"`. This subtransformation examines each λ–expression within `<primary>"2"` for an argument identical to `<expr>"3"`, the argument of the expandable λ–expression. If the two arguments are syntactically identical, a common subexpression has been found. The subtransformation in the replacement of transformation 1 eliminates the common

```
.transform *. {
<entity> {
.sd.
 lambda <var>"1" @ <var>"1" end (
   lambda <var>"2" @
    <entity>"2"
   end ( <expr>"3" )
   <op>"2" <primary>"2"
 )
==>
 lambda <var>"2" @
 lambda <var>"1" @ <var>"1" end (
 <entity>"2"
 <op>"2" <primary>"2" .transform 1. {
 <entity> {
 .sd.
   lambda <var>"11" @
    <entity>"11"
   end ( <expr>"3" )
 ==> .stop.
   <entity>"11" .transform 1. {
   <entity> { .sd.
               <var>"11"
              ==> .stop.
               <var>"2"
              .sc. }
   }
 .sc.
 }
 }
 )
 end ( <expr>"3" )
.sc.
.sd.
 lambda <var>"1" @ <var>"1" end (
 <op expr>"2"
 <op>"2" lambda <var>"2" @
          <entity>"2"
         end ( <expr>"3" )
 )
==>
 lambda <var>"2" @
 lambda <var>"1" @ <var>"1" end (
 <op expr>"2" <op>"2" <entity>"2"
 )
 end ( <expr>"3" )
.sc.
}
}
```

Figure 4: Transformations for the second canonical form

subexpression by removing the λ–abstraction to which it is bound in `<primary>"2"` and replacing all occurrences of the corresponding λ–variable (`<var>"11"`) by `<var>"2"`. (The sub–subtransformation, `.transform`

1. { ... } following <entity>"11" in the replacement of the subtransformation, accomplishes this replacement.) The first operand, <entity>"2", need not be examined for duplicate λ–expressions because the transformations in Figure 4 are applied to the original expression "bottom-up, left-to-right", and so any duplicates within <entity>"2" have already been found. If there is no duplicate λ–expression in <primary>"2", <primary>"2" is left unchanged.

As an example of of the application of transformation 1, consider the λ-expression shown in Figure 5, which is

```
lambda t_9 @
   t_9
end (
   lambda a_2 @
      lambda c_3 @
         lambda t_4 @
            lambda t_5 @
               t_5
            end ( b_1 + t_4 )
         end ( a_2 - c_3 )
      end ( c )
   end ( a ) *
   lambda a_6 @
      lambda c_7 @
         lambda t_8 @
            t_8
         end ( a_6 - c_7 )
      end ( c )
   end ( a )
)
```

Figure 5: λ-nest prior to commoning

a fragment from an intermediate form that occurs when transforming Expression (1) from *canonical–λ–form–1* to *canonical–λ–form–2*, which will be formally defined shortly. (One may wonder about the absence of a lambda b_1 expression from this fragment. This λ–expression is present in the complete expression but not in this fragment; the scope of lambda b_1 has already been expanded to include the entire fragment shown in Figures 5 and 6.)

Transformation 1 applies to this fragment, and it expands the scope of the lambda a_2 ... end(a) expression to encompass the outer (lambda t_9) expression. Then the subtransformation searches the nest of λ–expressions in <primary>"2" (beginning with lambda a_6) for any λ-expression with the same argument as lambda a_2. One such (lambda a_6) is found and removed, instances of its λ–variable being replaced with a_2. Figure 6 shows the resulting expression fragment.

Transformation 2 is similar to transformation 1, but simpler. It matches a λ–expression (lambda <var>"2"), the "expandable λ–expression" that appears as the *second* operand of the argument of the outer λ–expression. Transformation 2 expands the scope of the expandable λ–

```
lambda a_2 @
   lambda t_9 @
      t_9
   end (
      lambda c_3 @
         lambda t_4 @
            lambda t_5 @
               t_5
            end ( b_1 + t_4 )
         end ( a_2 - c_3 )
      end ( c ) *
      lambda c_7 @
         lambda t_8 @
            t_8
         end ( a_2 - c_7 )
      end ( c )
   )
end ( a )
```

Figure 6: λ-nest after one application of transformation 1

expression to include the outer λ–expression. Transformation 2 is simpler than transformation 1 in that it need not examine the first operand (<op expr>"2") for commoning, because transformation 1 will have been applied until no more λ–expressions remain in <op expr>"2". Thus, when transformation 2 applies <op expr>"2" is a variable and no commonable λ–expressions can exist within it.

The post–order traversal provided by the TAMPR transformation system causes these transformations to be applied to the parse tree of an expression from the bottom up (smallest subexpressions first). Thus, the scopes of various λ–abstractions will be repeatedly increased until all common subexpressions have been eliminated. The result of common–subexpression elimination, then, is a nest of λ–expressions in which no λ–expression occurs as an argument of another λ–expression. Figure 7 shows Expression (1) after transformations 1 and 2 have removed all

```
lambda b_1 @
   lambda a_2 @
      lambda c_3 @
         lambda t_4 @
            lambda t_5 @
               lambda t_9 @
                  t_9
               end ( t_5 * t_4 )
            end ( b_1 + t_4 )
         end ( a_2 - c_3 )
      end ( c )
   end ( a )
end ( b )
```

Figure 7: Expression (1) after eliminating common subexpressions

common subexpressions, starting from the form of the expression in Figure 3.

The expression in Figure 7 is in the canonical form whose formal description is

canonical–λ–form–2 → `lambda <ident> @`
canonical–λ–form–2
`end ( <expr> )`
| `lambda <ident> @`
`<ident>`
`end ( <expr> )`

Here `<expr>` cannot contain a λ–expression (i.e., it must be an `<ident>`, `<const>`, or a simple binary arithmetic expression involving λ–variables), and the second option must be an identity λ–expression (i.e., both `<ident>`s must be the same).

3.3 Scope reduction

The form of the expression shown in Figure 7 is valid for the generation of code. However, from an aesthetic viewpoint, it is undesirable because the scopes of some λ–abstractions are greater than necessary. For example, the scope of the λ–abstraction binding `b_1` could be reduced to encompass just the λ–abstraction binding `t_5`.

This aesthetically undesirable form has also a practically undesirable consequence: Recall that the goal is to transform an expression into a form that facilitates near–optimal generation of RISC machine instructions. In this form, a λ–variable represents a register. Thus, a λ–abstraction having an unnecessarily large scope unnecessarily ties up a valuable resource, its register, which could be used for evaluating another subexpression.

Optimal use of registers requires delaying register loads until just before the value loaded is needed, which is reflected in the requirement to put the expression of Figure 7 into a canonical form in which each λ–abstraction has minimal scope. That is, the scope of every λ–abstraction should be reduced to encompass just the outermost expression that references its λ–variable. The transformations discussed so far, which eliminate common subexpressions, do not, however, result in such a form.

Achieving the minimal–scope canonical form thus involves applying a third set of transformations that "push" an expression binding a λ–variable, say v_1, into the λ–nest until an expression using v_1 is encountered. Applying such transformations to the commoned nest of Figure 7 results in the more optimal arrangement that was presented in Figure 1(a), which presents our example expression in the final canonical form needed for efficient allocation of registers. The expression shown in this figure is in *canonical–λ–form–3*, which is the same as *canonical–λ–form–2* with the additional restriction that the λ–expressions binding the operands for a binary operation should be as close as possible to the λ–expression for that operation.

4 Proof That All Syntactically Identical Common Subexpressions are Commoned

For any sequence of sets of TAMPR transformations, there are two properties one may wish to prove: that the sets of transformations preserve correctness and that they achieve their goal or goals.

Proofs that the common–subexpression–elimination transformations preserve correctness are not particularly interesting. These proofs follow immediately from simple definitions and theorems in the λ calculus, such as those for identity λ–expressions and distribution properties of λ–expressions. Provided that the meaning of the λ–expression notation in the Poly grammar is the same as the meaning of λ–expressions in the λ calculus, the proofs are trivial. For more information on a methodology for carrying out a formal verification that TAMPR transformations preserve correctness, see [8].

Proof that the transformations discussed here achieve their goal—that they do common all syntactically identical λ–expressions—is more interesting, in part because the proof illustrates the important role of the canonical forms discussed in the preceding sections.

Informally, we wish to prove that the transformations common all syntactically identical subexpressions in an expression. Given that the final form of an expression after the three sets of transformations apply is in *canonical–λ–form–3*, and that every subexpression is an argument of a λ–expression, the non–existence of common subexpressions is equivalent to the following statement:

Theorem 1 *In canonical–λ–form–3, no λ–expression contains (directly or indirectly) in its body another λ–expression whose argument is syntactically identical to the argument of the containing λ–expression.*

The concept of λ–nest discussed informally in preceding sections plays an important role in the proof; its formal definition, stated recursively, is

Definition 2 *The following expressions are* λ–nests*:*

1. *A λ–variable*
2. *A λ–expression whose body is a λ–nest and whose argument is a program variable, a constant, or a binary operation connecting λ–variables.*

(This definition is essentially a restatement of the requirements of *canonical–λ–form–2*.) A consequence of this definition is that no λ–expression in a λ–nest contains in its argument, directly or indirectly, another λ–expression.

The following definitions simplify the statement of the theorem:

Definition 3 *One λ–expression is a* duplicate *of another if their arguments are syntactically identical.*

Definition 4 *A λ–nest is* fully commoned *provided no λ–expression in the nest contains (directly or indirectly) in its body a duplicate λ–expression.*

To prove Theorem 1, it is necessary to use *structural induction* on the subexpressions of the expression being transformed. As stated, Theorem 1 is not strong enough to permit the induction to go through; we must also show that applying the commoning transformations produces a λ–nest:

Theorem 2 *In canonical–λ–form–3, the entire expression is a fully commoned λ–nest.*

Clearly this stronger theorem implies Theorem 1.

Two questions might arise during the proof of this theorem; we discuss them first.

The first question is how, after the initialization transformations have applied, there can be any duplicate non–trivial λ–expressions (λ–expressions with binary–operation arguments) at all; applying the initialization transformations, which insert identity λ–expressions, destroys the syntactical identity of common subexpressions that are binary operations! Because the λ–variable names of the inserted expressions are unique, two instances of a common subexpression such as $a - c$ will differ in the λ–variable names bound to a and c. For example, in Figure 5, the first instance of the common subexpression $a - c$ is `a_2 - c_3` while the second is `a_6 - c_7`.

But observe that there are still some duplicate λ–expressions after application of the initialization transformations: the λ–expressions whose arguments are simple program variables or constants, for example, the arguments `a` of `lambda a_2` and `lambda a_6` in Figure 5.

These simple duplicate λ–expressions serve to "seed" the process of identifying all common subexpressions. The application of the commoning transformations gradually restores the syntactical identity of non–trivial common subexpressions. For example, after one application of transformation 1 to the intermediate stage of commoning shown in Figure 5, the second instance of $a - c$ becomes `a_2 - c_7` (Figure 6). Another application changes it to `a_2 - c_3`, which is now syntactically identical to the first instance. A third application of transformation 1 will then detect that `lambda t_4` and `lambda t_8` are duplicate λ–arguments having the non–trivial expression `a_2 - c_3` as argument.

The second question is what guarantees that the λ–variables in a non–trivial common subexpression (such as `a_2`, `a_6` and `c_3`, `c_7` in Figure 5) are commoned before attempting to common the non–trivial expression itself. The guarantee follows from the fact that the transformations preserve correctness. To be correct, the λ–expression for a non–trivial expression (such as `lambda t_4 @ ... end ( a_2 - c_3 )` in Figure 5) must be within the scopes of the λ–expressions binding any variables it uses (`a_2` and `c_3`). Because transformations 1 and 2 preserve correctness, they preserve this scoping property, guaranteeing that the scopes of the λ–expressions for the variables in a non–trivial argument to a λ–expression are expanded and examined for commoning before the non–trivial expression itself.

Against this background, we can prove Theorem 2.

Proof of Theorem 2:

It suffices to prove the theorem for expressions in *canonical–λ–form–2*, because the conversion to *canonical–λ–form–3* does not involve β–conversion (the substitution of the argument of a λ–expression for all instances of its λ–variable). Thus, the transformation to *canonical–λ–form–3* cannot create any new duplicate λ–expressions; if there are any λ–expressions in *canonical–λ–form–3* having syntactically identical arguments, they must be present in *canonical–λ–form–2*.

To prove the theorem for expressions in *canonical–λ–form–2*, assume that the initialization transformations of Figure 2 have been applied to put the expression in *canonical–λ–form–1*. The proof is based on structural induction on the λ–nests in this form of the expression.

Ground Case: The only λ–nests in an expression in *canonical–λ–form–1* are identity λ–expressions having program variables or constants as arguments (for example, `lambda a_2 @ a_2 end ( a )` and `lambda c_3 @ c_3 end ( c )` in Figure 3). Such a λ–expression is a fully commoned λ–nest, because its body is a λ–variable, and its argument is a program variable or constant. Moreover, because there are no λ–expressions in its body, there can be no duplicate λ–expression.

Inductive Case: At an arbitrary stage in the application of the commoning transformations, the set of transformations is being applied to an outer λ–expression, as discussed in Section 3.2. In the whole expression, this outer λ–expression is an innermost λ–expression that is not part of a λ–nest (for example, `lambda t_9` in Figure 5). By "innermost", we mean that, while the outer λ–expression is not part of a λ–nest, both operands of the binary operator in the argument of the outer λ–expression are λ–nests. By the inductive hypothesis, the λ–expressions in both operands of this binary operator are fully commoned λ–nests, so no λ–expression in either operand contains a duplicate.

We need to show that, after the exhaustive application of transformations 1 and 2 to this outer λ–expression, the λ–nesting and fully commoned properties are established for the resulting λ–expression.

Once application of transformations 1 and 2 to a given outer λ–expression starts, TAMPR applies transformation 1 repeatedly, until it no longer matches. Then, TAMPR applies transformation 2 repeatedly until it no longer matches. At that point exhaustive application to this outer λ–expression is complete, because application of transformation 2 cannot create any new instances to which transformation 1 could apply.

Case 1: Transformation 1 applies. It expands the scope of the outermost λ–expression in the first operand (the expandable λ–expression, `lambda a_2` in Figure 5) to en-

compass the outer λ–expression, and, in particular, the second operand of the binary expression in the argument of the outer λ–expression. The subtransformation in transformation 1 then searches the second operand for a λ–expression that duplicates the expandable λ–expression (`lambda a_6` in Figure 5). If such a λ–expression is found, transformation 1 commons it by removing it and substituting the λ–variable of the expandable λ–expression for all instances of the λ–variable of the duplicate one. (By the inductive hypothesis, there can be at most one commonable λ–expression.) At this point, the subtransformations have restored the fully commoned invariant for the expandable λ–expression: it contains within its body no duplicate λ–expression.

It is necessary to show that application of transformation 1 preserves the fully commoned λ–nest property for the operands of the binary operation of the outer λ–expression after transformation 1 applies (see Figure 6).

Clearly, the application of transformation 1 preserves the λ–nest and fully commoned properties of the remainder of the λ–nest in the first operand of the binary expression that is argument to the outer λ–expression, for this operand is the body of the λ–nest originally at this position, and the body of a fully commoned λ–nest is a fully commoned λ–nest.

The application of transformation 1 also preserves the λ–nest property of the second operand, because either the second operand remains unchanged (no matching duplicate λ–expression was found) or the change consists only of eliminating a λ–expression and substituting one λ–variable for another.

To show that the transformation preserves the fully commoned property for the second operand, it is necessary to show that the substitution of the λ–variable (`a_2`) of the expandable λ–expression for the λ–variable (`a_6`) of the duplicate λ–expression cannot lead to the creation of two or more λ–expressions having syntactically identical arguments. Suppose, by way of contradiction, that the substitution could create two such λ–expressions. Because they were not identical before the substitution (by the inductive hypothesis), one of the expressions must already have contained an instance of the λ–variable of the expandable λ–expression (`a_2`), while the other had the duplicate λ–expression (`lambda a_6`) at that position. But all λ–variable names are unique and the second operand expression was not originally within the scope of the expandable λ–expression, so the second operand cannot contain any instances of the λ–variable of the expandable λ–expression prior to the substitution, contradicting the assumption. Thus, the transformation preserves the fully commoned property of the second operand.

Finally, while the body of the expandable λ–expression (`lambda a_2` in Figure 6) may not yet be a λ–nest because exhaustive application of transformations 1 and 2 has not completed, each application of transformation 1 maintains a property that is needed to show that the final result of the exhaustive application is a λ–nest: the expandable λ–expression, which now surrounds the outer λ–expression, has only a program variable, a constant, or a binary expression connecting λ–variables as its argument. This property follows from the fact that, by the inductive hypothesis, the expandable λ–expression was part of a λ–nest.

Transformation 1 applies repeatedly until it no longer matches. At that point, the first operand of the binary expression that is argument to the outer λ–expression, which was initially a λ–nest, is a λ–variable—the variable of the innermost λ–expression of the original λ–nest (`t_5` in Figure 5).

Case 2: Transformation 1 does not apply; transformation 2 does apply. By the inductive hypothesis, the second operand of the binary expression argument of the outer λ–expression is a fully commoned λ–nest, and this property has been maintained by applications of transformation 1. Transformation 2 increases the scope of the λ–expression in the second operand (the expandable λ–expression). By the inductive hypothesis, only the first operand of the binary operator could contain a λ–expression that could duplicate a λ–expression in the second operand, but the first operand is now a variable and so contains no such λ–expression.

Clearly, the application of transformation 2 preserves the λ–nest and fully commoned properties of both the first and second operands of the binary expression that is argument to the outer λ–expression, because it does not alter the first operand and because the body of the λ–nest in the second operand, which is now the second operand, is a fully commoned λ–nest.

Finally, as in Case 1, each application of transformation 2 maintains the property that is needed to show that the final result of the exhaustive application is a λ–nest: the expandable λ–expression has only a program variable, a constant, or a binary expression connecting λ–variables as its argument.

Transformation 2 applies repeatedly until it no longer matches. At that point, the second operand of the binary expression that is argument to the outer λ–expression, which was initially a λ–nest, is a λ–variable—the variable of the innermost λ–expression of the original λ–nest.

Case 3: Neither transformation 1 nor transformation 2 applies. Then neither argument of the binary expression argument of the outer λ–expression contains a λ–expression. Each operand of the binary operand must be a λ–variable, because either it was originally a λ–variable or it was the innermost variable of a λ–nest that was transformed away. Finally, the entire fragment of the expression produced from this sequences of applications of transformations 1 and 2 is a λ–nest, because each expandable λ–expression has the property required of its argument, and what was originally the outer λ–expression is now the innermost λ–expression. Being an identity λ–expression, its body is a λ–variable (`t_9` in Figures 6 and 7).

Thus, the result of the exhaustive application of transformations 1 and 2 is a λ–nest and all duplicate λ–expressions in the originally outer λ–expression have been commoned.

Q.E.D.

5 Commoning Subexpressions That Are Not Syntactically Identical

One of the advantages of the algebraic approach is that it encourages breaking problems such as commoning subexpressions into a number of smaller, simpler problems. The preceding sections have shown how syntactically identical common subexpressions can be eliminated. With that problem solved, we can consider the problem of eliminating common subexpressions that differ by commutativity and associativity (assuming that the use of commutativity and associativity are acceptable in terms of the correctness required of the program).

Again, we use canonical forms to address this problem. The strategy is first to transform all subexpressions into a canonical form in which subexpressions that originally varied only by the use of commutativity or associativity are syntactically identical; then the transformations discussed in the preceding sections can be used to eliminate the common subexpressions.

Space does not permit a detailed discussion of the transformations required. However, the basic approach is to define a lexical order on identifiers and constants and to use this lexical order to order the variables and constants in subexpressions (where permitted by commutativity and associativity). Once the subexpressions have been placed into the lexically ordered canonical form, commutative-associative variants are identical except for the possible appearance of variables in one expression that do not appear in the other. For example, one might have two subexpressions of the form $(a + c + d - f + g + h - i - j)$ and $(a + b + d - e - f + h - i + j - k)$. The maximal common subexpression $(a + d - f + h - i)$ can be readily identified in these two expressions once they have been converted to yet another canonical form, in which the non-common variables are pulled to the end of the expressions: $((a + d - f + h - i) + c + g - j)$ and $((a + d - f + h - i) + b - e + j - k)$.

6 Conclusions

We have discussed a program-algebraic approach to the compiler optimization of eliminating common subexpressions. This approach has two major advantages:

- The approach is formulated in terms of algebraic manipulations of the expressions of the programming language (augmented with λ–expressions); the approach is therefore intuitive and both easy to understand and prove.
- The approach is implemented using transformations for which correctness-preservation is easy to prove; the approach can thus be used in a trusted compiler.

7 Acknowledgments

This work was supported by the BM/C3 directorate, Ballistic Missile Defense Organization, U.S. Department of Defense, and by the Grant–in–Aid Program for faculty of Virginia Commonwealth University. A portion of this work was done while Dr. Resler was on sabbatical at the University of Limerick, Ireland.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] H. P. Barendregt. *The Lambda Calculus—Its Syntax and Semantics.* North-Holland Publishing Company, 1981.

[3] James M. Boyle. Abstract programming and program transformation—an approach to reusing programs. In Ted. J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, volume I, chapter 15, pages 361–413. ACM Press (Addison-Wesley), 1989.

[4] James M. Boyle. Automatic, self-adaptive control of unfold-fold transformations. In E.-R Olderog, editor, *Programming Concepts, Methods and Calculi*, IFIP Transactions A-56, pages 83–103. Elsevier Science B.V. (North Holland), 1994.

[5] James M. Boyle and Terence J. Harmer. A practical functional program for the CRAY X-MP. *Journal of Functional Programming*, 2(1):81–126, January 1992.

[6] James M. Boyle, R. Daniel Resler, and Victor L. Winter. Do you trust your compiler? *IEEE Computer*, 32(5):65–73, May 1999.

[7] Z. Manna. *Mathematical Theory of Computation.* McGraw–Hill Inc., New York, 1974.

[8] Victor L. Winter and James M. Boyle. Proving refinement transformations for deriving high-assurance software. In *Proceedings, High-Assurance Systems Engineering Workshop, Niagara on the Lake, Ontario, Canada, Oct. 21-22, 1996*, pages 68–77. IEEE Computer Society Press, Los Alamitos, CA, 1997.

A A Portion of the TAMPR Subject–Language Grammar

```
<expr> ::= <op expr> | ...

<op expr> ::= <primary> |
              <op expr> <op> <primary>

<op> ::= <add op> | <mult op> | ...
```

```
<add op> ::= + | -

<mult op> ::= * | / | ...

<primary> ::= <entity> | ...

<entity> ::= <basic entity> <type info>

<type info> ::= <empty> | ...

<basic entity> ::= <const> | <var> | ...

<const> ::= <numerical constant> | ...

<var> ::=
   <function app> | <function expr> | ...

<function app> ::=
   <function expr> <pending args> | ...

<function expr> ::= <ident>
                  | <lambda abstrac-
tion>
                  | ( <expr> )
                  | ...

<ident> ::= <identifier> | ...

<pending args> ::=
   <args> | <args> <pending args>

<args> ::= ( ) | ( <expr list> )

<lambda abstraction> ::= lambda <body> end

<body> ::= <bound vars> <expr>

<bound vars> ::= <expr list> @ | ...

<expr list> ::=
   <expr> | <expr list> , <expr>
```

Note that this is a severely pruned version of the complete TAMPR subject–language grammar; portions of the grammar deemed not relevant to the understanding of the examples given in this paper have been removed.

An Assessment of Incomplete-LU Preconditioners for Nonsymmetric Linear Systems

John R. Gilbert
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304, USA
E-mail: gilbert@parc.xerox.com
AND
Sivan Toledo
School of Computer Science
Tel-Aviv University
Tel-Aviv 69978, Israel
E-mail: sivan@math.tau.ac.il

Keywords: Sparse nonsymmetric linear systems, iterative methods, direct methods, incomplete-LU preconditioners, pivoting.

Edited by: Marcin Paprzycki

Received: November 11, 1999 **Revised:** February 14, 2000 **Accepted:** June 8, 2000

We report on an extensive experiment to compare an iterative solver preconditioned by several versions of incomplete LU factorization with a sparse direct solver using LU factorization with partial pivoting. Our test suite is 24 nonsymmetric matrices drawn from benchmark sets in the literature.

On a few matrices, the best iterative method is more than 5 times as fast and more than 10 times as memory-efficient as the direct method. Nonetheless, in most cases the iterative methods are slower; in many cases they do not save memory; and in general they are less reliable. Our primary conclusion is that a direct method is currently more appropriate than an iterative method for a general-purpose black-box nonsymmetric linear solver.

We draw several other conclusions about these nonsymmetric problems: pivoting is even more important for incomplete than for complete factorizations; the best iterative solutions almost always take only 8 to 16 iterations; a drop-tolerance strategy is superior to a column-count strategy; and column MMD ordering is superior to RCM ordering.

The reader is advised to keep in mind that our conclusions are drawn from experiments with 24 matrices; other test suites might have given somewhat different results. Nonetheless, we are not aware of any other studies more extensive than ours.

1 Introduction

Black-box sparse nonsymmetric solvers, perhaps typified by the Matlab backslash operator, are usually based on a pivoting sparse LU factorization. Can we design a more efficient black-box solver that is based on an iterative solver with an incomplete LU preconditioner? This paper shows, using extensive experimental analysis, that the answer is no. An iterative solver with an incomplete LU preconditioner can sometimes be much more efficient than a direct solver in terms of both memory and time. But in most cases, the iterative solver is less less reliable and less efficient than a direct solver.

These conclusions are novel and correct, but one must keep in mind that they are drawn from a finite set of experiments. The conclusions are novel in the sense that no prior paper presented a systematic study that supports these conclusions. We are aware that our conclusions coincide with long-held viewpoints of some researchers, but these viewpoints were never substantiated by systematic study before. Hence, the novelty lies not in the conclusions themselves, but in the fact that they are supported by evidence. Other researchers hold opposite viewpoints—that preconditioned iterative solvers are more reliable and efficient that direct solvers. These viewpoints are typically based on some theoretical justification and/or on experimental results. We point out that there is not much theory on the convergence rates of *nonsymmetric* preconditioned iterative solvers, and that success with some matrices does not invalidate our conclusions, since our claim that that iterative solvers are less effective in most cases, not in all cases. To summarize, we present an experimental study, and *we draw conclusions from our data.* It is conceivable that a different test suite would have suggested somewhat different conclusions.

Large sparse linear solvers can be classified into three categories. Some solvers are problem specific and are of-

ten built into applications. Such solvers exploit structural and numerical properties that typify linear systems arising from a narrow application domain, and many of them use information about the problem that is not part of the linear system (for example, geometric information). The second category of solvers can be described as toolkits (see, for example, [10]). These solvers, often in the form of numerical libraries that are designed to be called from application programs, offer a choice of algorithms that can be combined to create linear solvers. The user must decide which algorithms to use and how to set tuning parameters that these algorithms may have. Typical toolkits provide several iterative solvers and several preconditioners. The third category of solvers are black-box solvers. These solvers solve linear systems using few assumptions on the origins of the systems and little or no guidance from the user. Most sparse direct solvers fall into this category. Problem-specific solvers often achieve high performance but require considerable effort from experts. Black-box solvers cannot always achieve the same level of performance, but are robust and easy to use. Toolkits are somewhere in-between. This paper evaluates incomplete-LU preconditioners only as candidates for inclusion in black-box solvers; it is by now clear that nonsymmetric incomplete-LU preconditioners should be included in toolkits and in some problem-specific applications.

Large sparse nonsymmetric linear systems are often solved by direct methods, the most popular of which is based on a complete sparse LU factorization of the coefficient matrix. Iterative solvers, usually preconditioned Krylov-space methods, are sometimes more efficient than direct methods. Iterative solvers can sometimes solve linear systems with less storage than direct methods, and they can sometimes solve systems faster than direct methods. The efficiency of iterative methods, in terms of both space and time, depends on the preconditioner that is used. In this paper we focus on a popular class of so-called general-purpose preconditioners, those based on incomplete LU factorization with or without partial pivoting. We do not consider other classes of general-purpose preconditioners, such as those based on sparse approximate inverses (see, for example, Grote and Huckle [8]), and algebraic multilevel solvers (see, for example, Shapira [12]). We also do not consider domain-specific preconditioners, such as domain-decomposition preconditioners for linear systems arising from discretizations of PDE's.

Incomplete-factorization preconditioners are constructed by executing a sparse factorization algorithm, but dropping some of the fill elements. Elements can be dropped according to numerical criteria (usually elements with a small absolute value), or structural criteria (e.g., so-called levels of fill)[1]. Since some of the fill elements are dropped, the resulting factorization is sparser and takes less time to compute than the complete factorization. If this factorization preconditions the linear system well, it enables the construction of an efficient iterative solver. We have implemented an algorithm that can construct such preconditioners. On some matrices, the resulting preconditioners are more than 10 times sparser than a complete factorization, and the iterative solution is more than 5 times faster than a direct solution. We plan to make our implementation, which can be used alone or as part of PETSc (a portable extensible toolkit for scientific computation [1]), publicly available for research purposes.

This strategy, however, can also fail. The algorithm can fail to compute a factorization due to a zero pivot, or it can compute a factorization that is unstable or inaccurate, which prevents the solver from converging. In other cases, the preconditioner can enable the iterative solution of the system, but without delivering the benefits that we expect. The running time can be slower than the running time of a direct solver, either because the iteration converges slowly or because the incomplete factorization is less efficient than a state-of-the-art complete factorization. The solver may need more space than a direct solver if the algorithm fails to drop many nonzeros, especially since an iterative solver cannot release the storage required for the matrix and needs storage for auxiliary vectors.

We have conducted extensive numerical experiments to determine whether incomplete factorizations can yield preconditioners that are reliable and efficient enough to be used in a black-box nonsymmetric linear solver. Our test cases are nonsymmetric linear systems that have been used to benchmark sparse direct solvers; all of them can be solved by complete sparse LU factorization with partial pivoting. The matrices range in size from about 1,100 to 41,000 rows and columns, and from about 3,700 to 1,600,000 nonzeros. Our main conclusion is that incomplete factorizations are *not* effective enough to be used in black-box solvers, even with partial pivoting. That is not to say that incomplete factorizations never produce effective preconditioners. In some cases they do. But in many cases state-of-the-art incomplete factorizations do not yield efficient preconditioners. Furthermore, in many other cases the resulting preconditioner is effective only within a small range of numerical dropping thresholds, and there are currently no methods for determining a near-optimal threshold. Therefore, current state-of-the-art incomplete factorizations cannot be used as preconditioners in iterative solvers that can be expected to be about as reliable and efficient as current direct solvers.

Our incomplete LU factorization algorithms are quite similar to Saad's ILTUP [11], but employ some additional techniques, which are described in Section 2. We describe our experimental methodology in Section 3. The discussion explains the structure of the experiments, the test matrices, and the hardware and software that were used. A summary of our experimental results is presented in Section 4. We discuss the results and present our conclusions

[1] We did not include level-of-fill dropping criteria for two reasons. First, we felt that the resulting preconditioners would be less effective and robust than those based on numerical dropping criteria. Second, level-of-fill criteria are *more difficult and computationally expensive* to implement in a pivoting factorization than numerical criteria since the level of every fill element must be kept in a data structure.

in Section 5.

2 Pivoting Incomplete LU Factorizations

This section describes our algorithm for incomplete LU factorization with partial pivoting. The algorithm is similar to Saad's ILUTP [11], but with some improvements.

Our algorithm is a sparse, left-looking, column-oriented algorithm with row exchanges. The matrix is stored in a compressed sparse-column format, and so are L and U. The row permutation is represented by an integer vector.

At step j of the algorithm, sparse column j of A is unpacked into a full zero column v. Updates from columns 1 through $j-1$ of L are then applied to v. These updates collectively amount to a triangular solve that computes the jth column of U, and a matrix-vector multiplication that computes the jth column of L. The algorithm determines which columns of L and U need to update v, as well as an admissible order for the updates, using a depth-first search (DFS) on the directed graph that underlies L. This technique was developed by Gilbert and Peierls [7].

Once all the updates have been applied to v, the algorithm factors v, using threshold partial pivoting. Specifically, the algorithm searches for the largest entry v_m in v_L, the lower part of v (we use v_U to denote the upper part of v). If $|v_d| > \tau|v_m|$, where $0 \leq \tau \leq 1$ is the *pivoting threshold* and v_d is the diagonal element in v, then we do not pivot. Otherwise we exchange rows d and m. (In the experiments below, we use either $\tau = 1$, which is ordinary partial pivoting, or $\tau = 0$, which amounts to no pivoting). The exact definition of a diagonal element in this algorithm is explained later in this section.

After the column is factored, we drop small elements from v_L and v_U. We never drop elements that are nonzero in A.[2] The algorithm can drop elements using one of two different criteria: (1) the algorithm can drop all but the largest k elements in v_U and the largest k elements in v_L, or (2) the algorithm can drop from v_U all the elements that are smaller[3] than $\delta \max_{i \in U} |v_i|$, and from v_L the elements that are smaller than $\delta \max_{i \in L} |v_i|$, where δ is the *drop threshold*. When we drop elements using a drop threshold δ, we use the same value of δ for all the columns. When we drop elements using a fill count k, we set k separately for each column. The value of k for a column j is a fixed multiple of the number of nonzeros in the jth column of A. We refer to this method as a *column-fill-ratio* method. After elements have been dropped, the remaining elements of v are copied to the sparse data structures that represent L and U and the algorithm proceeds to factor column $j+1$.

Our dropping rules differ somewhat from Saad's ILUT and ILUTP, in that we do not drop small elements of U during the triangular solve. Doing so would require us to base the drop threshold on the elements of A_j rather than on the elements of U_j, which we prefer. Also note that we compute the absolute drop threshold for a column separately for v_L and for v_U. We expect separate thresholds to give relatively balanced nonzero counts for L and for U, which is difficult to guarantee otherwise since their scaling is often quite different.

Our algorithm uses one more technique, which we call *matching maintenance* that attempts to maintain a trailing submatrix with a nonzero diagonal. The technique is illustrated in Figure 1. Before we start the factorization, we compute a row permutation that creates a nonzero diagonal for the matrix using a bipartite perfect-matching algorithm (this algorithm returns the identity permutation when the input matrix has a nonzero diagonal). When the algorithm exchange rows (pivots), the nonzero diagonal can be destroyed. For example, if in column 1 the algorithm exchanges rows 1 and i (in order to pivot on A_{i1}), and if A_{1i} (which moves to the diagonal) is zero, then we may encounter a zero diagonal element when we factor column i. The element A_{1i} will certainly be filled, since both A_{11} and A_{ii} are nonzero. Therefore, whether we encounter a zero diagonal element or not depends on whether A_{1i} is dropped or not after it is filled. Since A_{1i} will fill, our technique simply marks it so that it is not dropped even if it is small. In effect, we update the perfect matching of the trailing submatrix to reflect the fact that the diagonal of column i is now A_{1i} instead of A_{ii}, which is now in U. If we exchange row i with another row, say l, before we factor column i, we will replace A_{1i} by A_{li} as the diagonal element of column i. This marked diagonal element is not dropped even if we end up pivoting on another element in column i, say A_{ij}, because its existence ensures that the diagonal element in column j will be filled in. The resulting diagonal elements may be small, but barring exact cancellation they prevent zero pivots, at the cost of at most one fill per column.

The goal of the matching maintenance is to prevent structural zero pivots at the cost of one fill element per column. Our experiments show, however, that in very sparse factorizations such as with $\tau = 1$ (which we denote by ILU(0)), exact *numerical* cancellations are common even when this technique is employed. When we replace the numerical values of the matrix elements with random values, the factorizations do not break down. This experiment shows that the technique does indeed prevent structural breakdowns. We were somewhat surprised that exact numerical cancellations are so common in practice, even when structural breakdown is prevented. It remains an open problem to find a similarly inexpensive way to guarantee against exact numerical breakdown.

Before concluding this section, we would like to comment on two techniques that are employed in state-of-the-art complete LU factorization codes but that are not included in our incomplete LU code. The first technique

[2] We decided not to drop original nonzeros because we felt that dropping them might compromise the robustness of the preconditioner, but in some cases dropping original nonzeros may improve the efficiency of the preconditioner.

[3] All comparisons discussed in this section are of absolute values.

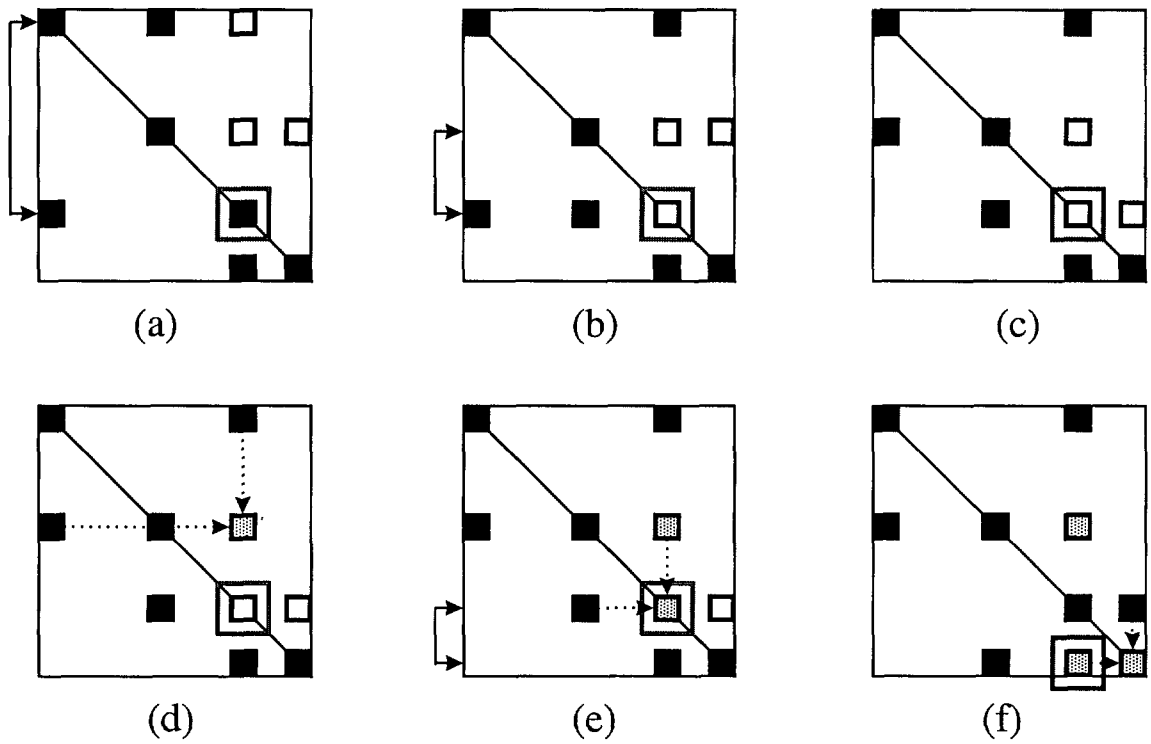


Figure 1: An example of the matching-maintenance method. Nonzero elements are represented by full squares, and zero elements by empty squares. A row exchange places a zero on the diagonal (a). This zero is marked as a diagonal element that must not be dropped, denoted by the enclosing (red) square. Another row exchange moves another zero to the diagonal (b). The new nonzero is marked and the previous one, which is now in U, is unmarked (c). The triangular solve fills in the first zero element, which is not dropped since we do not drop elements in U before the column factorization is complete (d). This fill element plus an original nonzero now cause the diagonal element to fill (e). A row exchange is performed, moving the element just filled off the diagonal. But since it is marked, it is not dropped (f), which ensures that the diagonal element in the last row will fill.

is called symmetric pruning [6]. This technique exploits structural symmetry by pruning the graph that the DFS searches for updating columns. The correctness of symmetric pruning depends on a complete factorization, so we could not use pruning in our code. The second technique is the exploitation of nonsymmetric supernodes [3] to improve the temporal locality in the factorization (and hence reduce cache misses). Maintaining supernodes in an incomplete factorization requires a restriction on the dropping rules (a supernodal algorithm would need to drop or retain entire supernode rows). This restriction would have increased the density of the factors, and we estimated that the increased density would offset the savings in running time gained from supernodes. Still, this technique could perhaps enhance performance on some matrices.

We have implemented this algorithm as a modification to the GP code [7]. The modifications are mostly restricted to the driver subroutine and to the column factorization subroutine. The subroutines that perform the DFS and update the current column are essentially unmodified. (We had to slightly modify all the routines in order to implement a column ordering mechanism). The perfect-matching code is by Pothen and Fan [9].

3 Experimental Methodology

This section describes our experimental methodology. We describe the structure of the experiments, the test matrices that we use, and the software and hardware platforms that we used to carry out the experiments. The experiments are summarized in Table 1.

Structure of the experiments

Our experiments compare a direct sparse LU solver with partial pivoting, SuperLU [3], with an iterative solver. We used a transpose-free quasi-minimum-residual (TFQMR) Krylov-space method with the pivoting incomplete LU preconditioner described above (see Saad [11], for example, for background on the Krylov-space methods discussed in this paper). For each matrix A we construct a random solution vector x (with elements uniformly distributed between 0 and 1), and multiply A by x to form a right-hand side b. We then solve the resulting linear system using SuperLU,

keeping track of the total solution time and the norm of the residual $A\hat{x} - b$, where $\hat{x}$ is the computed solution. We do not use iterative refinement. We then solve the same system several times using the Krylov-space iterative method with an incomplete-LU preconditioner, each time with a different value of the drop threshold. When the incomplete factorization breaks down due to zero pivots, we do not proceed with the iterative solver at all.

We chose TFQMR based on initial experiments that compared TFQMR, stabilized bi-conjugate gradients (BICGSTAB), and generalized minimum residual (GMRES), the last with restarts every 10, 20, and 40 iterations. These experiments, which are not reported here, showed that the overall performance and robustness of TFQMR and BICGSTAB are quite similar, with GMRES being less efficient. Since our goal is to evaluate iterative solvers as candidates for general-purpose black-box solvers, and not to compare different Krylov-space methods, we picked one solver for the experiments reported here.

The stopping criteria for the iterative solver are as follows. Convergence is defined as a residual whose 2-norm is at most 10^4 times the norm of the solution computed by SuperLU. While arbitrary, this choice reflects the fact that the accuracy achieved by a direct solver is often not needed in applications, while at the same time tying the required solution to the condition number of the system. Divergence is defined as a residual that grows by a factor of more than 10^4 relative to the initial residual. The solver also stops when the total solution time is more than 10 times the SuperLU solution time. We test for convergence after one iteration, 2, 4, 8, 16, and then every 16 iterations. This procedure reduces the overhead of convergence testing, while preventing the solver from iterating too many times when convergence is rapid. The convergence-testing subroutine computes true residuals.

Our time limit criterion and our convergence criterion cannot be implemented in applications, since they require the direct solution of the system. But they allow us to compare the iterative solver to the direct solver effectively without wasting too much computer time. (Even so, the experiments took about two weeks of computer time to complete.)

For each linear system, we ran four sets of experiments, two with partial pivoting (in both the complete and incomplete factorization), and two with no pivoting (sometimes called diagonal pivoting). The reference residual and running time used in the stopping criteria are always the ones from the SuperLU solver with partial pivoting. In two sets of experiments, one with and one without pivoting, we tested the drop-threshold preconditioner. In each set, we ran a direct solver and 33 iterative solvers, in which the drop threshold τ in the incomplete factorizations is set at $2^{-32}, 2^{-31}, \ldots, 2^{-1}, 1$. In the other two sets, also one with and one without pivoting, we tested column-fill-ratio preconditioners. We tested fill ratio $32, 16, 8, 4, 2$, and 1. (A fill ratio r means that when a column of A has n nonzeros, the corresponding columns of L and U each retain their largest rn elements plus the diagonal element and all the original A elements. Because the original nonzeros are never dropped, and because some columns may not fill by a factor of r, the total number of nonzeros in $U + L$ may be somewhat smaller or larger than $r\text{NNZ}(A)$.)

Most of the experiments were carried out using a column multiple-minimum-degree (MMD) ordering of the matrices, but we did run one set of experiments using a reverse-Cuthill-McKee (RCM) ordering. In this set, whose goal was to allow us to compare different orderings, we tested each matrix with two column-fill-ratio preconditioners, with ratios of 1 and 2.

We also ran three sets of experiments using symmetric-positive-definite (SPD) matrices. The first set was identical to the pivoting drop-threshold experiments carried out with nonsymmetric matrices. The other two sets compared an iterative solver specific to SPD matrices, denoted in Table 1 as CG+ICC, with a nonsymmetric iterative solver. These experiments are described more fully in Section 4.

Test Matrices

We performed the bulk of the experiments on a set of 24 test matrices, listed in Table 2. The table also lists the most important structural and numerical characteristics of the matrices, as well as whether pivoting was necessary for the complete and incomplete factorizations. The matrices are mostly taken from the Parallel SuperLU test set [4], where they are described more fully. The most important reason for choosing this set of matrices (except for availability) is that this is essentially the same set that is used to test SuperLU, which is currently one of the best black-box sparse nonsymmetric solvers. Therefore, this test set allows us to fairly assess whether preconditioned iterative methods are appropriate for a black-box solver.

We have also used a set of 12 symmetric positive-definite matrices in some experiments. Six of these matrices are from the Harwell-Boeing matrix collection and were retrieved from MatrixMarket, an online matrix collection maintained by NIST[4]. These include four structural engineering matrices (bcsstk08, bcsstk25, bcsstk27, bcsstk28), a power system simulation matrix (1108_bus), and a finite-differences matrix (gr_30_30). The other six matrices are image processing matrices contributed by Joseph Liu (den090, dis090, spa090, den120, dis120, spa120).

Software and Hardware Platforms

We used several mathematical libraries to carry out the experiments. The experiments were performed using calls to PETSc 2.0[5], an object-oriented library that implements several iterative linear solvers as well as numerous sparse matrix primitives [1]. PETSc is implemented in C and makes calls to the Basic Linear Algebra Subroutines

[4] Available online at http://math.nist.gov/MatrixMarket.

[5] Available online from http://www.mcs.anl.gov/petsc.

Experiment	Figures	Method	Number of Matrices	Type of Matrices	Ordering	Pivoting	Drop thresholds	Col fill ratios
I	2,3	TFQMR+ILU	24	NS	MMD on A^TA	Y	$2^{-32}, 2^{-31}, \ldots, 2^{-1}, 1$	
II	—	TFQMR+ILU	24	NS	MMD on A^TA	Y		$32, 16, 8, 4, 2, 1$
III	2,3	TFQMR+ILU	24	NS	MMD on A^TA	N	$2^{-32}, 2^{-31}, \ldots, 2^{-1}, 1$	
IV	—	TFQMR+ILU	24	NS	RCM on A	Y		$2, 1$
V	4	TFQMR+ILU	12	SPD	MMD on A^TA	Y	$2^{-32}, 2^{-31}, \ldots, 2^{-1}, 1$	
VI	4	TFQMR+ILU	12	SPD	MMD on A^TA	N	$2^{-32}, 2^{-31}, \ldots, 2^{-1}, 1$	
VII	—	QMR+ILU	6	SPD	RCM on A	N	$2^{-16}, 2^{-15}, \ldots, 2^{-1}, 1$	
VIII	—	CG+ICC	6	SPD	RCM on A	N	$2^{-16}, 2^{-15}, \ldots, 2^{-1}, 1$	

Table 1: A summary of the experiments reported in this paper. The table shows the iterative and preconditioning methods that are used in each set of experiments, the number of matrices and their type (general nonsymmetric, NS, or symmetric positive definite, SPD), the ordering of the matrices, whether pivoting was used, and the parameters of the incomplete-LU preconditioners. The first six sets of experiments were carried out using our own incomplete-LU implementation. The last two experiments, VIII and IX, were carried out using Matlab.
Figures 2, 3, and 4 give detailed data from some of the experiments, the other results are described in the main text.

(BLAS) to perform some operations on dense matrices and on vectors. PETSc includes several preconditioners, but it does not include a pivoting incomplete-LU preconditioner. We therefore added to PETSc two interfaces that call other libraries. The first interface enables PETSc to use the SuperLU library to order and factor sparse matrices. The second interface enables PETSc to use our modified version of the GP library to compute complete and incomplete LU factorizations.

SuperLU is a state-of-the-art library for sparse LU factorization with partial pivoting [3]. SuperLU achieves high performance by using a supernodal panel-oriented factorization, combined with other techniques such as panel DFS with symmetric pruning [6] and blocking for data reuse. It is implemented in C and calls the level 1 and 2 BLAS to perform computations on vectors and on dense submatrices. GP is a library for sparse LU factorization with partial pivoting [7]. GP is column oriented (that is, it does not use supernodes or panel updates). It uses column DFS, but no symmetric pruning. We modified GP to add the capability to compute incomplete factorizations with partial pivoting as explained in Section 2. GP is written in Fortran 77, except for some interface routines that are written in C.

We used the Fortran level-1 and level-2 BLAS. We used PETSc version 2.0.15. In SuperLU, we used the following optimization parameters: panels of 10 columns, relaxed supernodes of at most 5 columns, supernodes of at most 20 columns, and 2D blocking for submatrices with more than 20 rows or columns.

We ran the experiments on a Sun ULTRA Enterprise 1 workstation running the Solaris 2.5.1 operating system. This workstation has a 143 MHz UltraSPARC processor and 320 Mbytes of main memory. The processor has a 32 Kbytes on-chip cache, a 512 Kbytes off-chip cache, and a 288-bit-wide memory bus.

We used the Sunpro-3.0 C and Fortran 77 compilers, with the `-xO3` optimization option for C and the `-O3` optimization option for Fortran. Some driver functions (but no computational kernels) were compiled with the GCC C compiler version 2.7.2 with optimization option `-O3`.

4 Experimental Results

This section summarizes our results. This summary is quite long, and it is supported by many graphs that contain substantial amounts of information. This is a result of the complexity of the underlying data. We found that it was not possible to summarize the experiments concisely because each matrix or small group of matrices exhibited a different behavior. This complexity itself is part of our results, and we attempt to include enough information for readers to gauge it.

We begin with a broad classification of the matrices into those that require pivoting and those that do not. We then discuss each group separately. While most of the experiments were carried out with a column multiple-minimum-degree (MMD) ordering, we describe one set of experiments whose goal is to compare MMD to reverse-Cuthill-McKee (RCM) ordering. We also compare drop-threshold and column-fill-ratio factorizations with similar amounts of fill. We conclude the section with a discussion of two sets of experiments with symmetric-positive-definite matrices, whose goal is to determine whether the difficulties we encountered with the nonsymmetic matrices were due to the properties of the matrices, of the more general nonsymmetric solver, or of the incomplete-factorization paradigm itself.

General Classification of Matrices

In our experiments, matrices that are more than 50% structurally symmetric did not require pivoting for either the direct or the preconditioned iterative solvers. Matrices that

Matrix	N	NNZ	Structural Symmetry	Numerical Symmetry	Diagonal Dominance	Nonpivoting Direct	Nonpivoting Itererative
gre_1107	1107	5664	0.20	0.20	-1.0e+00		
orsirr_1	1030	6858	1.00	0.50	2.9e-04	Y	Y
mahindas	1258	7682	0.03	0.01	-1.0e+00	Y	
sherman4	1104	3786	1.00	0.29	2.0e-04	Y	Y
west2021	2021	7310	0.00	0.00	-1.0e+00		
saylr4	3564	22316	1.00	1.00	-6.1e-07	Y	Y
pores_2	1224	9613	0.66	0.47	-1.0e+00	Y	Y
extr1	2837	10969	0.00	0.00	-1.0e+00		
radfr1	1048	13299	0.06	0.01	-1.0e+00		
hydr1	5308	22682	0.00	0.00	-1.0e+00		
lhr01	1477	18428	0.01	0.00	-1.0e+00		
vavasis1	4408	95752	0.00	0.00	-1.0e+00		
rdist2	3198	56834	0.05	0.00	-1.0e+00		
rdist3a	2398	61896	0.15	0.01	-1.0e+00		
lhr04	4101	81067	0.02	0.00	-1.0e+00		
vavasis2	11924	306842	0.00	0.00	-1.0e+00		
onetone2	36057	222596	0.15	0.10	-1.0e+00		
onetone1	36057	335552	0.10	0.07	-1.0e+00		
bramley1	17933	962469	0.98	0.73	-1.0e+00	Y	Y
bramley2	17933	962537	0.98	0.78	-1.0e+00	Y	Y
psmigr_1	3140	543160	0.48	0.02	-1.0e+00		
psmigr_2	3140	540022	0.48	0.00	-1.0e+00	Y	
psmigr_3	3140	543160	0.48	0.01	-1.0e+00		
vavasis3	41092	1683902	0.00	0.00	-1.0e+00		

Table 2: The nonsymmetric matrices that are used in our experiments. The table shows the order N and number of nonzeros (NNZ) of the matrices, structural and numerical symmetry, diagonal dominance, and whether the matrices require pivoting in direct and iterative factorizations. The structural symmetry is the fraction of nonzeros whose symmetric matrix elements are also nonzeros, the numerical symmetry is the fraction of nonzeros whose symmetric elements have the same numerical value, and the diagonal dominance is defined as $\min_{i=1\ldots N}(|A_{ii}|/\sum_{j\neq i}|A_{ij}|) - 1$, so a matrix with nonnegative diagonal dominance is diagonally dominant, and a matrix with diagonal dominance -1 has a zero on the diagonal.

are less than 50% structurally symmetric generally require pivoting for both the direct and the preconditioned iterative solvers, with two exceptions: mahindas and psmigr_2 (3% and 48% structurally symmetric, respectively). A nonpivoting direct solver worked on these two matrices (although the solutions produced were significantly less accurate than solutions obtained with pivoting factorizations), but the iterative solvers converged only with pivoting preconditioners. In both cases the nonpivoting iterative solver detected divergence and stopped. The 2-norms of the forward errors were about 7 orders of magnitude larger with the nonpivoting direct solver than with the pivoting solver for mahindas, and 5 orders of magnitude larger for psmigr_2. The 2-norms of the residuals were also larger by similar factors.

We believe that the 50% structural symmetry cutoff point for the need to pivot is significantly influenced by the set of test matrices that we used. We believe that there are matrices that arise in applications with more than 50% structural symmetry that do not require pivoting and matrices with less than 50% structural symmetry that do require pivoting.

Nonpivoting Factorizations

Figure 2 summarizes the results of experiments I and III the 6 matrices did not require a pivoting preconditioner. On 5 of these 6 matrices, the iterative solver with the best drop-threshold preconditioner was faster than SuperLU. On 3 of the 6, even a pivoting preconditioner was faster than SuperLU. On the other 2 matrices in which a nonpivoting preconditioner was faster than SuperLU, bramley1 and bramley1, the pivoting preconditioners were not able to reduce the running time over either GP or SuperLU. On one matrix, sherman4, all the preconditioners converged, but none reduced the running time below either GP or SuperLU.

On 4 of the 6 matrices that did not require a pivoting preconditioner, the iterative solver converged with very sparse factorizations. On 3 of the 4, a factorization with no fill at all (ILU(0)) converged. On saylr4, factorizations with no fill or very little fill did not converge to an accurate solution. They did converge to a less accurate solution. The sparsest factorization that converged had only 117% of the fill of ILU(0) and only 5% of the fill of the complete factorization. On the two remaining matrices, bramley1 and bramley2, even the sparsest (ILU(0)) nonpivoting preconditioners converged, but all the pivoting preconditioners that converged were almost complete factorizations.

Generally speaking, the only failure mode for all 8 matrices that did not require a pivoting direct solver was exceeding the time limit. There were essentially no numerically zero pivots or unstable factorizations. In some cases we believe that a higher time limit would allow convergence; in some cases the solver has converged to a solution that was not accurate enough and could not further reduce the residual; and in some cases the solver exceeded the time limit without significantly reducing the residual at all. In two cases a single drop-threshold value produced an unstable factorization, once when pivoting (on pores_2), and once when not pivoting (on bramley2).

Pivoting Factorizations

We now discuss the results of experiments I and III with the 16 matrices that required pivoting for both complete and incomplete factorizations, as well as with the 2 matrices that did not require pivoting for a complete factorization, but did require pivoting incomplete factorizations. The results of these experiments are summarized in Figures 3a, 3b, and 3c.

On 7 of the 18 matrices, the iterative solver (with the best preconditioner) was faster than both SuperLU and GP (in 2 of these 7 cases the improvement over SuperLU was less than 20%). On 3 more matrices, the iterative solver was faster than GP but not faster than SuperLU (in one of the 3 cases the improvement over GP was less than 20%). On the remaining 8 matrices, the iterative solver did not reduce the solution time over either SuperLU or GP.

Twelve of the matrices converged with a preconditioner with 50% or less of the fill of a complete factorization. Only 7 converged with 40% or less, only 5 with 20% or less, and 2 with less than 10%. Only 2 matrices, psmigr_1 and psmigr_3, converged with an ILU(0) factorization.

Failure modes in the pivoting preconditioners on 12 of the 18 matrices included unstable factorizations that were detected as either numerically zero pivots during the factorization or divergence during the iterations (the 2-norm of the residual grows by a factor of 10^4 or more). Zero pivots were detected on 11 matrices, and divergence on 6. On the remaining 6 out of the 18 matrices, 2 matrices always converged, and on the other 4 the only failure mode was exceeding the time limit.

The Effect of Ordering on Convergence

Since RCM ordering produces significantly more fill than MMD in complete and nearly complete factorizations, we only tested RCM orderings on relatively sparse incomplete factorizations, namely, column-fill-ratio factorizations with ratios of 1 and 2. We now compare these RCM preconditioners, from experiment IV, with column-fill-raio MMD preconditioners with the same ratios from experiment II.

In only 18 of the 48 experiments (24 matrices with 2 fill ratios each), both orderings converged. In 7 more the MMD-ordered preconditioner converged but the RCM-one exceeded the time limit (which was identical for both orderings and based on the SuperLU time with an MMD ordering). There were no other cases.

When both converged, MMD was faster in 8 experiments and RCM in 10. But when MMD was faster, the RCM preconditioner took on average 207% more time to converge (that is, RCM was on average 3 times slower), whereas when RCM was faster, MMD took on average 47% more time (1.5 times slower). The MMD and RCM

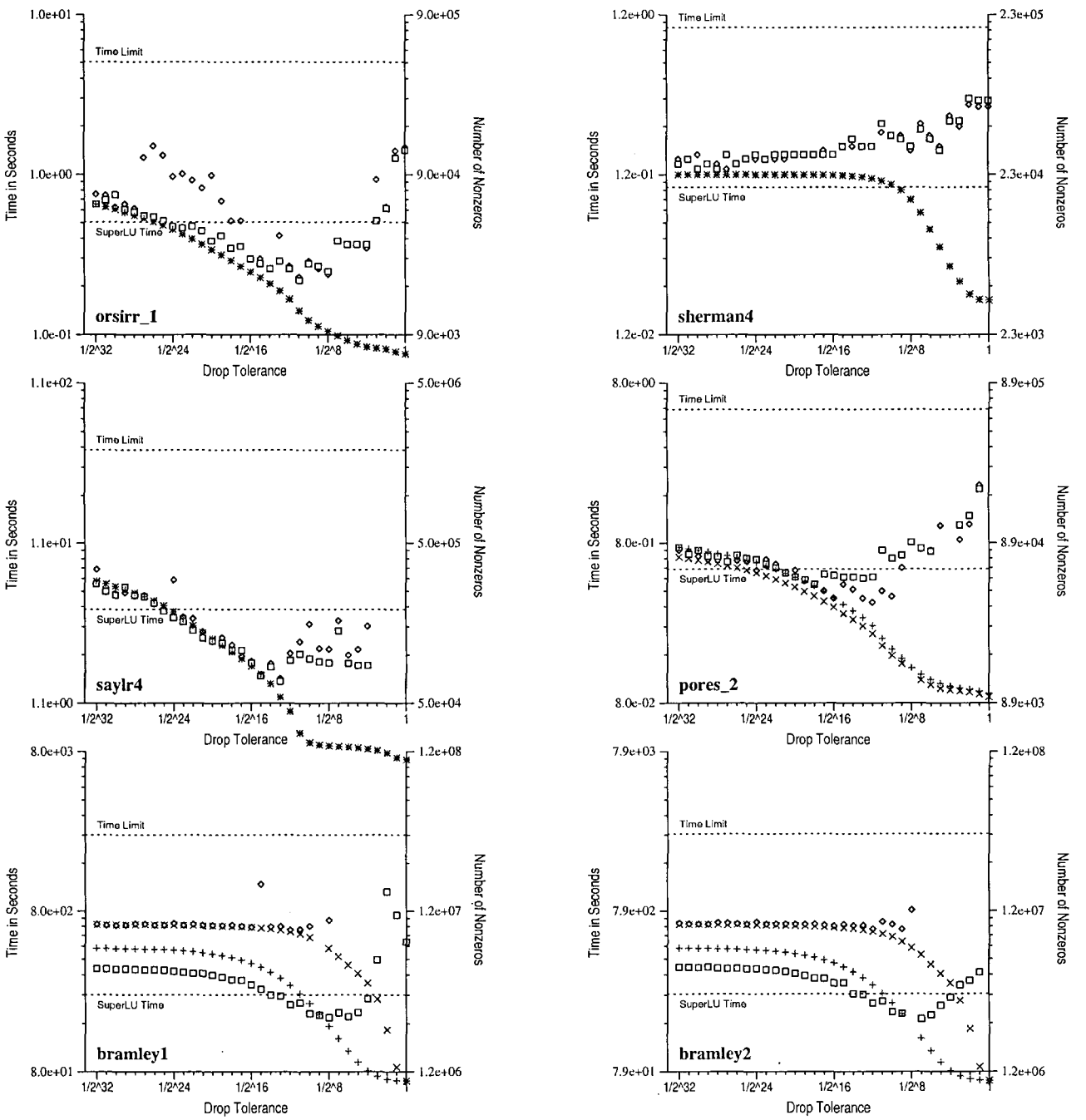


Figure 2: Experiments I and III. Running times (factorization+iteration) and nonzero counts for matrices that can be incompletely factored without pivoting, as a function of the drop threshold. The running times with pivoting are denoted by diamonds, and the running times without pivoting by squares. The nonzero counts are denoted by (red) x's for factorizations with pivoting, and by (red) crosses without pivoting. The y-axes are scaled so that the running time and nonzero count of the complete GP factorization with pivoting (and triangular solution, for time) would fall on the middle hash marks. The scale on both axes is logarithmic. The time limit for iterative solutions is 10 times the total factor-and-solve time for SuperLU.

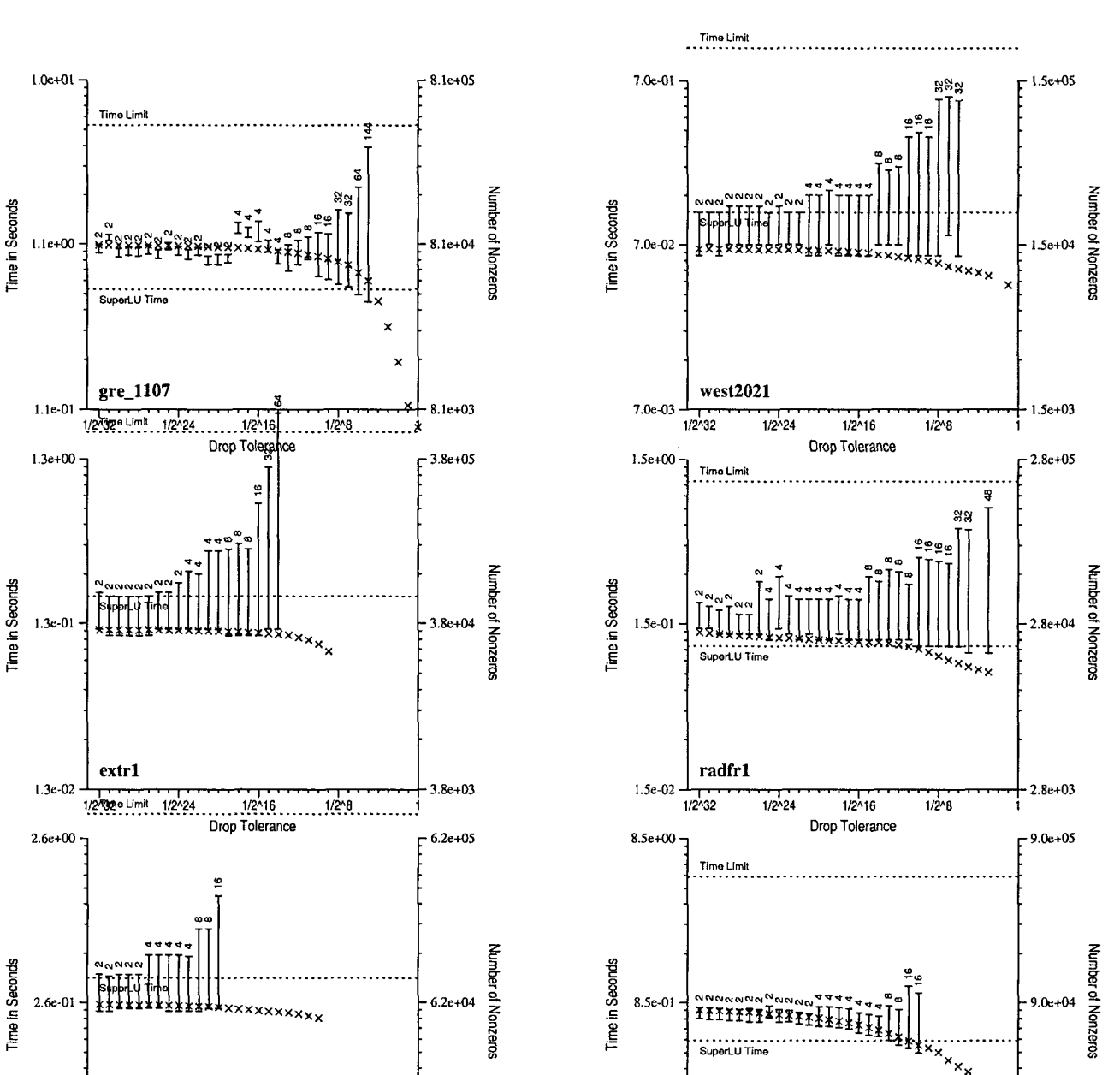


Figure 3a: Experiments I and III. Running times and nonzero counts for matrices that cannot be incompletely factored without pivoting, as a function of the drop threshold. The running times are denoted by vertical bars. The low end of the bar denotes the factorization time, and the high end shows the entire solution time (factorization+iteration). The nonzero counts are denoted by (red) x's. The y-axes are scaled so that the running time and nonzero count of the complete GP factorization with pivoting (and triangular solution, for time) would fall on the middle hash marks. The scale on both axes is logarithmic. The time limit for iterative solutions is 10 times the total factor-and-solve time for SuperLU.

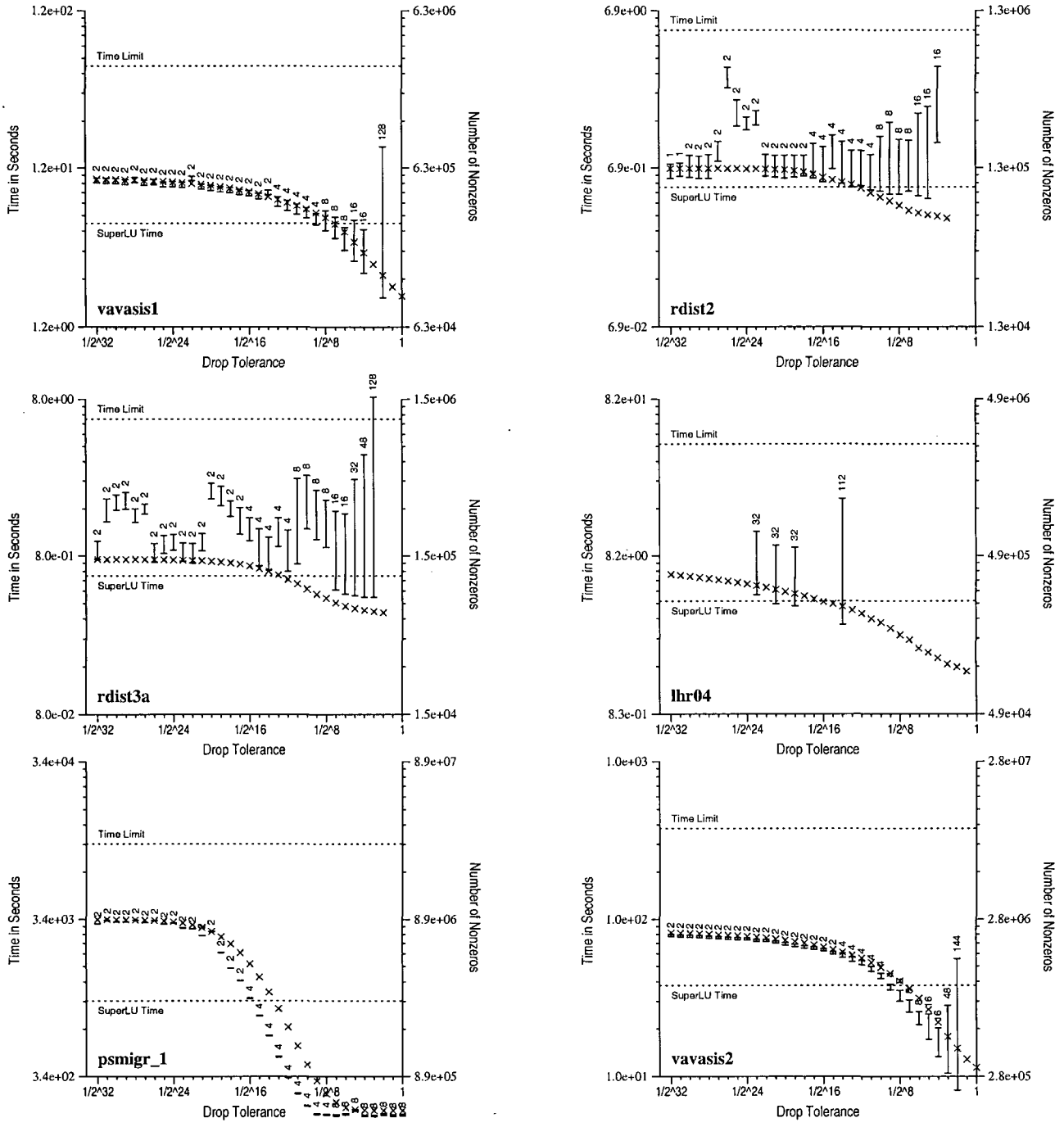


Figure 3b: Experiments I and III. Running times and nonzero counts for matrices that cannot be incompletely factored without pivoting (continued).

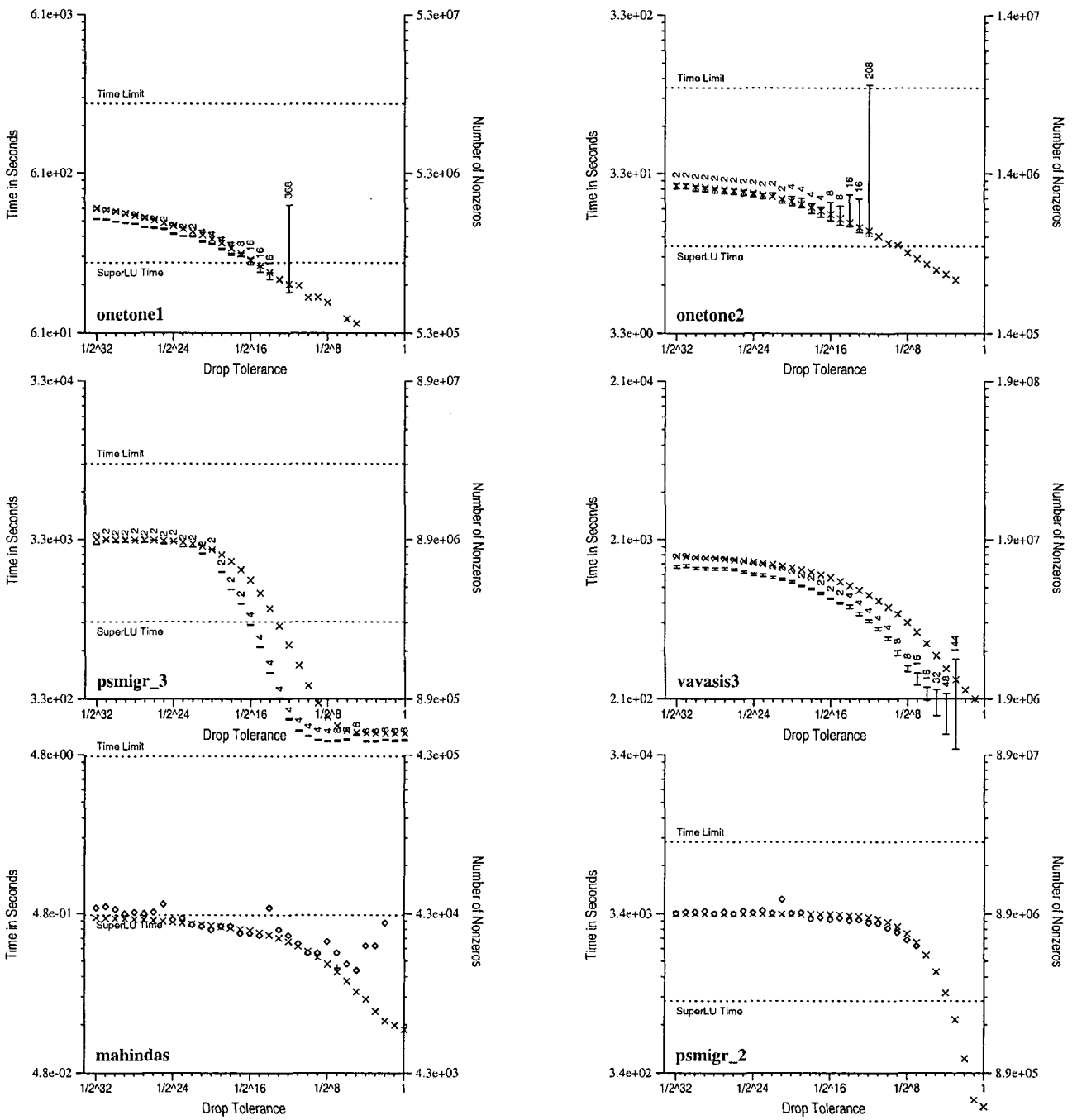


Figure 3c: Experiments I and III. Running times and nonzero counts for matrices that cannot be incompletely factored without pivoting (continued). SuperLU ran out of memory on vavasis3, so no SuperLU time is shown. The time limit for the iterative solution of vavasis3 was arbitrarily set at 10,000 seconds.

preconditioners had similar numbers of fill elements, but the MMD preconditioners were a little sparser on average in both cases.

Our experiments show that pivoting incomplete factorizations with column ordering based on an MMD ordering of A^TA usually converge faster than pivoting incomplete factorizations with column ordering based on an RCM ordering of A.

One possible reason for the difference in performance is that the MMD preconditioners retain a larger fraction of the fill of a complete factorization than the RCM ones. We expect an MMD ordering to yield a sparser complete factorization than an RCM ordering. Since the column-fill-ratio preconditioners had about the same number of fill elements with both orderings, more fill was dropped from the RCM preconditioners than from the MMD ones.

Column-Fill-Ratio Incomplete Factorizations

To compare the quality of drop-tolerance preconditioners and column-fill-ratio preconditioners, we matched each column-fill-ratio preconditioner from experiment II with a drop-tolerance preconditioner with a similar number of nonzeros from experiment I. Specifically, we paired with a column-fill-ratio preconditioner with k nonzeros a drop-tolerance preconditioner with between $0.9k$ and k nonzeros, if there was one. We broke ties by choosing the preconditioner with the largest number of nonzeros among all admissible ones.

Out of 144 drop tolerance preconditioners, 108 were paired. In 15 of the pairs neither preconditioner converged within the time limit. In 16 pairs only the drop-tolerance preconditioner converged. Among the 77 pairs in which both preconditioners converged, the column-fill-ratio preconditioners required, on average, factors of 21.7 more iterations and 2.67 more time to converge (time including both factorization and iterations). There were only 6 pairs in which the column-fill-ratio preconditioner converged faster. Among these 6, the column-fill-ratio preconditioners, required on average a factor of 0.87 less time to converge. We conclude that for a given number of nonzeros, a drop-tolerance preconditioner is usually dramatically better, and never much worse.

Symmetric Positive Definite Matrices

Although this paper only focuses on nonsymmetric matrices, we did perform some experiments on symmetric-positive-definite matrices. The goal of these experiments was not to assess iterative solvers for SPD matrices, but to answer two specific questions: (a) does our iterative solver perform better on SPD matrices than on nonsymmetric matrices, and if not, (b) can an iterative solver that exploits the properties of SPD matrices perform better?

We ran two sets of experiments. In the first set, consisting of experiments V and VI, we used exactly the same Krylov-space method and the same preconditioner, nonsymmetric LU with and without partial pivoting, to solve 12 SPD matrices. Our goal in this first set was to determine whether the behavior of our solver is significantly better when the matrices are SPD.

The results, which are described in Figures 4a and 4b, show that even SPD matrices can cause the solver difficulties. Out of the 12 matrices, only 4 can be solved with large drop tolerances. There is only one case (bcsstk08) of spectacular reduction in running time relative to the direct solver. Most of the failures are caused by exceeding the time limit, but in a few cases the factorization is unstable and causes the solver to diverge. There is no significant difference in performance between pivoting and nonpivoting factorizations.

In the second set, consisting of experiments VII and VII, we compared a symmetric and a nonsymmetric solver on 6 of the matrices (bcsstk08, bcsstk25, bcsstk27, bcsstk28, 1138_bus, and gr_30_30). We compared a conjugate gradient (CG) method using a drop-tolerance incomplete Cholesky preconditioner to a QMR method using a drop-tolerance incomplete LU preconditioner. The first solver exploits the fact that the matrices are SPD, while the second is quite similar to our nonsymmetric iterative solver, except that it does not pivot. We used Matlab 5.0 for this experiment, and symmetrically permuted the matrices using RCM ordering. We again set the required residual to be 10^4 times less accurate than a residual obtained by direct solution. We set the maximum number of iterations to 512.

The results of this experiment show no significant qualitative differences between the symmetric and the nonsymmetric solver. The symmetric solver was able to solve the matrices bcsstk08, bcsstk27, gr_30_30, and 1138_bus even with very large drop tolerances (more than 512 iterations were required on 1138_bus with the two sparsest factorizations). The symmetric solver failed to converge in 512 iterations with most of the values of the drop tolerance on the other two matrices. Reduction in running time relative to the direct solver was obtained only on bcsstk08. This behavior is similar to the behavior of our nonsymmetric solver.

5 Conclusions

The primary conclusion from our experiments is that iterative solvers with incomplete LU preconditioners can be very effective for some nonsymmetric linear systems, but they are not robust enough for inclusion in general-purpose black-box linear solvers.

Iterative solvers sometimes save a factor of about 10 in both time and space relative to a state-of-the-art direct sparse solver. But in most cases even the best drop-threshold value does not produce a very effective preconditioner. Also, to our knowledge there are no known techniques for determining an optimal or near-optimal drop-threshold value. Therefore, a black-box solver is likely to operate most of the time with sub-optimal drop thresholds,

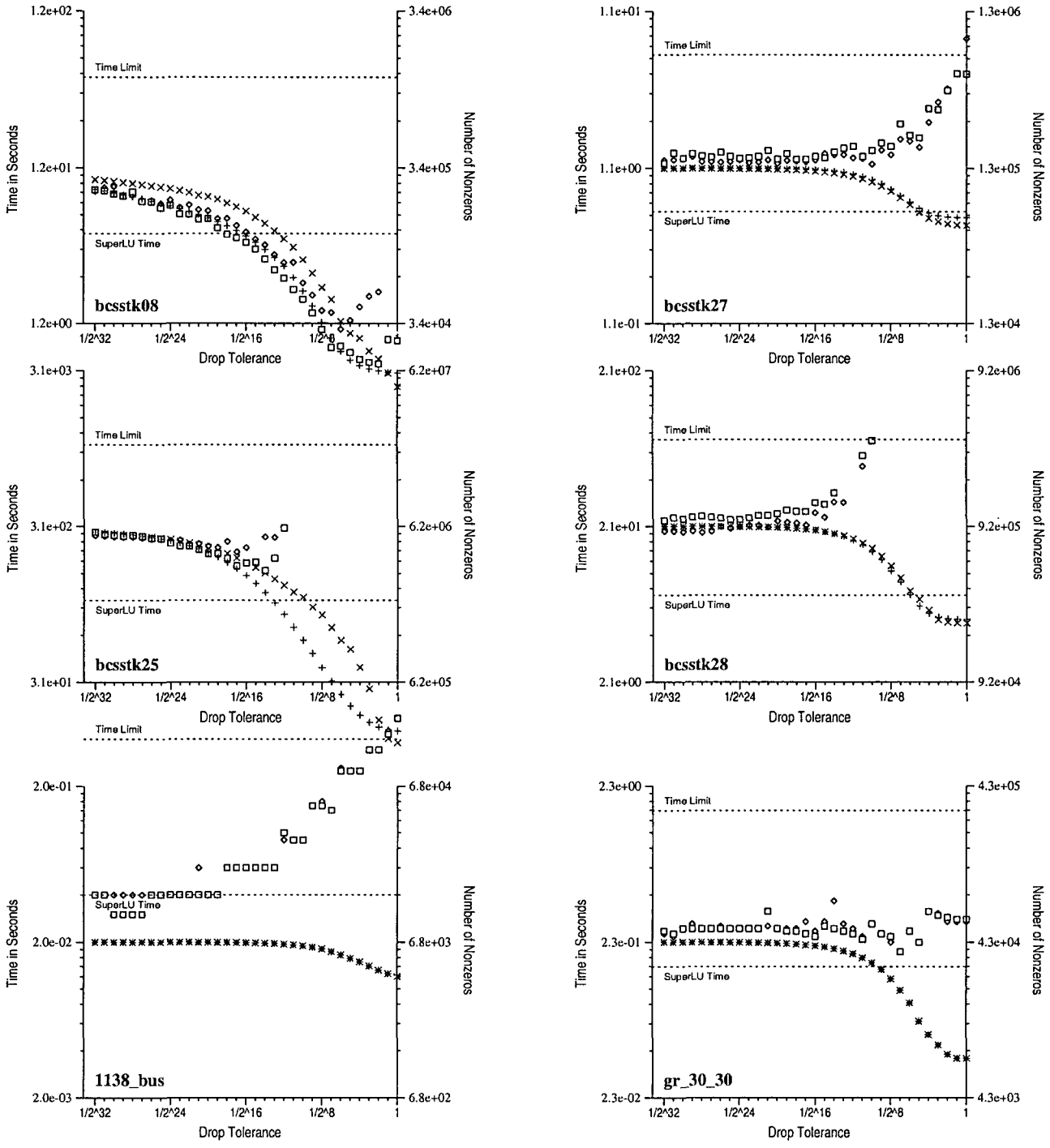


Figure 4a: Experiments V and VI. Running times and nonzero counts for SPD matrices ordered using MMD on A^TA, as a function of the drop threshold. The running times with pivoting are denoted by diamonds, and the running times without pivoting by squares. The nonzero counts are denoted by (red) x's for factorizations with pivoting, and by (red) crosses without pivoting. The y-axes are scaled so that the running time and nonzero count of the complete GP factorization with pivoting (and triangular solution, for time) would fall on the middle hash marks. The scale on both axes is logarithmic. The time limit for iterative solutions is 10 times the total factor-and-solve time for SuperLU.

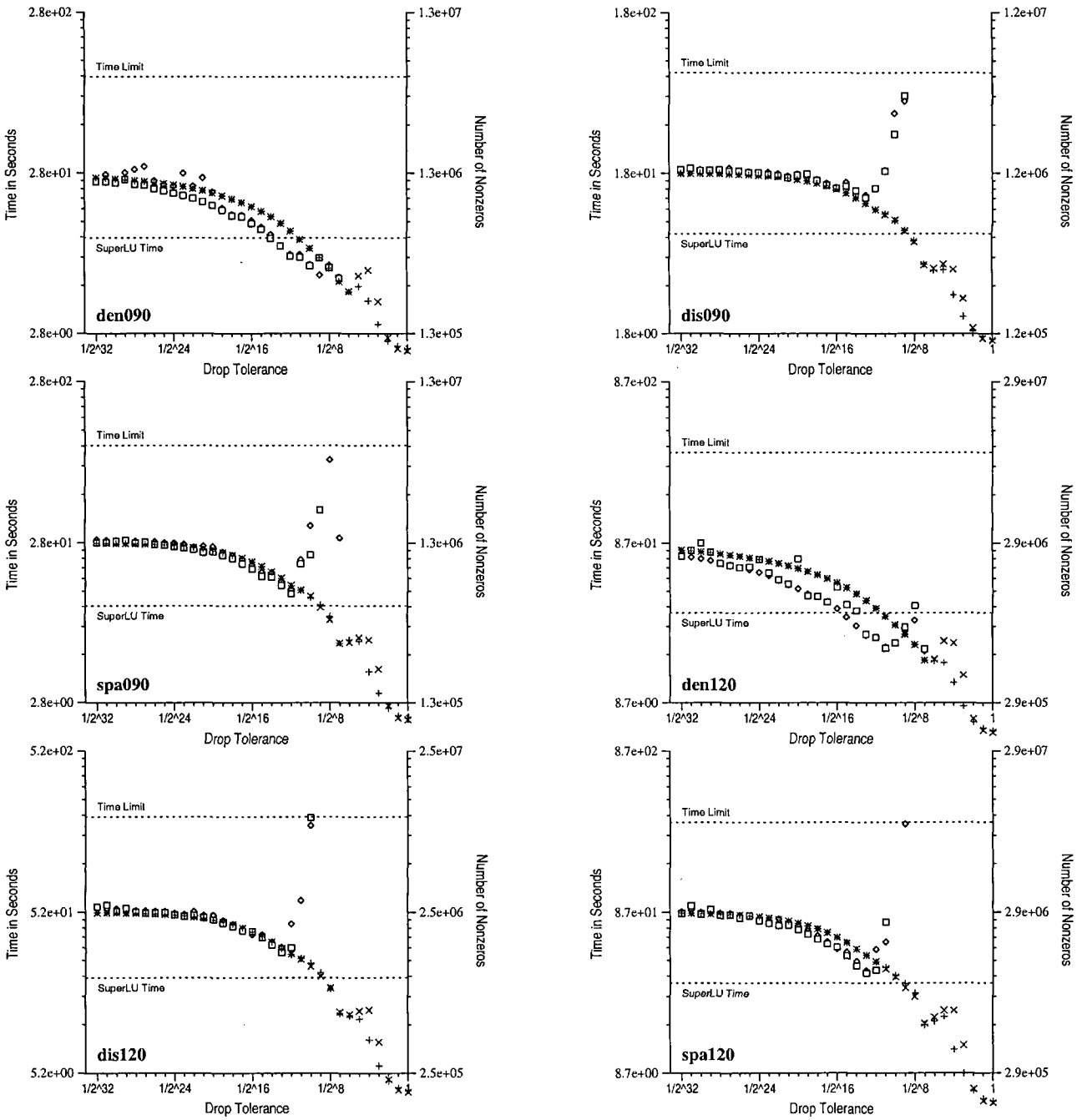


Figure 4b: Experiments V and VI. Running times and nonzero counts for SPD matrices ordered using MMD on A^TA (continued).

which can lead to slow convergence or no convergence. Out of hundreds of iterative solutions, few were more than 10 times faster than a direct solver, but many were more than 10 times slower.

Our experiments on SPD matrices, while limited, suggest that our primary conclusion remains valid even if we restrict our attention to SPD matrices, and perhaps even to SPD matrices solved by symmetric methods. These experiments, however, are limited in scope, and were only meant to indicate whether the nonsymmetry of the matrices or of the solvers caused the difficulties that we have reported. They were not meant to provide an evaluation of iterative solvers for SPD matrices and should not be used as such.

We also draw some secondary conclusions from the data on nonsymmetric matrices.

- First, pivoting in incomplete LU is necessary in many cases, even though we always begin by permuting the matrices to create a nonzero diagonal. Pivoting is necessary whenever pivoting is required for the direct solution, and it is necessary even for some systems that can be directly solved without pivoting. In other words, pivoting is more important in the incomplete case than in the complete case.

- Second, the best overall running times for the iterative solution of single linear systems (as opposed to multiple systems with the same matrix) are almost always achieved with around 8 to 16 iterations.

- Third, drop-tolerance preconditioners are more effective than column-fill-ratio preconditioners with a similar amount of fill. This is unfortunate, since column-fill-ration and other fixed-fill strategies allow solvers to tailor the preconditioner to the amount of available main memory.

- Fourth, MMD column ordering yields more efficient preconditioners than RCM column ordering. (Note that Duff and Meurant [5] showed that for SPD matrices, RCM is often a more effective ordering for incomplete-Cholesky preconditioners with no fill.)

Iterative solvers are suitable for toolkits for the solution of sparse linear systems. Toolkits implement multiple algorithms and enable the user to construct a solver that can efficiently solve a given problem. An iterative solver that works well on one matrix may be inefficient or even fail to converge on another. For example, Grote and Huckle [8] switch from right to left preconditioning in order to achieve convergence with a sparse-approximate-inverse preconditioner on pores2, and Chow and Saad [2] switch from row to column-oriented factorization to achieve convergence with lhr01. Chow and Saad also use a variety of other techniques to solve other systems. There are no established criteria that can guide an automatic system as to which solver is appropriate for a given matrix. Therefore, it is necessary to give the user control over which algorithms are used to solve a linear system, which is exactly what toolkits do.

Direct solvers, on the other hand, are suitable for black-box solvers. A single direct solver with a single ordering was reliable and efficient on all of our test matrices. In comparison, our experiments have not turned up any single iterative solver (say, TFQMR with a specific drop-threshold preconditioner) that can rival the overall reliability and performance of this direct solver. While tuning a direct solver—by changing the ordering, for example—can sometimes improve its performance, we believe that direct solvers, even "right out of the box" with no tuning at all, are more reliable and more efficient than iterative solvers.

The discussion in the preceding paragraphs suggests that there are problems that can be solved, but not by black-box solvers. We did not include problems that are too large to be solved by a direct solver on a high-end workstation in our test set because we would not be able to compare direct and iterative solvers on them. Our experiments show that some problems can be solved by an iterative solver with a drop-threshold preconditioner with no or little fill, and that this solver requires significantly less memory than the direct solver. The direct solver would ran out of memory trying to solve similar but larger problems, but the iterative solver should be able to solve them. This implies that black-box solvers can solve all problems up to a certain size with reasonable efficiency, and that larger problems can sometimes be solved by more specialized solvers.

One potentially useful technique is to adaptively search for an efficient preconditioner, hoping not to waste too much time in the search. Two facts can guide us in designing the search. Since an efficient preconditioner usually yields a solution in 8–16 iterations, we can abort the solver after about 20 iterations, or if we encounter a zero pivot, and try to construct a new preconditioner. Since most of the solution time is spent in the factorization phase when the preconditioner is relatively dense, one should start the search with very sparse preconditioners, so that aborting and refactoring is not too expensive. One flaw in this idea is that some matrices do not fill very much (e.g., west2021), so each aborted iterative solution can be almost as expensive as a direct solution.

We believe that studying iterative solvers in the context of the reliability and performance of a direct solver is important. While comparisons of iterative solution techniques to one another can be very informative, they do not provide practitioners with specific practical advice. Since practitioners have the option to use direct solvers, which are generally reliable and efficient, they need to know whether the iterative solver under study outperforms state-of-the-art direct solvers. The knowledge that one iterative solver outperforms another is usually not sufficient for deciding to deploy it. We hope to see more direct/iterative comparative studies in the future, at least for nonsymmetric matrices, especially since SuperLU is freely available on NETLIB.

To summarize, we believe that incomplete LU preconditioners with partial pivoting are useful components in a toolkit for the iterative solution of linear systems, such as PETSc. Such preconditioners can be very effective in indi-

vidual applications that give rise to a limited class of linear systems, so that the drop threshold and other parameters (e.g., ordering) can be tuned and the entire solver can be tested for reliability. But such iterative solvers cannot currently rival the reliability and performance of direct sparse solvers.

Finally, early responses to this paper convice us that more such studies are needed.

Acknowledgments

Thanks to Jim Demmel and Xiaoye Li for helpful discussions. Thanks to Barry Smith and the rest of the PETSc team for their dedicated support of PETSc. Thanks to Padma Raghavan and Esmond Ng for helpful discussions and for contributing the image processing matrices. Thanks to Alex Pothen for his perfect-matching code. Thanks to Cleve Ashcraft, Ed Rothberg, Yousef Saad, and Barry Smith for helpful comments on a draft of this paper. Thanks to the editor, Marcin Paprzycki, and to the anonymous referees for their comments.

This research was supported in part by DARPA contract number DABT63-95-C-0087 and by NSF contract number ASC-96-26298.

References

[1] Satish Balay, William Gropp, Lois Curfman McInnes, and Barry Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 — Revision 2.0.15, Argonne National Laboratory, 1996.

[2] Edmond Chow and Yousef Saad. Experimental study of ILU preconditioners for indefinite matrices. Technical Report UMSI 97/97, Supercomputer Institute, University of Minnesota, June 1997.

[3] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. Technical Report CSL 95/03, Xerox Palo Alto Research Center, September 1995.

[4] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. Technical Report CSD–97–943, Computer Science Division, University of California, Berkeley, February 1997.

[5] Iain S. Duff and Gérard Meurant. The effect of ordering on preconditioned conjugate gradient. *BIT*, 29(4):635–657, 1989.

[6] S .C. Eisenstat and J. W. H. Liu. Exploiting structural symmetry in a sparse partial pivoting code. *SIAM Journal on Scientific and Statistical Computing*, 14:253–257, 1993.

[7] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal on Scientific and Statistical Computing*, 9:862–874, 1988.

[8] Marcus J. Grote and Thomas Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18(3):838–853, 1997.

[9] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 16(4):303–324, December 1990.

[10] Yousef Saad. Sparskit: A basic tool-kit for sparse matrix computations, version 2. Software and documentation available online from http://www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html, University of Minnesota, Department of Computer Science and Engineering, 1994.

[11] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996.

[12] Yair Shapira. A multi-level method for sparse linear systems. Technical Report Technical Report LA-UR-97-2551, Los Alamos National Laboratory, 1997.

THE MINISTRY OF SCIENCE AND TECHNOLOGY OF THE REPUBLIC OF SLOVENIA

Address: Trg OF 13, 1000 Ljubljana,
Tel.: +386 61 178 46 00, Fax: +386 61 178 47 19.
http://www.mzt.si, e-mail: info@mzt.si
Minister: Lojze Marinček, Ph.D.

Slovenia realises that that its intellectual potential and all activities connected with its beautiful country are the basis for its future development. Therefore, the country has to give priority to the development of knowledge in all fields. The Slovenian government uses a variety of instruments to encourage scientific research and technological development and to transfer the results of research and development to the economy and other parts of society.

The Ministry of Science and Technology is responsible, in co-operation with other ministries, for most public programmes in the fields of science and technology. Within the Ministry of Science and Technology the following offices also operate:

Slovenian Intellectual Property Office (SIPO) is in charge of industrial property, including the protection of patents, industrial designs, trademarks, copyright and related rights, and the collective administration of authorship. The Office began operating in 1992 - after the Slovenian Law on Industrial Property was passed.

The Standards and Metrology Institute of the Republic of Slovenia (SMIS) By establishing and managing the systems of metrology, standardisation, conformity assessment, and the Slovenian Award for Business Excellence, SMIS ensures the basic quality elements enabling the Slovenian economy to become competitive on the global market, and Slovenian society to achieve international recognition, along with the protection of life, health and the environment.

Office of the Slovenian National Commission for UNESCO is responsible for affairs involving Slovenia's co-operation with UNESCO, the United Nations Educational, Scientific and Cultural Organisation, the implementation of UNESCO's goals in Slovenia, and co-operation with National commissions and bodies in other countries and with non- governmental organisations.

General Approaches – Science Policy

Educating top-quality researchers/experts and increasing their number, increasing the extent of research activity and achieving a balanced coverage of all the basic scientific disciplines necessary for:
- quality undergraduate and postgraduate education,
- the effective transfer and dissemination of knowledge from abroad,
- cultural, social and material development,
- promoting the application of science for national needs,
- promoting the transfer of R&D results into production and to the market,
- achieving stronger integration of research into the networks of international co-operation (resulting in the complete internationalisation of science and partly of higher education),
- broadening and deepening public understanding of science (long-term popularisation of science, particularly among the young).

General Approaches – Technology Policy

- promotion of R&D co-operation among enterprises, as well as between enterprises and the public sector,
- strengthening of the investment capacities of enterprises,
- strengthening of the innovation potential of enterprises,
- creation of an innovation-oriented legal and general societal framework,
- supporting the banking sector in financing innovation-orientated and export-orientated business
- development of bilateral and multilateral strategic alliances,
- establishment of ties between the Slovenian R&D sector and foreign industry,
- accelerated development of professional education and the education of adults,
- protection of industrial and intellectual property.

An increase of total invested assets in R&D to about 2.5% of GDP by the year 2000 is planned (of this, half is to be obtained from public sources, with the remainder to come from the private sector). Regarding the development of technology, Slovenia is one of the most technologically advanced in Central Europe and has a well-developed research infrastructure. This has led to a significant growth in the export of high-tech goods. There is also a continued emphasis on the development of R&D across a wide field which is leading to the foundation and construction of technology parks (high -tech business incubators), technology centres (technology-transfer units within public R&D institutions) and small private enterprise centres for research.

R&D Human Potential

There are about 750 R&D groups in the public and private sector, of which 102 research groups are at 17 government (national) research institutes, 340 research groups are at universities and 58 research groups are at medical institutions. The remaining R&D groups are located in business enterprises (175 R&D groups) or are run by about 55 public and private non-profit research organizatios.

According to the data of the Ministry of Science and Technology there are about 7000 researchers in Slovenia. The majority (43%) are lecturers working at the two universities, 15% of researchers are employed at government (national) research institutes, 22% at other institutions and 20% in research and development departments of business enterprises.

JOŽEF STEFAN INSTITUTE

Jožef Stefan (1835-1893) was one of the most prominent physicists of the 19th century. Born to Slovene parents, he obtained his Ph.D. at Vienna University, where he was later Director of the Physics Institute, Vice-President of the Vienna Academy of Sciences and a member of several scientific institutions in Europe. Stefan explored many areas in hydrodynamics, optics, acoustics, electricity, magnetism and the kinetic theory of gases. Among other things, he originated the law that the total radiation from a black body is proportional to the 4th power of its absolute temperature, known as the Stefan–Boltzmann law.

The Jožef Stefan Institute (JSI) is the leading independent scientific research institution in Slovenia, covering a broad spectrum of fundamental and applied research in the fields of physics, chemistry and biochemistry, electronics and information science, nuclear science technology, energy research and environmental science.

The Jožef Stefan Institute (JSI) is a research organisation for pure and applied research in the natural sciences and technology. Both are closely interconnected in research departments composed of different task teams. Emphasis in basic research is given to the development and education of young scientists, while applied research and development serve for the transfer of advanced knowledge, contributing to the development of the national economy and society in general.

At present the Institute, with a total of about 700 staff, has 500 researchers, about 250 of whom are postgraduates, over 200 of whom have doctorates (Ph.D.), and around 150 of whom have permanent professorships or temporary teaching assignments at the Universities.

In view of its activities and status, the JSI plays the role of a national institute, complementing the role of the universities and bridging the gap between basic science and applications.

Research at the JSI includes the following major fields: physics; chemistry; electronics, informatics and computer sciences; biochemistry; ecology; reactor technology; applied mathematics. Most of the activities are more or less closely connected to information sciences, in particular computer sciences, artificial intelligence, language and speech technologies, computer-aided design, computer architectures, biocybernetics and robotics, computer automation and control, professional electronics, digital communications and networks, and applied mathematics.

The Institute is located in Ljubljana, the capital of the independent state of **Slove**nia (or S♡nia). The capital today is considered a crossroad between East, West and Mediterranean Europe, offering excellent productive capabilities and solid business opportunities, with strong international connections. Ljubljana is connected to important centers such as Prague, Budapest, Vienna, Zagreb, Milan, Rome, Monaco, Nice, Bern and Munich, all within a radius of 600 km.

In the last year on the site of the Jožef Stefan Institute, the Technology park "Ljubljana" has been proposed as part of the national strategy for technological development to foster synergies between research and industry, to promote joint ventures between university bodies, research institutes and innovative industry, to act as an incubator for high-tech initiatives and to accelerate the development cycle of innovative products.

At the present time, part of the Institute is being reorganized into several high-tech units supported by and connected within the Technology park at the Jožef Stefan Institute, established as the beginning of a regional Technology park "Ljubljana". The project is being developed at a particularly historical moment, characterized by the process of state reorganisation, privatisation and private initiative. The national Technology Park will take the form of a shareholding company and will host an independent venture-capital institution.

The promoters and operational entities of the project are the Republic of Slovenia, Ministry of Science and Technology and the Jožef Stefan Institute. The framework of the operation also includes the University of Ljubljana, the National Institute of Chemistry, the Institute for Electronics and Vacuum Technology and the Institute for Materials and Construction Research among others. In addition, the project is supported by the Ministry of Economic Relations and Development, the National Chamber of Economy and the City of Ljubljana.

Jožef Stefan Institute
Jamova 39, 1000 Ljubljana, Slovenia
Tel.:+386 1 4773 900, Fax.:+386 1 219 385
Tlx.:31 296 JOSTIN SI
WWW: http://www.ijs.si
E-mail: matjaz.gams@ijs.si
Contact person for the Park: Iztok Lesjak, M.Sc.
Public relations: Natalija Polenec

INFORMATICA

AN INTERNATIONAL JOURNAL OF COMPUTING AND INFORMATICS

INVITATION, COOPERATION

Submissions and Refereeing

Please submit three copies of the manuscript with good copies of the figures and photographs to one of the editors from the Editorial Board or to the Contact Person. At least two referees outside the author's country will examine it, and they are invited to make as many remarks as possible directly on the manuscript, from typing errors to global philosophical disagreements. The chosen editor will send the author copies with remarks. If the paper is accepted, the editor will also send copies to the Contact Person. The Executive Board will inform the author that the paper has been accepted, in which case it will be published within one year of receipt of e-mails with the text in Informatica LaTeX format and figures in `.eps` format. The original figures can also be sent on separate sheets. Style and examples of papers can be obtained by e-mail from the Contact Person or from FTP or WWW (see the last page of Informatica).

Opinions, news, calls for conferences, calls for papers, etc. should be sent directly to the Contact Person.

QUESTIONNAIRE

☐ Send Informatica free of charge

☐ Yes, we subscribe

Please, complete the order form and send it to Dr. Rudi Murn, Informatica, Institut Jožef Stefan, Jamova 39, 61111 Ljubljana, Slovenia.

Since 1977, Informatica has been a major Slovenian scientific journal of computing and informatics, including telecommunications, automation and other related areas. In its 16th year (more than five years ago) it became truly international, although it still remains connected to Central Europe. The basic aim of Informatica is to impose intellectual values (science, engineering) in a distributed organisation.

Informatica is a journal primarily covering the European computer science and informatics community - scientific and educational as well as technical, commercial and industrial. Its basic aim is to enhance communications between different European structures on the basis of equal rights and international refereeing. It publishes scientific papers accepted by at least two referees outside the author's country. In addition, it contains information about conferences, opinions, critical examinations of existing publications and news. Finally, major practical achievements and innovations in the computer and information industry are presented through commercial publications as well as through independent evaluations.

Editing and refereeing are distributed. Each editor can conduct the refereeing process by appointing two new referees or referees from the Board of Referees or Editorial Board. Referees should not be from the author's country. If new referees are appointed, their names will appear in the Refereeing Board.

Informatica is free of charge for major scientific, educational and governmental institutions. Others should subscribe (see the last page of Informatica).

ORDER FORM – INFORMATICA

Name: .....

Title and Profession (optional): .....

.....

Home Address and Telephone (optional): .....

.....

Office Address and Telephone (optional): .....

.....

E-mail Address (optional): .....

Signature and Date: .....

Informatica WWW:

http://ai.ijs.si/informatica/
http://orca.st.usm.edu/informatica/

Referees:

Witold Abramowicz, David Abramson, Adel Adi, Kenneth Aizawa, Suad Alagić, Mohamad Alam, Dia Ali, Alan Aliu, Richard Amoroso, John Anderson, Hans-Jurgen Appelrath, Vladimir Bajič, Grzegorz Bartoszewicz, Catriel Beeri, Daniel Beech, Fevzi Belli, Francesco Bergadano, Istvan Berkeley, Azer Bestavros, Andraž Bežek, Balaji Bharadwaj, Ralph Bisland, Jacek Blazewicz, Laszlo Boeszoermenyi, Damjan Bojadžijev, Jeff Bone, Ivan Bratko, Jerzy Brzezinski, Marian Bubak, Leslie Burkholder, Frada Burstein, Wojciech Buszkowski, Rajkumar Bvyya, Netiva Caftori, Jason Ceddia, Ryszard Choras, Wojciech Cellary, Wojciech Chybowski, Andrzej Ciepielewski, Vic Ciesielski, David Cliff, Maria Cobb, Travis Craig, Noel Craske, Matthew Crocker, Tadeusz Czachorski, Milan Češka, Honghua Dai, Deborah Dent, Andrej Dobnikar, Sait Dogru, Georg Dorfner, Ludoslaw Drelichowski, Matija Drobnič, Maciej Drozdowski, Marek Druzdzel, Jozo Dujmović, Pavol Ďuriš, Johann Eder, Hesham El-Rewini, Warren Fergusson, Pierre Flener, Wojciech Fliegner, Vladimir A. Fomichov, Terrence Forgarty, Hans Fraaije, Hugo de Garis, Eugeniusz Gatnar, James Geller, Michael Georgiopolus, Jan Goliński, Janusz Gorski, Georg Gottlob, David Green, Herbert Groiss, Inman Harvey, Elke Hochmueller, Jack Hodges, Rod Howell, Tomáš Hruška, Don Huch, Alexey Ippa, Ryszard Jakubowski, Piotr Jedrzejowicz, A. Milton Jenkins, Eric Johnson, Polina Jordanova, Djani Juričič, Sabhash Kak, Li-Shan Kang, Orlando Karam, Roland Kaschek, Jacek Kierzenka, Jan Kniat, Stavros Kokkotos, Kevin Korb, Gilad Koren, Henryk Krawczyk, Ben Kroese, Zbyszko Krolikowski, Benjamin Kuipers, Matjaž Kukar, Aarre Laakso, Phil Laplante, Bud Lawson, Ulrike Leopold-Wildburger, Joseph Y-T. Leung, Barry Levine, Xuefeng Li, Alexander Linkevich, Raymond Lister, Doug Locke, Peter Lockeman, Matija Lokar, Jason Lowder, Kim Teng Lua, Andrzej Małachowski, Bernardo Magnini, Peter Marcer, Andrzej Marciniak, Witold Marciszewski, Vladimir Marik, Jacek Martinek, Tomasz Maruszewski, Florian Matthes, Daniel Memmi, Timothy Menzies, Dieter Merkl, Zbigniew Michalewicz, Gautam Mitra, Roland Mittermeir, Madhav Moganti, Reinhard Moller, Tadeusz Morzy, Daniel Mossé, John Mueller, Hari Narayanan, Rance Necaise, Elzbieta Niedzielska, Marian Niedq'zwiedziński, Jaroslav Nieplocha, Jerzy Nogieć, Stefano Nolfi, Franc Novak, Antoni Nowakowski, Adam Nowicki, Tadeusz Nowicki, Hubert Österle, Wojciech Olejniczak, Jerzy Olszewski, Cherry Owen, Mieczyslaw Owoc, Tadeusz Pankowski, William C. Perkins, Warren Persons, Mitja Peruš, Stephen Pike, Niki Pissinou, Aleksander Pivk, Ullin Place, Gustav Pomberger, James Pomykalski, Dimithu Prasanna, Gary Preckshot, Dejan Rakovič, Cveta Razdevšek Pučko, Ke Qiu, Michael Quinn, Gerald Quirchmayer, Luc de Raedt, Ewaryst Rafajlowicz, Sita Ramakrishnan, Wolf Rauch, Peter Rechenberg, Felix Redmill, David Robertson, Marko Robnik, Ingrid Russel, A.S.M. Sajeev, Bo Sanden, Vivek Sarin, Iztok Savnik, Walter Schempp, Wolfgang Schreiner, Guenter Schmidt, Heinz Schmidt, Dennis Sewer, Zhongzhi Shi, William Spears, Hartmut Stadtler, Olivero Stock, Janusz Stokłosa, Przemysław Stpiczyński, Andrej Stritar, Maciej Stroinski, Tomasz Szmuc, Zdzislaw Szyjewski, Jure Šilc, Metod Škarja, Jiří Šlechta, Chew Lim Tan, Zahir Tari, Jurij Tasič, Piotr Teczynski, Stephanie Teufel, Ken Tindell, A Min Tjoa, Wieslaw Traczyk, Roman Trobec, Marek Tudruj, Andrej Ule, Amjad Umar, Andrzej Urbanski, Marko Uršič, Tadeusz Usowicz, Elisabeth Valentine, Kanonkluk Vanapipat, Alexander P. Vazhenin, Zygmunt Vetulani, Olivier de Vel, John Weckert, Gerhard Widmer, Stefan Wrobel, Stanislaw Wrycza, Janusz Zalewski, Damir Zazula, Yanchun Zhang, Zonling Zhou, Robert Zorc, Anton P. Železnikar

EDITORIAL BOARDS, PUBLISHING COUNCIL

Informatica is a journal primarily covering the European computer science and informatics community; scientific and educational as well as technical, commercial and industrial. Its basic aim is to enhance communications between different European structures on the basis of equal rights and international refereeing. It publishes scientific papers accepted by at least two referees outside the author's country. In addition, it contains information about conferences, opinions, critical examinations of existing publications and news. Finally, major practical achievements and innovations in the computer and information industry are presented through commercial publications as well as through independent evaluations.

Editing and refereeing are distributed. Each editor from the Editorial Board can conduct the refereeing process by appointing two new referees or referees from the Board of Referees or Editorial Board. Referees should not be from the author's country. If new referees are appointed, their names will appear in the list of referees. Each paper bears the name of the editor who appointed the referees. Each editor can propose new members for the Editorial Board or referees. Editors and referees inactive for a longer period can be automatically replaced. Changes in the Editorial Board are confirmed by the Executive Editors.

The coordination necessary is made through the Executive Editors who examine the reviews, sort the accepted articles and maintain appropriate international distribution. The Executive Board is appointed by the Society Informatika. Informatica is partially supported by the Slovenian Ministry of Science and Technology.

Each author is guaranteed to receive the reviews of his article. When accepted, publication in Informatica is guaranteed in less than one year after the Executive Editors receive the corrected version of the article.

Volume 24 Number 3 September 2000 ISSN 0350-5596

Informatica

An International Journal of Computing and Informatics

Introduction		285
Attribute Grammars as Record Calculus — A Structure-Oriented Denotational Semantics of Attribute Grammars by Using Cardelli's Record Calculus	K. Gondow T. Katayama	287
Reference Attributed Grammars	G. Hedin	301
Multiple Attribute Grammar Inheritance	M. Mernik M. Lenič E. Avdičaušević V. Žumer	319
First-class Attribute Grammars	O. de Moor K. Backhouse S.D. Swierstra	329
Equational Semantics	L. Correnson	343
Two-dimensional Approximation Coverage	J. Harm R. Lämmel	355
A Multi-phase Parallel Algorithm for the Eigenelements Problem	A. Benaini D. Laiymani	371
DD-Mod: A library for distributed programming	J.M. Milán-Franco R. Jiménez-Peris M. Patiño-Martínez	379
A Technique for Computing Watermarks from Digital Images	C.-C. Chang C.-S. Tsai	391
A Program–Algebraic Approach to Eliminating Common Subexpressions	J.M. Boyle R.D. Resler	397
An Assessment of Incomplete-LU Preconditioners for Nonsymmetric Linear Systems	J.R. Gilbert S. Toledo	409
Reports and Announcements		427