LIBER AMICORUM

Paul Klint
25 jaar SMC/CWI

Dat is dus heel interessant

Dat is dus heel interessant

Liber Amicorum voor Paul Klint

Uitgave van het CWI ter gelegenheid van het 25-jarig dienstverband van Paul Klint in september 1997.

Bijeengebracht door: Mieke Bruné, Arie van Deursen en Jan Heering
Omslag: Tobias Baanders
Druk: Jan Schipper en Jos van der Werf

Table of Contents

Paul Likes Executable Specifications 1

Discrete-Time Process Algebra with Empty Process
J.C.M. Baeten & J.J. Vereijken 5

Paul and the Others: 1 + 55
J.W. de Bakker 25

Process Algebra Primitives for File Transfer
Jan A. Bergstra & Alban Ponse 33

Looking Back
Inge Bethke, Jan Willem Klop & Roel de Vrijer 43

The Missing Links
Mark van den Brand 55

The Leap Year Problem
Arie van Deursen 61

Een Paul-Klint-Go-Probleem
Niek van Diepen 75

'Oġm ··· 'Oġm'
't din 79

The Origins of Error Messages
T.B. Dinesh & Frank Tip 81

ASF+SDF voor Marsbewoners
Peter van Emde Boas 101

Why Men (and Octopuses) Cannot Juggle a Four Ball Cascade
A. Engels & S. Mauw 109

Let's Accept Rejects, But Only After Repairs
Jan van Eijck 117

Beste Paul, ...
Teus Hagen & Ricky Ras 129

SEMANOL, BASIS/1-12 en ASF+SDF
Jan Heering 131

Kwaliteit van een Hoogleraar
Paul Hendriks 135

Sympa!
Wilco Koorn 143

Ubiquitous Computational Steering
Robert van Liere & Jurriaan D. Mulder 145

The M* Problem
Lambert Meertens 149

Lieve Paul, ...
Emma van der Meulen, Casper Dik & Josephine 151

Space/Time Efficiency and Redundancy in Terms
Pieter Olivier 153

TOOLBUS
Gerard van Oortmerssen 159

Als een van de velen ...
Jan Rekers 161

Meersoortige Algebra met Mogelijk Lege Soorten: Vrije Uitbreidingen en de *HSP*-Stelling
P.H. Rodenburg 163

Regular Expressions and Coinduction
Jan Rutten 189

Evolution of COBOL Software in Dutch IT Industry
Alex Sellink & Chris Verhoef 199

ToolBus and Tree Substitution
Yerhpmuh Trebor Sretlaw 209

Wat Deed Paul de Dag Nadat Kennedy Vermoord Was?
Joke Sterringa 213

From Context-Free Grammars with Priorities to Character Class Grammars
Eelco Visser 217

Paul Likes Executable Specifications,

so I can't resist to offer him a couple of mine. They are all written in a programming language Alma-0, an extension of a subset of Modula-2 by a grand total of nine features that require seven new keywords. Alma-0 can be run on all Unix platforms. For more information see `http://www.cwi.nl/alma`.

In what follows, in order not to intimidate numerous readers, I shall use only the following four features of Alma-0:

- boolean expressions can be used as statements,
- statement sequences can be used as boolean expressions,
- the `SOME` statement, a "dual" to the `FOR` statement;

  ```
  SOME i := 1 TO N DO S END; T
  ```

  It attempts to find an `i` in the range `[1..n]` for which the remainder of the program succeeds;
- double use of equality — as a comparison test and as an assignment. The latter happens when one side is an uninitialized variable and the other an expression with a known value.

What shall we start with? How about the problem of specifying a permutation? So here is the official definition:

> A sequence $\mathtt{out}_1, \ldots, \mathtt{out}_\mathtt{N}$ is a permutation of $\mathtt{in}_1, \ldots, \mathtt{in}_\mathtt{N}$ if for some function π from $[1..\mathtt{N}]$ onto itself we have
>
> $$\mathtt{out}_1, \ldots, \mathtt{out}_\mathtt{N} = \mathtt{in}_{\pi(1)}, \ldots, \mathtt{in}_{\pi(\mathtt{N})}.$$

In Alma-0 this can be written as follows:

```
TYPE Vector = ARRAY [1..N] OF INTEGER;

PROCEDURE Permutation(VAR in,out: Vector);
VAR pi: Vector;
    i,j: INTEGER;
BEGIN
  FOR i := 1 TO N DO
    SOME j := 1 TO N DO
      pi[j] = i
    END
  END; (* pi is a function from 1..N onto itself *)
  FOR i := 1 TO N DO
    out[i] = in[pi[i]]
  END  (* out is obtained by applying pi to the indices of in *)
END Permutation;
```

Well, that was not difficult.

I guess we are now ready for a more ambitious example. So let us turn to classics. My eye fell on the problem of finding the lexicographically next permutation, discussed in Dijkstra (1976).

As somebody never returned my copy of this book, [1] I have to resort to its translation to Polish that I did not lend to anybody (because it bears the dedication of the author, in English):

> Wiadomo, że uporządkowany ciąg wartości c(1), c(2), ..., c(N) stanowi pewną permutację liczb 1, ..., N, na pewno jednak **nie** ostatnią w porządku "alfabetycznym", tj. nie N, N − 1, ..., 1. Zadaniem bloku wewnętrznego, którego mamy napisać, będzie przekształcenie ciągu c(1), c(2), ..., c(N) w jego bezpośredni następnik w porządku alfabetycznym.

To specify the problem we need first to define the lexicographic ordering. We use the following definition: the sequence $a_1, \ldots, a_N$ precedes lexicographically $b_1, \ldots, b_N$ if some i in the range [1..N] exists such that for all j in [1..i-1] we have a[j] = b[j] and a[i] < b[i].

[1] I hope that the person who borrowed it after having read this note will promptly return the book to me. I left the bookmark, if I remember correctly, on page 65 where the author discusses the youngest of his two dogs.

In Alma-0 we write it as follows:

```
PROCEDURE Lex(a,b: Vector);
VAR i: INTEGER;
BEGIN
  NOT FOR i:= 1 TO N DO a[i] = b[i] END;
  a[i] < b[i]
END Lex;
```

Now b is the lexicographically next permutation of a if

- b is a permutation of a,
- a precedes b lexicographically,
- no permutation exists that is lexicographically between a and b.

This leads us to the following procedure **Next** that uses an auxiliary procedure **Between**:

```
PROCEDURE Between(a,b: Vector);
VAR c: Vector;
BEGIN
  Permutation(a,c);
  Lex(a,c);
  Lex(c,b)
END Between;

PROCEDURE Next(VAR a,b: Vector);
BEGIN
  Permutation(a,b);
  Lex(a,b);
  NOT Between(a,b)
END Next;
```

This concludes the presentation of the program. Note that it does not use any assignment. Using it I could confirm the example given in Dijkstra's book, namely that `1 4 6 2 9 7 3 5 8` is the lexicographically next permutation of `1 4 6 2 9 5 8 7 3`.

Now, let us try to go beyond Dijkstra's book and find the lexicographically previous permutation. Thanks to the fact that Alma-0 supports *inverse engineering* (the definition left to the reader) we can use for this purpose the same procedure `Next` — it just suffices to pass now the given permutation as the second parameter. This way I succeeded in establishing that `1 4 6 2 9 5 8 3 7` is the lexicographically previous permutation of `1 4 6 2 9 5 8 7 3`, which was an open problem.

My last example dealt with executable specifications of the functioning of CWI after the "kanteling" operation — a research that is undoubtedly a societally relevant occupation. The specifications ran for 22 pages. Unfortunately, Arie asked me politely to drop them, so I complied. The program is still running.

Let me conclude with a more technical remark. The problem deals with the rules for professional soccer that were introduced July 1 this year (see NRC Handelsblad, August 19th). I started to formalize them one by one until I encountered the following one:

> Zogenaamde wielerbroeken moeten dezelfde hoofdkleur hebben als de broeken waaronder zij worden gedragen.

I did not succeed in specifying this rule in Alma-0. Perhaps it would be possible to do this in ASF+SDF? If yes, I am looking forward to a solution.

My inborn modesty does not allow me to sign this note.

References

Dijkstra, E. W. (1976), *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J.

Discrete-Time Process Algebra with Empty Process

J. C. M. Baeten and J. J. Vereijken

Department of Computing Science, Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

http://www.win.tue.nl/cs/fm/{josb,janjoris}/
josb@win.tue.nl, janjoris@acm.org

Abstract

This article is about modularity in theory design: it combines the empty process and discrete time in the theory of process algebra.

Note: This article is an extended abstract of Chapter 6 of the dissertation of the second author (VEREIJKEN [20]). It is dedicated to Paul Klint, who knows many things about modularity in system design and system decomposition.

1 Introduction

One of the main features of process algebra is its modularity: it is relatively easy to add features for specific purposes, and, given two of such extensions, it is often straightforward to combine them into one process algebra.

In the past, the feature of the *empty process*, the process that does nothing, and terminates successfully (in contrast with the *deadlock process*, which does nothing, and terminates unsuccessfully), has been studied extensively (BAETEN AND VAN GLABBEEK [7], KOYMANS AND VRANCKEN [15], and VRANCKEN [21, 22]).

On the other hand, recently there has been described an elegant and coherent way to incorporate *discrete-time extensions* into the process-algebra framework (BAETEN AND BERGSTRA [3, 4, 5], BAETEN AND RENIERS [9], RENIERS AND VEREIJKEN [19]), and a case study has been made (BOS AND RENIERS [11]).

Given the fact that both extensions, empty process and discrete time, have been researched very well, it seemed reasonable to study how these two extensions go together. This is especially important, as the empty process does have great advantages in the specification of protocols. For example, a stack over an infinite data type can only be specified in finitely many equations (three, to be exact), when we have the empty process at our disposal (KOYMANS AND VRANCKEN [15]). Furthermore, the empty process is needed to give a process-algebra based semantics to the specification languages SDL (see ITU-T [12]) and MSC (see ITU-TS [13, 14], and the dissertation of RENIERS [18]), and hence a timed empty process is needed to give such a semantics to timed variants of these languages.

But, as the introduction of the empty process has historically often been viewed as troublesome, the modular combination of it with other features has hardly been studied. In this contribution, we remedy this situation for discrete-time process algebra with relative timing.

2 Design Goals for the Discrete-Time Empty Process

Often, when extending an already existing process algebra with some additional constant or operator, one has a lot of "maneuvering room" to make choices regarding the exact implementation and the fine details of its behavior. For example, a very broad class of "deadlock-like" processes has been described in the literature. To name a few variants (in order of increasing degree of catastrophe): "classic" (δ, see e.g. BAETEN AND WEIJLAND [10]), "immediate" ($\dot{\delta}$, see BAETEN AND BERGSTRA [4, 5]), "inconsistent state" ($\perp$, see BAETEN AND BERGSTRA [6]), and "true zero" (0, see BAETEN AND BERGSTRA [1, 2]). A similar observation holds for the implementation of the concept "choice": half a dozen versions, ranging from completely deterministic to completely non-deterministic have been described.

It appears that the concept "empty process" is different; it leaves little room for differing implementations.

When we started thinking about the process that "does nothing" and terminates successfully (in the context of discrete-time process algebra), we had three design goals in mind for the empty process:

(i). it should be a unit element with respect to the $\cdot$ and $\|$ operators,

(ii). it ought not to destroy the commutativity and associativity of the $\|$ and $|$ operators,

(iii). it ought not to destroy the weak time-factorization property.

These properties we deemed essential; violating one of them would render our empty process useless. And, as it turned out, these goals very much restricted our freedom in choosing our definitions. In this chapter, we will motivate our choices on the grounds of the above design goals.

Finally, we had a few other design goals of lesser importance; we wanted that the existing "smaller" process algebras (without empty process, without time, or without either) could be embedded in the new process algebras in a simple way (i.e., in the terminology of BAETEN AND BERGSTRA [2], the old process algebras should be *Subalgebras of a Reduced Model* (SRM's) of the new process algebras).

3 Discrete-Time Process Algebra

We start by recalling the definition of the simplest discrete-time process algebra without empty process, $\mathrm{BPA}^{-}_{\mathrm{drt}}$-ID, from BAETEN AND BERGSTRA [3, 4, 5], BAETEN AND RENIERS [9], RENIERS AND VEREIJKEN [19]. It concerns a discrete-time process algebra, thus time is divided into *slices*. Further, this is a *relative-time* process algebra, so the timing of an action is measured with respect to the previous action. The name of this algebra has a superscript minus, since there are no delayable actions, and a suffix -ID, since the immediate deadlock constant $\dot{\delta}$ is not part of the signature.

A parameter of this algebra (and all algebras to come) is a given (finite) set of atomic actions A. We write $A_\delta = A \cup \{\delta\}, A_\sigma = A \cup \{\sigma\}$.

Definition 3.1.1 (Signature of $\mathrm{BPA}^{-}_{\mathrm{drt}}$-ID)
The signature of $\mathrm{BPA}^{-}_{\mathrm{drt}}$-ID consists of the *undelayable actions* $\{\underline{\underline{a}} | a \in A\}$, the *undelayable deadlock constant* $\underline{\underline{\delta}}$, the *time-unit delay operator* σ, the *alternative composition operator* $+$, the *sequential composition operator* $\cdot$, and the *"now" operator* ν.

Definition 3.1.2 (Axioms of $\mathrm{BPA}^{-}_{\mathrm{drt}}$-ID)
The process algebra $\mathrm{BPA}^{-}_{\mathrm{drt}}$-ID is axiomatized by the axioms of BPA (A1–A5), and in addition Axioms DRT1-DRT4 and DCS1-DCS4, shown in Table 1: $\mathrm{BPA}^{-}_{\mathrm{drt}}$-ID = A1-A5 + DRT1-DRT4 + DCS1-DCS4.

In this axiom table (as in all axiom tables to come) we have $a \in A_\delta$.

$x + y = x + y$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y) \cdot z = x \cdot z + y \cdot z$	A4
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5
$x + \underline{\underline{\delta}} = x$	DRT1
$\underline{\underline{\delta}} \cdot x = \underline{\underline{\delta}}$	DRT2
$\sigma(x) + \sigma(y) = \sigma(x + y)$	DRT3
$\sigma(x) \cdot y = \sigma(x \cdot y)$	DRT4
$\nu(\underline{\underline{a}}) = \underline{\underline{a}}$	DCS1
$\nu(x + y) = \nu(x) + \nu(y)$	DCS2
$\nu(x \cdot y) = \nu(x) \cdot y$	DCS3
$\nu(\sigma(x)) = \underline{\underline{\delta}}$	DCS4

Table 1: Axioms of $\mathrm{BPA}^{-}_{\mathrm{drt}}$-ID.

4 The Empty Process with Undelayable Actions

We introduce the process algebras $BPA^-_{drt,\varepsilon}$-ID, $PA^-_{drt,\varepsilon}$-ID, and $ACP^-_{drt,\varepsilon}$-ID that contain the empty process. These are all based on BPA^-_{drt}-ID, and hence do not contain delayable actions or the immediate deadlock.

4.1 $BPA^-_{drt,\varepsilon}$-ID

In this section, we define the process algebra $BPA^-_{drt,\varepsilon}$-ID, which is basically the process algebra BPA^-_{drt}-ID of the previous Section, extended with the empty process (indicated by the subscript "$_\varepsilon$").

Definition 4.1.1 (Signature of $BPA^-_{drt,\varepsilon}$-ID)
The signature of $BPA^-_{drt,\varepsilon}$-ID consists of the *undelayable actions* $\{\underline{\underline{a}} | a \in A\}$, the *undelayable deadlock constant* $\underline{\underline{\delta}}$, the *time-unit delay constant* $\underline{\underline{\sigma}}$, the *undelayable empty process constant* $\underline{\underline{\varepsilon}}$, the *alternative composition operator* $+$, the *sequential composition operator* $\cdot$, and the *"now" operator* ν.

Remark 4.1.2 (The $\underline{\underline{\varepsilon}}$ Constant and the $\underline{\underline{\sigma}}$ Constant)
The *undelayable empty process* $\underline{\underline{\varepsilon}}$ in the signature of $BPA^-_{drt,\varepsilon}$-ID has the intuitive meaning of doing nothing, and then terminating successfully. As such, it is a proper unit element of the sequential composition: $\underline{\underline{\varepsilon}} \cdot x = x \cdot \underline{\underline{\varepsilon}} = x$.

Now that we have a unit element, we do not need the time unit delay *operator* anymore. This is because we can replace it by the time-unit delay *constant* $\underline{\underline{\sigma}}$ (intuitively: $\underline{\underline{\sigma}} = \sigma(\underline{\underline{\varepsilon}})$), the process that can move on to the following time-slice, and terminate there. The process formerly expressed as $\sigma(x)$ can now be represented by $\underline{\underline{\sigma}} \cdot x$, as, intuitively, $\sigma(x) = \sigma(\underline{\underline{\varepsilon}} \cdot x) = \sigma(\underline{\underline{\varepsilon}}) \cdot x = \underline{\underline{\sigma}} \cdot x$.

Definition 4.1.3 (Axioms of $BPA^-_{drt,\varepsilon}$-ID)
The process algebra $BPA^-_{drt,\varepsilon}$-ID is axiomatized by the axioms A1–A5 of BPA and the axioms DRT1–DRT2 given in Table 1, and Axioms DRTE1–DRTE2, TF, and DCSE1–DCSE4 shown in Table 2: $BPA^-_{drt,\varepsilon}$-ID = A1–A5 + DRT1–DRT2 + DRTE1–DRTE2 + TF + DCSE1–DCSE4.

The undelayable deadlock behaves as before. Axioms DRTE1 and DRTE2 express that the undelayable empty process is a unit element with respect to the sequential composition.

When we compare $BPA^-_{drt,\varepsilon}$-ID with BPA^-_{drt}-ID, the presence of the undelayable empty process allows for two simplifications in the axioms. First, to express the weak time-factorization property associated with $\underline{\underline{\sigma}}$, formerly done by Axiom DRT3, we now use Axiom TF. Secondly, the purpose of Axiom DRT4 disappears, as it becomes a special case of Axiom A5, the associativity of sequential composition: $(\underline{\underline{\sigma}} \cdot x) \cdot y = \underline{\underline{\sigma}} \cdot (x \cdot y)$.

Finally, Axioms DCSE1–DCSE4 express the properties of the "now" operator in a context with the undelayable empty process. Note that we do not anymore have that $\nu(x \cdot y) = \nu(x) \cdot y$, as that would lead to $\nu(\underline{\underline{\sigma}}) = \nu(\underline{\underline{\varepsilon}} \cdot \underline{\underline{\sigma}}) = \nu(\underline{\underline{\varepsilon}}) \cdot \underline{\underline{\sigma}} = \underline{\underline{\varepsilon}} \cdot \underline{\underline{\sigma}} = \underline{\underline{\sigma}}$, which is of course undesirable. Otherwise,

$x \cdot \underline{\underline{\varepsilon}} = x$	DRTE1
$\underline{\underline{\varepsilon}} \cdot x = x$	DRTE2
$\underline{\underline{\sigma}} \cdot x + \underline{\underline{\sigma}} \cdot y = \underline{\underline{\sigma}} \cdot (x + y)$	TF
$\nu_{\mathrm{rel}}(\underline{\underline{\varepsilon}}) = \underline{\underline{\varepsilon}}$	DCSE1
$\nu_{\mathrm{rel}}(x + y) = \nu_{\mathrm{rel}}(x) + \nu_{\mathrm{rel}}(y)$	DCSE2
$\nu_{\mathrm{rel}}(\underline{\underline{a}} \cdot x) = \underline{\underline{a}} \cdot x$	DCSE3
$\nu_{\mathrm{rel}}(\underline{\underline{\sigma}} \cdot x) = \underline{\underline{\delta}}$	DCSE4

Table 2: Additional axioms for $\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}$-ID.

Axioms DCSE1–DCSE4 behave very much like their $\mathrm{BPA}^{-}_{\mathrm{drt}}$-ID counterparts DCS1–DCS4 from Table 1.

Definition 4.1.4 (Semantics of $\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}$-ID)
The semantics of $\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}$-ID are given by the term-deduction system named $T(\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}\text{-ID})$, induced by the deduction rules for $\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}$-ID shown in Table 3.

In this table of operational rules (as in all such tables) we have $a \in A$.

We give the semantics in the format of so-called structured operational semantics. Here, we define binary relations $\xrightarrow{a}$, for each atomic action $a \in A$, denoting the execution of an action, a unary predicate $\downarrow$ denoting a termination option, and binary relation $\xrightarrow{\sigma}$ denoting a time step, by structural induction. What makes the definition of these rules subtle, is the requirement of weak time factorization or time determinism: a term may have at most one outgoing σ-step.

The subtlety shows up mostly in the deduction rules for the sequential composition with respect to time transitions, shown in Table 3. We will try to provide the intuition behind them with an example.

Consider the expression $(\underline{\underline{\sigma}} \cdot \underline{\underline{a}} + \underline{\underline{\varepsilon}}) \cdot (\underline{\underline{\sigma}} \cdot \underline{\underline{b}})$. Algebraically, we have:

$$\begin{aligned}
(\underline{\underline{\sigma}} \cdot \underline{\underline{a}} + \underline{\underline{\varepsilon}}) \cdot (\underline{\underline{\sigma}} \cdot \underline{\underline{b}}) &= \underline{\underline{\sigma}} \cdot \underline{\underline{a}} \cdot \underline{\underline{\sigma}} \cdot \underline{\underline{b}} + \underline{\underline{\varepsilon}} \cdot \underline{\underline{\sigma}} \cdot \underline{\underline{b}} \\
&= \underline{\underline{\sigma}} \cdot \underline{\underline{a}} \cdot \underline{\underline{\sigma}} \cdot \underline{\underline{b}} + \underline{\underline{\sigma}} \cdot \underline{\underline{b}} \\
&= \underline{\underline{\sigma}} \cdot (\underline{\underline{a}} \cdot \underline{\underline{\sigma}} \cdot \underline{\underline{b}} + \underline{\underline{b}})
\end{aligned}$$

therefore, in our model we should have $(\underline{\underline{\sigma}} \cdot \underline{\underline{a}} + \underline{\underline{\varepsilon}}) \cdot (\underline{\underline{\sigma}} \cdot \underline{\underline{b}}) \xrightarrow{\sigma} \underline{\underline{a}} \cdot \underline{\underline{\sigma}} \cdot \underline{\underline{b}} + \underline{\underline{b}}$ and in particular we should *not* have $(\underline{\underline{\sigma}} \cdot \underline{\underline{a}} + \underline{\underline{\varepsilon}}) \cdot (\underline{\underline{\sigma}} \cdot \underline{\underline{b}}) \xrightarrow{\sigma} \underline{\underline{a}} \cdot \underline{\underline{\sigma}} \cdot \underline{\underline{b}}$, as such a transition would lose the option to do just a $\underline{\underline{b}}$, while the principle of weak time factorization mentioned before explicitly forbids that a time transition determines a choice here.

To obtain such behavior in our term-deduction system, we need four rules to define the interaction between time transitions and the sequential composition. The first two express that if x can do a σ-step to x', then $x \cdot y$

$$
\begin{array}{ccc}
\underline{\underline{a}} \xrightarrow{a} \underline{\underline{\varepsilon}} & \underline{\underline{\sigma}} \xrightarrow{\sigma} \underline{\underline{\varepsilon}} & \underline{\underline{\varepsilon}}\downarrow \\[2ex]
\dfrac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'} & \dfrac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'} & \\[3ex]
\dfrac{x\downarrow}{(x + y)\downarrow} & \dfrac{y\downarrow}{(x + y)\downarrow} & \\[3ex]
\dfrac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} & \dfrac{x\downarrow,\ y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'} & \dfrac{x\downarrow,\ y\downarrow}{(x \cdot y)\downarrow} \\[3ex]
\dfrac{x \xrightarrow{\sigma} x',\ y \xrightarrow{\sigma} y'}{x + y \xrightarrow{\sigma} x' + y'} & \dfrac{x \xrightarrow{\sigma} x',\ y \overset{\sigma}{\nrightarrow}}{x + y \xrightarrow{\sigma} x'} & \dfrac{x \overset{\sigma}{\nrightarrow},\ y \xrightarrow{\sigma} y'}{x + y \xrightarrow{\sigma} y'} \\[3ex]
\dfrac{x \xrightarrow{\sigma} x',\ x\not\downarrow}{x \cdot y \xrightarrow{\sigma} x' \cdot y} & \dfrac{x \xrightarrow{\sigma} x',\ y \overset{\sigma}{\nrightarrow}}{x \cdot y \xrightarrow{\sigma} x' \cdot y} & \dfrac{x \overset{\sigma}{\nrightarrow},\ x\downarrow,\ y \xrightarrow{\sigma} y'}{x \cdot y \xrightarrow{\sigma} y'} \\[3ex]
\dfrac{x \xrightarrow{\sigma} x',\ x\downarrow,\ y \xrightarrow{\sigma} y'}{x \cdot y \xrightarrow{\sigma} x' \cdot y + y'} & \dfrac{x \xrightarrow{a} x'}{\nu(x) \xrightarrow{a} x'} & \dfrac{x\downarrow}{\nu(x)\downarrow}
\end{array}
$$

Table 3: Deduction rules for $BPA^{-}_{drt,\varepsilon}$-ID.

can do a σ-step to $x' \cdot y$, *provided* either x does not have option to terminate, or y cannot do a σ-step. The fourth rule expresses that if this provision is not met, we get the behavior as sketched in the example above. These three rules together cover all cases where x can do a σ-step.

Remains the case that x cannot do a σ-step. If that is so, and x also does not have the option to terminate, then obviously $x \cdot y$ cannot do a σ-step, and hence there is no rule for this case. If x does have the option to terminate, and y can do a σ-step to y', then $x \cdot y$ can do a σ-step to y', as is expressed by the third rule. Finally, if x does have the option to terminate, but y cannot do a σ-step, then again $x \cdot y$ cannot do a σ-step, so there is no rule for that case either.

As a result, we have time determinism, i.e. if we can derive both $x \xrightarrow{\sigma} y$ and $x \xrightarrow{\sigma} y'$, it follows that $y \equiv y'$.

Definition 4.1.5 (Bisimulation for $BPA^{-}_{drt,\varepsilon}$-ID)
Bisimulation for $BPA^{-}_{drt,\varepsilon}$-ID is defined as follows; a binary relation R on process terms is a bisimulation iff the following transfer conditions hold for all process terms p and q:

(i). If $R^S(p,q)$ and $p \xrightarrow{u} p'$, where $u \in A \cup \{\sigma\}$, then there exists a process term q' such that $q \xrightarrow{u} q'$ and $R^S(p',q')$,

(ii). If $R^S(p,q)$ and $p\downarrow$, then $q\downarrow$.

Here, R^S denotes the symmetric closure of relation R.

Now, the bisimulation model for $\text{BPA}^-_{\text{drt},\varepsilon}$-ID is defined as usual: elements are the bisimulation equivalence classes, and the operators are defined by taking representatives.

Definition 4.1.6 (Basic Terms of $\text{BPA}^-_{\text{drt},\varepsilon}$-ID)
We define *basic terms* inductively as follows:

(i). The constant $\underline{\underline{\varepsilon}}$ is a basic term,

(ii). if $a \in A_\delta$ and t is a basic term, then $\underline{\underline{a}} \cdot t$ is a basic term,

(iii). if s and t are basic terms, then $s + t$ is a basic term,

(iv). if t is a basic term, then $\underline{\underline{\sigma}} \cdot t$ is a basic term.

Example 4.1.7 (Basic Terms of $\text{BPA}^-_{\text{drt},\varepsilon}$-ID)
The following are basic terms of $\text{BPA}^-_{\text{drt},\varepsilon}$-ID: $\underline{\underline{\delta}}\cdot\underline{\underline{\varepsilon}}, \underline{\underline{a}}\cdot\underline{\underline{\varepsilon}}, \underline{\underline{a}}\cdot(\underline{\underline{b}}\cdot\underline{\underline{c}}\cdot\underline{\underline{\varepsilon}}+\underline{\underline{d}}\cdot\underline{\underline{\varepsilon}})$. Note, however, that the following are *not* basic terms: $\underline{\underline{\delta}}, \underline{\underline{a}}, \underline{\underline{a}}\cdot(\underline{\underline{b}}+\underline{\underline{c}})$.

Theorem 4.1.8 (General Form of Basic Terms of $\text{BPA}^-_{\text{drt},\varepsilon}$-ID)
Modulo the commutativity and associativity of the +, all basic terms t of $\text{BPA}^-_{\text{drt},\varepsilon}$-ID are of the form:

$$t \equiv \sum_{i<m} \underline{\underline{a_i}} \cdot s_i + \sum_{j<n} \underline{\underline{\sigma}} \cdot u_j + \sum_{k<p} \underline{\underline{\varepsilon}}$$

for $m, n, p \in \mathcal{N}$, $m + n + p \geq 1$, $a_i \in A_\delta$, and basic terms s_i and u_j.

Lemma 4.1.9 (Representation of $\text{BPA}^-_{\text{drt},\varepsilon}$-ID Terms)
Let t be a basic term. Then either $\text{BPA}^-_{\text{drt},\varepsilon}\text{-ID} \vdash t = \nu(t)$, or there exists a basic term s such that $\text{BPA}^-_{\text{drt},\varepsilon}\text{-ID} \vdash t = \nu(t) + \underline{\underline{\sigma}} \cdot s$.

We can also define basic terms in a slightly different way, so that a basic term contains at most one $\underline{\underline{\varepsilon}}$ summand, and at most one summand of the form $\underline{\underline{\sigma}}\cdot x$. This will be useful in order to axiomatize the *left merge operator* to be treated in Section 4.2.

Theorem 4.1.10 (Elimination for $\text{BPA}^-_{\text{drt},\varepsilon}$-ID)
Let t be a closed $\text{BPA}^-_{\text{drt},\varepsilon}$-ID term. Then there is a basic term s such that $\text{BPA}^-_{\text{drt},\varepsilon}\text{-ID} \vdash s = t$.

Theorem 4.1.11 (Soundness of $\text{BPA}^-_{\text{drt},\varepsilon}$-ID)
The set of closed $\text{BPA}^-_{\text{drt},\varepsilon}$-ID terms modulo bisimulation equivalence is a model of the axioms of $\text{BPA}^-_{\text{drt},\varepsilon}$-ID.

Theorem 4.1.12 (Completeness of $\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}$-ID)
The axiom system $BPA^{-}_{drt,\varepsilon}$-ID is a complete axiomatization of the set of closed $BPA^{-}_{drt,\varepsilon}$-ID terms modulo bisimulation equivalence.

4.2 $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID

In this section, we extend the process algebra $\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}$-ID to the process algebra $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID by introducing axioms for the free merge. We add to the signature the *free merge operator* $\|$, and the *left merge operator* $⌊\!⌊$.

The process algebra named $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID is axiomatized by the axioms of $\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}$-ID given in Definition 4.1.3, and Axioms DRTEM1-DRTEM12 shown in Table 4: $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID = A1-A5 + DRT1-DRT2 + DRTE1-DRTE2 + TF + DCSE1-DCSE4 + DRTEM1-DRTEM12.

$x \parallel y = x ⌊\!⌊ y + y ⌊\!⌊ x$	DRTEM1
$\underline{\underline{a}} \cdot x ⌊\!⌊ y = \underline{\underline{a}} \cdot (x \parallel y)$	DRTEM2
$(x+y) ⌊\!⌊ z = x ⌊\!⌊ z + y ⌊\!⌊ z$	DRTEM3
$\underline{\underline{\varepsilon}} ⌊\!⌊ \underline{\underline{\varepsilon}} = \underline{\underline{\varepsilon}}$	DRTEM4
$\underline{\underline{\varepsilon}} ⌊\!⌊ \underline{\underline{a}} \cdot x = \underline{\underline{\delta}}$	DRTEM5
$\underline{\underline{\varepsilon}} ⌊\!⌊ \underline{\underline{\sigma}} \cdot x = \underline{\underline{\delta}}$	DRTEM6
$\underline{\underline{\varepsilon}} ⌊\!⌊ (x+y) = \underline{\underline{\varepsilon}} ⌊\!⌊ x + \underline{\underline{\varepsilon}} ⌊\!⌊ y$	DRTEM7
$\underline{\underline{\sigma}} \cdot x ⌊\!⌊ (\underline{\underline{a}} \cdot y + z) = \underline{\underline{\sigma}} \cdot x ⌊\!⌊ z$	DRTEM8
$\underline{\underline{\sigma}} \cdot x ⌊\!⌊ \underline{\underline{\delta}} = \underline{\underline{\delta}}$	DRTEM9
$\underline{\underline{\sigma}} \cdot x ⌊\!⌊ \underline{\underline{\varepsilon}} = \underline{\underline{\sigma}} \cdot x$	DRTEM10
$\underline{\underline{\sigma}} \cdot x ⌊\!⌊ \underline{\underline{\sigma}} \cdot y = \underline{\underline{\sigma}} \cdot (x ⌊\!⌊ y)$	DRTEM11
$\underline{\underline{\sigma}} \cdot x ⌊\!⌊ (\underline{\underline{\sigma}} \cdot y + \underline{\underline{\varepsilon}}) = \underline{\underline{\sigma}} \cdot (x ⌊\!⌊ y)$	DRTEM12

Table 4: Additional axioms for $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID.

Axioms DRTEM1-DRTEM3 are standard in an axiomatization of the free merge, so we will not elaborate on them.

Axioms DRTEM4-DRTEM7 ensure that $x \parallel y$ has a summand $\underline{\underline{\varepsilon}}$ iff x and y both have a summand $\underline{\underline{\varepsilon}}$. Note that $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}\text{-ID} \vdash \underline{\underline{\varepsilon}} ⌊\!⌊ x = \underline{\underline{\delta}}$ iff $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}\text{-ID} \nvdash x = \underline{\underline{\varepsilon}}$ (we will prove this in Proposition 4.2.7).

Axioms DRTEM8-DRTEM12 define the interaction between the undelayable empty process and the left merge. They work by first eliminating all summands $\underline{\underline{a}} \cdot y$ on the right-hand side of the left merge (Axiom DRTEM8), and then distinguishing between terms that only have a $\underline{\underline{\sigma}} \cdot y$ summand left (Axiom DRTEM11), only a $\underline{\underline{\varepsilon}}$ summand left (Axiom DRTEM10), both (Axiom DRTEM12), or none (Axiom DRTEM9). Notice that this case distinction follows the remark given after Lemma 4.1.9.

Remark 4.2.1 (Axioms of $PA^{-}_{\mathbf{drt},\varepsilon}$-ID, Part I)
Note that, unlike in a setting without empty process, we do not need an axiom $\underline{\underline{a}} \mathbin{|\!\!\lfloor} x = \underline{\underline{a}} \cdot x$, as that equality is derivable for all closed terms x (we will prove this in Proposition 4.2.6). For example:

$$\begin{aligned} \underline{\underline{a}} \mathbin{|\!\!\lfloor} \underline{\underline{b}} &= \underline{\underline{a}} \cdot \underline{\underline{\varepsilon}} \mathbin{|\!\!\lfloor} \underline{\underline{b}} = \underline{\underline{a}} \cdot (\underline{\underline{\varepsilon}} \parallel \underline{\underline{b}}) = \underline{\underline{a}} \cdot (\underline{\underline{\varepsilon}} \mathbin{|\!\!\lfloor} \underline{\underline{b}} + \underline{\underline{b}} \mathbin{|\!\!\lfloor} \underline{\underline{\varepsilon}}) = \underline{\underline{a}} \cdot (\underline{\underline{\delta}} + \underline{\underline{b}} \cdot \underline{\underline{\varepsilon}} \mathbin{|\!\!\lfloor} \underline{\underline{\varepsilon}}) \\ &= \underline{\underline{a}} \cdot \underline{\underline{b}} \cdot (\underline{\underline{\varepsilon}} \parallel \underline{\underline{\varepsilon}}) = \underline{\underline{a}} \cdot \underline{\underline{b}} \cdot (\underline{\underline{\varepsilon}} \mathbin{|\!\!\lfloor} \underline{\underline{\varepsilon}} + \underline{\underline{\varepsilon}} \mathbin{|\!\!\lfloor} \underline{\underline{\varepsilon}}) = \underline{\underline{a}} \cdot \underline{\underline{b}} \cdot (\underline{\underline{\varepsilon}} \mathbin{|\!\!\lfloor} \underline{\underline{\varepsilon}}) = \underline{\underline{a}} \cdot \underline{\underline{b}} \cdot \underline{\underline{\varepsilon}} \\ &= \underline{\underline{a}} \cdot \underline{\underline{b}} \end{aligned}$$

Remark 4.2.2 (Axioms of $PA^{-}_{\mathbf{drt},\varepsilon}$-ID, Part II)
The axiomatization as given in Table 4 has some consequences that may not be obvious at first sight. For example, consider what we get if we eliminate the merge operator from the expression $(\underline{\underline{a}} + \underline{\underline{\varepsilon}}) \parallel \underline{\underline{b}}$ using the axioms:

$$\begin{aligned} (\underline{\underline{a}} + \underline{\underline{\varepsilon}}) \parallel \underline{\underline{b}} &= (\underline{\underline{a}} + \underline{\underline{\varepsilon}}) \mathbin{|\!\!\lfloor} \underline{\underline{b}} + \underline{\underline{b}} \mathbin{|\!\!\lfloor} (\underline{\underline{a}} + \underline{\underline{\varepsilon}}) \\ &= \underline{\underline{a}} \mathbin{|\!\!\lfloor} \underline{\underline{b}} + \underline{\underline{\varepsilon}} \mathbin{|\!\!\lfloor} \underline{\underline{b}} + \underline{\underline{b}} \cdot (\underline{\underline{a}} + \underline{\underline{\varepsilon}}) \\ &= \underline{\underline{a}} \cdot \underline{\underline{b}} + \underline{\underline{\delta}} + \underline{\underline{b}} \cdot (\underline{\underline{a}} + \underline{\underline{\varepsilon}}) \\ &= \underline{\underline{a}} \cdot \underline{\underline{b}} + \underline{\underline{b}} \cdot (\underline{\underline{a}} + \underline{\underline{\varepsilon}}) \end{aligned}$$

So, if we execute a b first, then an a can always still follow. This may seem counter-intuitive; apparently it is not possible for the $\underline{\underline{\varepsilon}}$ to the left of the merge to execute before the b does, in which case a would *not* be enabled after the execution of the b.

The rationale behind this, is that the $\underline{\underline{\varepsilon}}$ is not something that "executes"; it merely represents an *option* to terminate. Hence, the option remains open until the process actually *does terminate* or performs an action. It does not "just get lost". So for example the process $\underline{\underline{a}} + \underline{\underline{\varepsilon}}$ cannot drop the $\underline{\underline{\varepsilon}}$ to turn itself into the process $\underline{\underline{a}}$. If it wants to lose the $\underline{\underline{\varepsilon}}$, it has to execute an a.

In the context of a (multiple) merge, this means that an $\underline{\underline{\varepsilon}}$ summand in one of the merge components does not manifest itself as long as the other components of the merge still have to execute one or more actions before they can terminate or move on to the following time-slice.

One could be tempted to "repair" this behavior, for example by dropping Axioms DRTEM4–DRTEM7, and adding an axiom like $\underline{\underline{\varepsilon}} \mathbin{|\!\!\lfloor} x = x$ (i.e., treating $\underline{\underline{\varepsilon}}$ like it was an ordinary action). Although this does give us the "desired" equality $(\underline{\underline{a}} + \underline{\underline{\varepsilon}}) \parallel \underline{\underline{b}} = (\underline{\underline{a}} + \underline{\underline{\varepsilon}}) \cdot \underline{\underline{b}} + \underline{\underline{b}} \cdot (\underline{\underline{a}} + \underline{\underline{\varepsilon}})$, it backfires immediately, as it destroys the associativity of the merge. This can be easily checked by expanding $((\underline{\underline{a}}+\underline{\underline{\varepsilon}}) \parallel \underline{\underline{b}}) \parallel \underline{\underline{c}}$ and $(\underline{\underline{a}}+\underline{\underline{\varepsilon}}) \parallel (\underline{\underline{b}} \parallel \underline{\underline{c}})$; it turns out that the second process has a summand $\underline{\underline{c}} \cdot \underline{\underline{b}}$, while the first process does not. This is of course unacceptable, as we already stated in our design goals. Any attempt at such "repairs" appears to have this consequence, unless one is willing to sacrifice the right-distributivity of the $\cdot$ over the $+$ (Axiom A4), which, again, would violate our design goals.

For other discussions of the above dilemma, see BAETEN AND VAN GLABBEEK [7] and VRANCKEN [22].

Remark 4.2.3 (Axioms of $PA^{-}_{\mathbf{drt},\varepsilon}$-ID, Part III)
The behavior of $\underline{\underline{\varepsilon}}$ with respect to the merge manifests itself even more unexpectedly when time comes into play. For example, consider the expression

$(\underline{\underline{\sigma}}\cdot\underline{\underline{a}} + \underline{\underline{\varepsilon}}) \parallel \underline{\underline{\sigma}}\cdot\underline{\underline{b}}$. Applying the axioms we get:

$$\begin{aligned}(\underline{\underline{\sigma}}\cdot\underline{\underline{a}} + \underline{\underline{\varepsilon}}) \parallel \underline{\underline{\sigma}}\cdot\underline{\underline{b}} &= (\underline{\underline{\sigma}}\cdot\underline{\underline{a}} + \underline{\underline{\varepsilon}}) \mathbin{\|\!\_} \underline{\underline{\sigma}}\cdot\underline{\underline{b}} + \underline{\underline{\sigma}}\cdot\underline{\underline{b}} \mathbin{\|\!\_} (\underline{\underline{\sigma}}\cdot\underline{\underline{a}} + \underline{\underline{\varepsilon}})\\ &= \underline{\underline{\sigma}}\cdot\underline{\underline{a}} \mathbin{\|\!\_} \underline{\underline{\sigma}}\cdot\underline{\underline{b}} + \underline{\underline{\varepsilon}} \mathbin{\|\!\_} \underline{\underline{\sigma}}\cdot\underline{\underline{b}} + \underline{\underline{\sigma}}\cdot\underline{\underline{b}} \mathbin{\|\!\_} (\underline{\underline{\sigma}}\cdot\underline{\underline{a}} + \underline{\underline{\varepsilon}})\\ &= \underline{\underline{\sigma}}\cdot\underline{\underline{a}} \mathbin{\|\!\_} \underline{\underline{\sigma}}\cdot\underline{\underline{b}} + \underline{\underline{\delta}} + \underline{\underline{\sigma}}\cdot\underline{\underline{b}} \mathbin{\|\!\_} \underline{\underline{\sigma}}\cdot\underline{\underline{a}}\\ &= \underline{\underline{\sigma}}\cdot\underline{\underline{a}} \parallel \underline{\underline{\sigma}}\cdot\underline{\underline{b}}\end{aligned}$$

So, the $\underline{\underline{\varepsilon}}$-summand on the left side of the merge is immaterial! How should we interpret this?

Again, by viewing $\underline{\underline{\varepsilon}}$ as the option to terminate (in the first time-slice, in this case), we can see that this option can never be exercised. That is to say, because the right component of the merge cannot terminate in the first time-slice, the option to terminate of the left component remains open till we enter the second time-slice. But as we leave the first time-slice, the $\underline{\underline{\varepsilon}}$ disappears, as it only presents an option to terminate *in that time-slice*. Hence, it vanishes into thin air.

As we have argued before, attempting to "repair" this behavior by treating $\underline{\underline{\varepsilon}}$ as an action, means we lose the associativity of the merge. Alternatively, we could replace Axioms DRTEM12 with the following:

$$\underline{\underline{\sigma}}\cdot x \mathbin{\|\!\_} (\underline{\underline{\sigma}}\cdot y + \underline{\underline{\varepsilon}}) = \underline{\underline{\sigma}}\cdot x \mathbin{\|\!\_} \underline{\underline{\sigma}}\cdot y + \underline{\underline{\sigma}}\cdot x \mathbin{\|\!\_} \underline{\underline{\varepsilon}}$$

Then the $\underline{\underline{\varepsilon}}$ does not vanish, but leads to an extra summand $\underline{\underline{\sigma}}\cdot b$ in the above example. This alternative, however, violates the time-determinism principle: $\underline{\underline{\sigma}}\cdot\underline{\underline{a}} \mathbin{\|\!\_} (\underline{\underline{\sigma}}\cdot\underline{\underline{b}} + \underline{\underline{\varepsilon}})$ can do a time step to $\underline{\underline{a}} \parallel \underline{\underline{b}}$, while $\underline{\underline{\sigma}}\cdot\underline{\underline{a}} \mathbin{\|\!\_} \underline{\underline{\sigma}}\cdot\underline{\underline{b}} + \underline{\underline{\sigma}}\cdot\underline{\underline{b}}$ can do a time step to $\underline{\underline{a}} \parallel \underline{\underline{b}} + \underline{\underline{b}}$. As stated in our design goals, this is not acceptable. Again, we are stuck.

Now look at the following example. Applying our axioms, we have:

$$\begin{aligned}(\underline{\underline{a}} + \underline{\underline{\varepsilon}}) \parallel \underline{\underline{\sigma}}\cdot\underline{\underline{b}} &= (\underline{\underline{a}} + \underline{\underline{\varepsilon}}) \mathbin{\|\!\_} \underline{\underline{\sigma}}\cdot\underline{\underline{b}} + \underline{\underline{\sigma}}\cdot\underline{\underline{b}} \mathbin{\|\!\_} (\underline{\underline{a}} + \underline{\underline{\varepsilon}})\\ &= \underline{\underline{a}} \mathbin{\|\!\_} \underline{\underline{\sigma}}\cdot\underline{\underline{b}} + \underline{\underline{\varepsilon}} \mathbin{\|\!\_} \underline{\underline{\sigma}}\cdot\underline{\underline{b}} + \underline{\underline{\sigma}}\cdot\underline{\underline{b}} \mathbin{\|\!\_} (\underline{\underline{a}} + \underline{\underline{\varepsilon}})\\ &= \underline{\underline{a}}\cdot\underline{\underline{\sigma}}\cdot\underline{\underline{b}} + \underline{\underline{\delta}} + \underline{\underline{\sigma}}\cdot\underline{\underline{b}} \mathbin{\|\!\_} \underline{\underline{\varepsilon}}\\ &= \underline{\underline{a}}\cdot\underline{\underline{\sigma}}\cdot\underline{\underline{b}} + \underline{\underline{\sigma}}\cdot\underline{\underline{b}}\\ &= (\underline{\underline{a}} + \underline{\underline{\varepsilon}})\cdot\underline{\underline{\sigma}}\cdot\underline{\underline{b}}\end{aligned}$$

Here, the $\underline{\underline{\varepsilon}}$-summand on the left side of the merge does materialize. This is due to the fact that $\underline{\underline{a}} + \underline{\underline{\varepsilon}}$ cannot move on to the second time-slice, while $\underline{\underline{\sigma}}\cdot\underline{\underline{b}}$ can. So, $\underline{\underline{a}} + \underline{\underline{\varepsilon}}$ is forced to terminate before $\underline{\underline{\sigma}}\cdot\underline{\underline{b}}$ can move on, and the $\underline{\underline{\varepsilon}}$ does not vanish.

If, for the sake of orthogonality, we would want the $\underline{\underline{\varepsilon}}$-summand to disappear in this case also, we could for example replace Axioms DRTEM10 by $\underline{\underline{\sigma}}\cdot x \mathbin{\|\!\_} \underline{\underline{\varepsilon}} = \underline{\underline{\delta}}$. That however, would lead to $\underline{\underline{\sigma}} \parallel \underline{\underline{\varepsilon}} = \underline{\underline{\delta}}$, destroying the unit property of $\underline{\underline{\varepsilon}}$ with respect to the $\parallel$. Furthermore, this leads to $\underline{\underline{\sigma}} \parallel \underline{\underline{a}} = \underline{\underline{a}}\cdot\underline{\underline{\delta}}$, in itself enough reason to dismiss this alternative. If we attempt to save what can be saved, for example by putting $\underline{\underline{\varepsilon}} \mathbin{\|\!\_} x = x$, we again lose the associativity of the merge.

Concluding: when x can terminate, and y can do a time step, then the termination option of x materializes in $x \parallel y$ *if and only if* x cannot do a time step. This may seem, and probably is, counter intuitive. However, a simpler way is not conceivable. If the termination option would *always* materialize, we either lose the associativity of the merge or time determinism. If it would *never* materialize, we lose the unit property with respect to the merge and the associativity of the merge. So, both simple ways are too simple, and lead to violations of our design goals.

Remark 4.2.4 (Axioms of $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID, Part IV)
In the case where one of the n components in a multiple merge has *only* the option to terminate (so the component has no enabled actions or time steps), while the other components do not all have that option, the $\underline{\varepsilon}$ again vanishes, collapsing the merge to $n-1$ components. So for example, $\underline{\varepsilon} \parallel \underline{\underline{\sigma}} \cdot \underline{\underline{a}} \parallel \underline{\underline{\sigma}} \cdot \underline{\underline{b}} = \underline{\underline{\sigma}} \cdot \underline{\underline{a}} \parallel \underline{\underline{\sigma}} \cdot \underline{\underline{b}}$. Or, more generally, $\underline{\varepsilon} \parallel x = x \parallel \underline{\varepsilon} = x$ for any closed process term x: with respect to closed terms, $\underline{\varepsilon}$ is also a proper unit element for the merge. We will prove this in Proposition 4.2.6.

Definition 4.2.5 (Semantics of $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID)
The semantics of $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID are given by the term-deduction system named $T(\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}\text{-ID})$, induced by the deduction rules for $\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}$-ID given in Definition 4.1.4, and the additional deduction rules for $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID shown in Table 5.

$$\frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y} \qquad \frac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'} \qquad \frac{x \xrightarrow{a} x'}{x \mathbin{\lfloor\!\lfloor} y \xrightarrow{a} x' \parallel y}$$

$$\frac{x \xrightarrow{\sigma} x',\ y \xrightarrow{\sigma} y'}{x \parallel y \xrightarrow{\sigma} x' \parallel y'} \qquad \frac{x \xrightarrow{\sigma} x',\ y \xrightarrow{\sigma} y'}{x \mathbin{\lfloor\!\lfloor} y \xrightarrow{\sigma} x' \mathbin{\lfloor\!\lfloor} y'}$$

$$\frac{x \overset{\sigma}{\nrightarrow},\ x\downarrow,\ y \xrightarrow{\sigma} y'}{x \parallel y \xrightarrow{\sigma} y'} \qquad \frac{x \xrightarrow{\sigma} x',\ y \overset{\sigma}{\nrightarrow},\ y\downarrow}{x \parallel y \xrightarrow{\sigma} x'} \qquad \frac{x \xrightarrow{\sigma} x',\ y \overset{\sigma}{\nrightarrow},\ y\downarrow}{x \mathbin{\lfloor\!\lfloor} y \xrightarrow{\sigma} x'}$$

$$\frac{x\downarrow,\ y\downarrow}{(x \parallel y)\downarrow} \qquad \frac{x\downarrow,\ y\downarrow}{(x \mathbin{\lfloor\!\lfloor} y)\downarrow}$$

Table 5: Additional deduction rules for $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID.

The deduction rules for the parallel composition with respect to time transitions, shown in Table 5, suffer from the same complications we saw in the deduction rules for the sequential composition with respect to time transitions. We give two examples to clarify them.

First, as we have $\underline{\varepsilon} \parallel \underline{\underline{\sigma}} \cdot \underline{\underline{a}} = \underline{\underline{\sigma}} \cdot \underline{\underline{a}}$, we should have $\underline{\varepsilon} \parallel \underline{\underline{\sigma}} \cdot \underline{\underline{a}} \xrightarrow{\sigma} \underline{\underline{a}}$. Likewise, as $(\underline{\underline{\sigma}} \cdot \underline{\underline{a}} + \underline{\varepsilon}) \parallel \underline{\underline{\sigma}} \cdot \underline{\underline{b}} = \underline{\underline{\sigma}} \cdot (\underline{\underline{a}} \parallel \underline{\underline{b}})$, we should have $(\underline{\underline{\sigma}} \cdot \underline{\underline{a}} + \underline{\varepsilon}) \parallel \underline{\underline{\sigma}} \cdot \underline{\underline{b}} \xrightarrow{\sigma} \underline{\underline{a}} \parallel \underline{\underline{b}}$, and not $(\underline{\underline{\sigma}} \cdot \underline{\underline{a}} + \underline{\varepsilon}) \parallel \underline{\underline{\sigma}} \cdot \underline{\underline{b}} \xrightarrow{\sigma} \underline{\underline{b}}$.

As a consequence we obtain the same results as before: time determinism holds, we have a bisimulation model, we have an elimination theorem, soundness theorem and completeness theorem.

Proposition 4.2.6 (Properties of $PA^{-}_{drt,\varepsilon}$-ID, Part I)
Let x be a closed term and $a \in A_\delta$. Then the following properties hold:

(i). $PA^{-}_{drt,\varepsilon}\text{-}ID \vdash x \parallel \underline{\underline{\varepsilon}} = x$

(ii). $PA^{-}_{drt,\varepsilon}\text{-}ID \vdash \underline{\underline{\varepsilon}} \parallel x = x$

(iii). $PA^{-}_{drt,\varepsilon}\text{-}ID \vdash \underline{\underline{a}} \mathbin{\lfloor\!\lfloor} x = \underline{\underline{a}} \cdot x$

(iv). $PA^{-}_{drt,\varepsilon}\text{-}ID \vdash \underline{\underline{\delta}} \mathbin{\lfloor\!\lfloor} x = \underline{\underline{\delta}}$

(v). $PA^{-}_{drt,\varepsilon}\text{-}ID \vdash x \mathbin{\lfloor\!\lfloor} \underline{\underline{\varepsilon}} = x$

Proposition 4.2.7 (Properties of $PA^{-}_{drt,\varepsilon}$-ID, Part II)
Let x, y, z be closed $PA^{-}_{drt,\varepsilon}$-ID terms. Then the following properties hold:

(i). $PA^{-}_{drt,\varepsilon}\text{-}ID \vdash \underline{\underline{\varepsilon}} \mathbin{\lfloor\!\lfloor} x = \begin{cases} \underline{\underline{\varepsilon}} & \text{if } x\downarrow \\ \underline{\underline{\delta}} & \text{if } x\not\downarrow \end{cases}$

(ii). $PA^{-}_{drt,\varepsilon}\text{-}ID \vdash \underline{\underline{\sigma}} \cdot x \mathbin{\lfloor\!\lfloor} \nu(y) = \begin{cases} \underline{\underline{\sigma}} \cdot x & \text{if } y\downarrow \\ \underline{\underline{\delta}} & \text{if } y\not\downarrow \end{cases}$

(iii). $PA^{-}_{drt,\varepsilon}\text{-}ID \vdash \underline{\underline{\sigma}} \cdot x \mathbin{\lfloor\!\lfloor} (\nu(y) + \underline{\underline{\sigma}} \cdot z) = \underline{\underline{\sigma}}(x \mathbin{\lfloor\!\lfloor} z)$

(iv). $PA^{-}_{drt,\varepsilon}\text{-}ID \vdash \nu(x) \mathbin{\lfloor\!\lfloor} y = \nu(\nu(x) \mathbin{\lfloor\!\lfloor} y)$.

(v). $PA^{-}_{drt,\varepsilon}\text{-}ID \vdash \nu(x) \parallel \nu(y) = \nu(\nu(x) \parallel \nu(y))$.

Theorem 4.2.8 (Axioms of Standard Concurrency for $PA^{-}_{drt,\varepsilon}$-ID)
Let x, y, z be closed $PA^{-}_{drt,\varepsilon}$-ID terms. Then the following properties hold:

(i). $PA^{-}_{drt,\varepsilon}\text{-}ID \vdash x \parallel y = y \parallel x$

(ii). $PA^{-}_{drt,\varepsilon}\text{-}ID \vdash (x \mathbin{\lfloor\!\lfloor} y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} (y \parallel z)$

(iii). $PA^{-}_{drt,\varepsilon}\text{-}ID \vdash (x \parallel y) \parallel z = x \parallel (y \parallel z)$

Thus, we have commutativity and associativity of the merge operator for closed terms. We usually assume these laws for all processes.

4.3 $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}$-ID

In this section, we modify the process algebra $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID by extending the free merge to a merge, where the axioms for the communication merge are based on the inductive definition of basic terms. The resulting process algebra is called $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}$-ID.

We add to the signature of $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID the *communication merge operator* $|$, and the *encapsulation operator* ∂_H. As an additional parameter to the theory, we assume given a binary *communication function* γ on A_δ, that is commutative and associative, and results in δ whenever at least one of the arguments is δ.

The process algebra named $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}$-ID is axiomatized by the axioms of $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID given in Table 4, *minus* Axiom DRTEM1, *plus* Axioms DRTECM1-DRTECM9, DRTECF, and DRTED1-DRTED5 shown in Table 6: $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}$-ID = A1-A5 + DRT1-DRT2 + DRTE1-DRTE2 + TF + DCSE1-DCSE4 + DRTEM2-DRTEM12 + DRTECM1-DRTECM9 + DRTECF + DRTED1-DRTED6.

$x \parallel y = x \mathbin{⌊\!⌊} y + y \mathbin{⌊\!⌊} x + x \mid y$		DRTECM1
$\underline{\underline{a}} \cdot x \mid \underline{\underline{b}} \cdot y = (\underline{\underline{a}} \mid \underline{\underline{b}}) \cdot (x \parallel y)$		DRTECM2
$\underline{\underline{\sigma}} \cdot x \mid \underline{\underline{\sigma}} \cdot y = \underline{\underline{\sigma}} \cdot (x \mid y)$		DRTECM3
$\underline{\underline{\sigma}} \cdot x \mid \underline{\underline{a}} \cdot y = \underline{\underline{\delta}}$		DRTECM4
$\underline{\underline{a}} \cdot x \mid \underline{\underline{\sigma}} \cdot y = \underline{\underline{\delta}}$		DRTECM5
$x \mid \underline{\underline{\varepsilon}} = \underline{\underline{\delta}}$		DRTECM6
$\underline{\underline{\varepsilon}} \mid x = \underline{\underline{\delta}}$		DRTECM7
$(x + y) \mid z = x \mid z + y \mid z$		DRTECM8
$x \mid (y + z) = x \mid y + x \mid z$		DRTECM9
$\underline{\underline{a}} \mid \underline{\underline{b}} = \underline{\underline{c}}$	if $\gamma(a,b) = c$	DRTECF
$\partial_H(\underline{\underline{a}}) = \underline{\underline{a}}$	if $a \notin H$	DRTED1
$\partial_H(\underline{\underline{a}}) = \underline{\underline{\delta}}$	if $a \in H$	DRTED2
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$		DRTED3
$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$		DRTED4
$\partial_H(\underline{\underline{\sigma}}) = \underline{\underline{\sigma}}$		DRTED5
$\partial_H(\underline{\underline{\varepsilon}}) = \underline{\underline{\varepsilon}}$		DRTED6

Table 6: Additional axioms for $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}$-ID.

The axioms DRTECM1, DRTECM2, DRTECM8, DRTECM9, DRTECF, and DRTED1-DRTED4 all correspond to identical axioms in the theory without empty process or the theory without time, so we will not elaborate on them.

Axiom DRTECM3 expresses that the time-unit delay constant distributes over the communication merge (due to the fact that time steps do not communicate and weak time factorization). Axioms DRTECM4 and DRTECM5 express that if one side of a communication merge can only do a time step, and the other side cannot do a time step, then the communication merge collapses to deadlock.

Axioms DRTECM6 and DRTECM7 express that the undelayable empty process does not communicate: if either side of a communication merge consist of just an undelayable empty process, the whole communication merge again collapses to deadlock.

Finally, Axioms DRTED5 and DRTED6 express that the time-unit delay constant and the undelayable empty process are immune to encapsulation.

Definition 4.3.1 (Semantics of $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}$-ID)
The semantics of $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}$-ID are given by the term-deduction system named $T(\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}\text{-ID})$, induced by the deduction rules for $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID given in Definition 4.2.5, and the additional deduction rules for $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}$-ID shown in Table 7.

$$\frac{x \xrightarrow{a} x',\ y \xrightarrow{b} y',\ \gamma(a,b) = c}{x \parallel y \xrightarrow{c} x' \parallel y'} \qquad \frac{x \xrightarrow{a} x',\ y \xrightarrow{b} y',\ \gamma(a,b) = c}{x \mid y \xrightarrow{c} x' \parallel y'}$$

$$\frac{x \xrightarrow{\sigma} x',\ y \xrightarrow{\sigma} y'}{x \mid y \xrightarrow{\sigma} x' \mid y'} \qquad \frac{x \xrightarrow{a} x',\ a \notin H}{\partial_H(x) \xrightarrow{a} \partial_H(x')}$$

$$\frac{x{\downarrow}}{\partial_H(x){\downarrow}} \qquad \frac{x \xrightarrow{\sigma} x'}{\partial_H(x) \xrightarrow{\sigma} \partial_H(x')}$$

Table 7: Additional deduction rules for $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}$-ID.

These deduction rules are entirely straightforward, so we will not discuss them. As a consequence, we again have time determinism, the bisimulation model, elimination, soundness and completeness.

Proposition 4.3.2 (Properties of $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}$-ID)
Let x be a closed term and $a \in A_\delta$. Then the following properties hold:

(i). $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}\text{-ID} \vdash x \parallel \underline{\underline{\varepsilon}} = x$

(ii). $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}\text{-ID} \vdash \underline{\underline{\varepsilon}} \parallel x = x$

(iii). $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}\text{-ID} \vdash \underline{\underline{a}} \mathbin{\|\!\!\_} x = \underline{\underline{a}} \cdot x$

(iv). $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}\text{-ID} \vdash \underline{\underline{\delta}} \mathbin{\|\!\!\_} x = \underline{\underline{\delta}}$

(v). $ACP^{-}_{drt,\varepsilon}\text{-}ID \vdash x \mathbin{\lfloor\!\lfloor} \underline{\underline{\varepsilon}} = x$

Theorem 4.3.3 (Axioms of Standard Concurrency for $ACP^{-}_{drt,\varepsilon}$-ID)
Let x, y, z be closed $ACP^{-}_{drt,\varepsilon}$-ID terms. Then the following properties hold:

(i). $ACP^{-}_{drt,\varepsilon}\text{-}ID \vdash x \mid y = y \mid x$

(ii). $ACP^{-}_{drt,\varepsilon}\text{-}ID \vdash x \parallel y = y \parallel x$

(iii). $ACP^{-}_{drt,\varepsilon}\text{-}ID \vdash (x \mid y) \mid z = x \mid (y \mid z)$

(iv). $ACP^{-}_{drt,\varepsilon}\text{-}ID \vdash (x \mathbin{\lfloor\!\lfloor} y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} (y \parallel z)$

(v). $ACP^{-}_{drt,\varepsilon}\text{-}ID \vdash x \mid (y \mathbin{\lfloor\!\lfloor} z) = (x \mid y) \mathbin{\lfloor\!\lfloor} z$

(vi). $ACP^{-}_{drt,\varepsilon}\text{-}ID \vdash (x \parallel y) \parallel z = x \parallel (y \parallel z)$

5 The Empty Process with Delayable Actions

Now, we extend the process algebras of Section 4 with *delayable actions*. We define the process algebra $BPA_{drt,\varepsilon}$-ID, and step-by-step extend it with the free merge and the merge. For each extension we give an axiomatization, an operational semantics, and a description of all concepts that are introduced. Furthermore, we give the considerations that have led us to construct these algebras in the way we have done.

5.1 $BPA_{drt,\varepsilon}$-ID

In this section, we define the process algebra $BPA_{drt,\varepsilon}$-ID, which is basically the process algebra $BPA^{-}_{drt,\varepsilon}$-ID extended with delayable actions (note the absence of the superscript "$-$").

We add to the signature only the *delayable empty process constant* ε. The *delayable empty process* ε in the signature of $BPA_{drt,\varepsilon}$-ID is the delayable counterpart of $\underline{\underline{\varepsilon}}$: it can terminate successfully in the current time-slice, but it can also move on to a following time-slice.

Definition 5.1.1 (Axioms of $BPA_{drt,\varepsilon}$-ID)
The process algebra named $BPA_{drt,\varepsilon}$-ID is axiomatized by the axioms of $BPA^{-}_{drt,\varepsilon}$-ID given in Definition 4.1.3, and Axioms DEP and DA shown in Table 8: $BPA_{drt,\varepsilon}$-ID = A1–A5 + DRTE1–DRTE4 + TF + DCSE1–DCSE4 + DEP + DA.

Axiom DEP defines the ε: the choice between terminating now, or moving on to the next time-slice. Axiom DA then uses ε to define the delayable actions: a delayable action corresponds to moving to some time-slice, followed by the execution of an a.

$$\varepsilon = \underline{\underline{\varepsilon}} + \underline{\underline{\sigma}} \cdot \varepsilon \qquad \text{DEP}$$
$$a = \varepsilon \cdot \underline{\underline{a}} \qquad \text{DA}$$

Table 8: Additional axioms for $\mathrm{BPA}_{\mathrm{drt},\varepsilon}$-ID.

Definition 5.1.2 (Recursion Principle for ε)
Besides the axioms mentioned in Definition 5.1.1, the system $\mathrm{BPA}^{+}_{\mathrm{drt},\varepsilon}$-ID also contains the *recursion principle* RSP(DEP) shown in Table 9.

$$y = x + \underline{\underline{\sigma}} \cdot y \quad \Longrightarrow \quad y = \varepsilon \cdot x \qquad \text{RSP(DEP)}$$

Table 9: Recursive specification principle for delayable empty process.

The principle RSP(DEP) is a special case of the *Recursive Definition Principle* RSP, see e.g. BAETEN AND WEIJLAND [10], that states that a guarded recursive specification can have at most one solution. In terms of the Kleene star, we have intuitively the equality $\sigma^*(x) = \varepsilon \cdot x$.

Remark 5.1.3 (The Recursion Principle RSP(DEP))
The recursion principle RSP(DEP) will be used to derive equalities between terms that contain delayable actions. As it turns out, RSP(DEP) is very powerful: for every new operator we add to $\mathrm{BPA}^{+}_{\mathrm{drt},\varepsilon}$-ID, we only have to add axioms to eliminate this operator from terms that contain only *undelayable* actions. The elimination of the new operator from terms that also contain *delayable* actions is then possible using RSP(DEP).

The price to be paid for this power, is the fact that RSP(DEP) is formulated as a conditional axiom. Hence, we lose the strict equationality of our process algebra. If so desired, it is possible to maintain strict equationality, but that requires the addition of new axioms to deal with delayable actions for every new operator added.

In Example 5.1.4 we show how to apply RSP(DEP) to derive simple equalities in $\mathrm{BPA}^{+}_{\mathrm{drt},\varepsilon}$-ID. In Example 5.2.1 we show how to apply it to derive equalities involving a newly introduced operator, namely the free merge.

Example 5.1.4 (Use of RSP(DEP))
First, we show how to derive the equality $a = \varepsilon \cdot a$ in $\mathrm{BPA}^{+}_{\mathrm{drt},\varepsilon}$-ID. To begin with, we derive $a = \varepsilon \cdot \underline{\underline{a}} = (\underline{\underline{\varepsilon}} + \underline{\underline{\sigma}} \cdot \varepsilon) \cdot \underline{\underline{a}} = \underline{\underline{\varepsilon}} \cdot \underline{\underline{a}} + \underline{\underline{\sigma}} \cdot \varepsilon \cdot \underline{\underline{a}} = \underline{\underline{a}} + \underline{\underline{\sigma}} \cdot a$. Using this equality, we can then derive $a = \underline{\underline{a}} + \underline{\underline{\sigma}} \cdot a = \underline{\underline{a}} + \underline{\underline{\sigma}} \cdot a + \underline{\underline{\sigma}} \cdot a = a + \underline{\underline{\sigma}} \cdot a$. Applying RSP(DEP), we now get the desired result $a = \varepsilon \cdot a$. Note that the converse, namely $a = a \cdot \varepsilon$, does not hold, as $a \cdot \varepsilon$ can still idle after it has done an a, while a cannot.

Secondly, we show how to derive the equality $\varepsilon \cdot x + \varepsilon \cdot y = \varepsilon \cdot (x + y)$. Applying the axioms, we get $\varepsilon \cdot x + \varepsilon \cdot y = (\underline{\underline{\varepsilon}} + \underline{\underline{\sigma}} \cdot \varepsilon) \cdot x + (\underline{\underline{\varepsilon}} + \underline{\underline{\sigma}} \cdot \varepsilon) \cdot y = \underline{\underline{\varepsilon}} \cdot x + \underline{\underline{\sigma}} \cdot \varepsilon \cdot x + \underline{\underline{\varepsilon}} \cdot y + \underline{\underline{\sigma}} \cdot \varepsilon \cdot y = x + y + \underline{\underline{\sigma}} \cdot (\varepsilon \cdot x + \varepsilon \cdot y)$. Applying RSP(DEP), we now get the desired result $\varepsilon \cdot x + \varepsilon \cdot y = \varepsilon \cdot (x + y)$.

Thirdly, we show how to derive the equality $\underline{\underline{\sigma}} \cdot \varepsilon = \varepsilon \cdot \underline{\underline{\sigma}}$. Applying the axioms, we get $\underline{\underline{\sigma}} \cdot \varepsilon = \underline{\underline{\sigma}} \cdot (\underline{\underline{\varepsilon}} + \underline{\underline{\sigma}} \cdot \varepsilon) = \underline{\underline{\sigma}} \cdot \underline{\underline{\varepsilon}} + \underline{\underline{\sigma}} \cdot \underline{\underline{\sigma}} \cdot \varepsilon = \underline{\underline{\sigma}} + \underline{\underline{\sigma}} \cdot \underline{\underline{\sigma}} \cdot \varepsilon$. Applying RSP(DEP), we now get the desired result $\underline{\underline{\sigma}} \cdot \varepsilon = \varepsilon \cdot \underline{\underline{\sigma}}$.

Definition 5.1.5 (Semantics of $\mathrm{BPA}_{\mathrm{drt},\varepsilon}$–ID)
The semantics of $\mathrm{BPA}_{\mathrm{drt},\varepsilon}$–ID are given by the term-deduction system named $T(\mathrm{BPA}_{\mathrm{drt},\varepsilon}\text{–ID})$, induced by the deduction rules for $\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}$–ID given in Definition 4.1.4, and the additional deduction rules for $\mathrm{BPA}_{\mathrm{drt},\varepsilon}$–ID shown in Table 10.

$$a \xrightarrow{a} \underline{\underline{\varepsilon}} \qquad a \xrightarrow{\sigma} a \qquad \delta \xrightarrow{\sigma} \delta \qquad \varepsilon \xrightarrow{\sigma} \varepsilon \qquad \varepsilon\downarrow$$

Table 10: Additional deduction rules for $\mathrm{BPA}_{\mathrm{drt},\varepsilon}$–ID.

The new deduction rules are very simple: a, δ, and ε can do a time-step to themselves, a can do an a-step to $\underline{\underline{\varepsilon}}$, and ε has the option to terminate.

As a result, we retain time determinism, and can define a bisimulation model as before. To the clauses for the definition of basic terms, we add one extra clause:

- if t is a basic term, then $\varepsilon \cdot t$ is a basic term.

On the basis of this extended definition, we obtain an elimination theorem, and soundness and completeness follow. All details can be found in VEREIJKEN [20].

5.2 $\mathrm{PA}_{\mathrm{drt},\varepsilon}$–ID

In this section, we define the process algebra $\mathrm{PA}_{\mathrm{drt},\varepsilon}$–ID, which is basically the process algebra $\mathrm{BPA}_{\mathrm{drt},\varepsilon}$–ID extended with the free merge. Adding the free merge to $\mathrm{BPA}_{\mathrm{drt},\varepsilon}$–ID to get $\mathrm{PA}_{\mathrm{drt},\varepsilon}$–ID is entirely similar to adding the free merge in the case without delayable actions (treated Section 4.2), so we contend ourselves with only giving one example to illustrate the new process algebra.

Example 5.2.1 (Use of RSP(DEP) with $\mathrm{PA}_{\mathrm{drt},\varepsilon}$–ID)
We show how to derive the equality $a \parallel b = a \cdot b + b \cdot a$. Applying the axioms, we get $a \parallel b = a \mathbin{\lfloor\!\lfloor} b + b \mathbin{\lfloor\!\lfloor} a = (\underline{\underline{a}} + \underline{\underline{\sigma}} \cdot a) \mathbin{\lfloor\!\lfloor} b + (\underline{\underline{b}} + \underline{\underline{\sigma}} \cdot b) \mathbin{\lfloor\!\lfloor} a = \underline{\underline{a}} \mathbin{\lfloor\!\lfloor} b + \underline{\underline{\sigma}} \cdot a \mathbin{\lfloor\!\lfloor} (\underline{\underline{b}} + \underline{\underline{\sigma}} \cdot b) + \underline{\underline{b}} \mathbin{\lfloor\!\lfloor} a + \underline{\underline{\sigma}} \cdot b \mathbin{\lfloor\!\lfloor} (\underline{\underline{a}} + \underline{\underline{\sigma}} \cdot a) = \underline{\underline{a}} \cdot b + \underline{\underline{\sigma}} \cdot (a \mathbin{\lfloor\!\lfloor} b) + \underline{\underline{b}} \cdot a + \underline{\underline{\sigma}} \cdot (b \mathbin{\lfloor\!\lfloor} a) = \underline{\underline{a}} \cdot b + \underline{\underline{b}} \cdot a + \underline{\underline{\sigma}} \cdot (a \parallel b)$. Applying RSP(DEP), and using $\varepsilon \cdot (x + y) = \varepsilon \cdot x + \varepsilon \cdot y$ (see Example 5.1.4), we now get $a \parallel b = \varepsilon \cdot (\underline{\underline{a}} \cdot b + \underline{\underline{b}} \cdot a) = \varepsilon \cdot \underline{\underline{a}} \cdot b + \varepsilon \cdot \underline{\underline{b}} \cdot a = a \cdot b + b \cdot a$.

5.3 $\mathrm{ACP}_{\mathrm{drt},\varepsilon}$-ID

Finally, we define the process algebra $\mathrm{ACP}_{\mathrm{drt},\varepsilon}$-ID, which is basically the process algebra $\mathrm{PA}_{\mathrm{drt},\varepsilon}$-ID with the free merge modified to merge. Again, adding the merge is entirely similar to adding the merge in the case without delayable actions (treated in Section 4.3). Further details can be found in VEREIJKEN [20].

6 Embeddings

There exist many embeddings between the process algebras given in this contribution and other process algebras found in the literature.

First, we can embed the untimed process algebras $\mathrm{BPA}_{\delta\varepsilon}$, $\mathrm{PA}_{\varepsilon}$, $\mathrm{ACP}_{\varepsilon}$ (see BAETEN AND VAN GLABBEEK [7]) into the timed counterparts $\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}$-ID, $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID, $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}$-ID presented here by projecting a onto $\underline{\underline{a}}$ for $a \in A_{\delta\varepsilon}$, and everything else onto itself.

Second, we can embed the untimed process algebras $\mathrm{BPA}_{\delta\varepsilon}$, $\mathrm{PA}_{\varepsilon}$, $\mathrm{ACP}_{\varepsilon}$ into the timed algebras $\mathrm{BPA}_{\mathrm{drt},\varepsilon}$-ID, $\mathrm{PA}_{\mathrm{drt},\varepsilon}$-ID, $\mathrm{ACP}^{-}_{\mathrm{drt}}$-ID presented here in two different ways: either by projecting the untimed process a onto the undelayable process $\underline{\underline{a}}$ for $a \in A_{\delta\varepsilon}$, and everything else onto itself, or by projecting the untimed process a onto the delayable process $a \cdot \varepsilon$ for $a \in A_{\delta\varepsilon}$, and everything else onto itself. Note that projecting the untimed process a onto the delayable process a for $a \in A_{\delta\varepsilon}$ will *not* work, as the untimed empty process ε is a unit element for the sequential composition in $\mathrm{BPA}_{\delta\varepsilon}$, whereas the delayable empty process ε is not a unit element for the sequential composition in $\mathrm{BPA}_{\mathrm{drt},\varepsilon}$-ID; we have $\mathrm{BPA}_{\delta\varepsilon} \vdash a = a \cdot \varepsilon$, but $\mathrm{BPA}_{\mathrm{drt},\varepsilon}\text{-ID} \nvdash a = a \cdot \varepsilon$. By projecting a onto $a \cdot \varepsilon$ we do get a proper embedding, as $\mathrm{BPA}_{\mathrm{drt},\varepsilon}\text{-ID} \vdash (a \cdot \varepsilon) = (a \cdot \varepsilon) \cdot (\varepsilon \cdot \varepsilon)$ does indeed hold.

Next, we can embed the process algebras $\mathrm{BPA}^{-}_{\mathrm{drt}}$-ID, $\mathrm{PA}^{-}_{\mathrm{drt}}$-ID, $\mathrm{ACP}^{-}_{\mathrm{drt}}$-ID without empty process (see BAETEN AND BERGSTRA [3, 4, 5], BAETEN AND RENIERS [9], RENIERS AND VEREIJKEN [19]) into the process algebras $\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}$-ID, $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}$-ID, $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}$-ID by projecting $\sigma_{\mathrm{rel}}(x)$ onto $\underline{\underline{\sigma}} \cdot x$, and everything else onto itself.

Finally, we also have the natural embeddings $\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}\text{-ID} \subseteq \mathrm{BPA}_{\mathrm{drt},\varepsilon}\text{-ID}$, $\mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}\text{-ID} \subseteq \mathrm{PA}_{\mathrm{drt},\varepsilon}\text{-ID}$, $\mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}\text{-ID} \subseteq \mathrm{ACP}_{\mathrm{drt},\varepsilon}\text{-ID}$, $\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}\text{-ID} \subseteq \mathrm{PA}^{-}_{\mathrm{drt},\varepsilon}\text{-ID}$, $\mathrm{BPA}^{-}_{\mathrm{drt},\varepsilon}\text{-ID} \subseteq \mathrm{ACP}^{-}_{\mathrm{drt},\varepsilon}\text{-ID}$, $\mathrm{BPA}_{\mathrm{drt},\varepsilon}\text{-ID} \subseteq \mathrm{PA}_{\mathrm{drt},\varepsilon}\text{-ID}$, $\mathrm{BPA}_{\mathrm{drt},\varepsilon}\text{-ID} \subseteq \mathrm{ACP}_{\mathrm{drt},\varepsilon}\text{-ID}$, where everything is projected onto itself.

7 Conclusions

We have successfully introduced the empty process within the context of discrete-time process algebra with relative timing. In doing so, we found that there is not much room for choice: the constraints of the unit-element property with respect to sequential composition and merge, associativity of the merge, time determinism, and taking $\mathrm{BPA}^{-}_{\mathrm{drt}}$-ID as a basis almost completely determine which course to take.

The axioms we have given lead to a sound and complete axiomatization of our bisimulation model. For closed terms, the axioms of standard concurrency are derivable.

As the behavior of the empty process is not always into accordance with one's first intuition (see Remark 4.2.2 and Remark 4.2.3), one should be very careful when verifying protocols, to make sure that the protocol that is coded in process algebra, is indeed the same as the one that is supposed to be under study.

The discrete-time empty process makes for a worthwhile addition to process algebra. It can potentially be very useful in giving a formal semantics to specification languages such as SDL and MSC. It also extends the class of processes that can be finitely specified.

The usefulness of the empty process with respect to real-life protocol verification remains to be determined.

References

[1] J. C. M. Baeten and J. A. Bergstra. *Process algebra with a zero object.* In [8], pages 83–98, 1990.

[2] J. C. M. Baeten and J. A. Bergstra. *On sequential composition, action prefixes and process prefix.* Formal Aspects of Computing, **6**(3):250–268, 1994.

[3] J. C. M. Baeten and J. A. Bergstra. *Discrete time process algebra with abstraction.* In [17], pages 1–15, 1995.

[4] J. C. M. Baeten and J. A. Bergstra. *Discrete time process algebra.* Formal Aspects of Computing, **8**(2):188–208, 1996.

[5] J. C. M. Baeten and J. A. Bergstra. *Discrete time process algebra: Absolute time, relative time and parametric time.* Fundamenta Informaticæ, **29**(1, 2):51–76, 1997.

[6] J. C. M. Baeten and J. A. Bergstra. *Process algebra with propositional signals.* Theoretical Computer Science, **177**(2):381–405, 1997.

[7] J. C. M. Baeten and R. J. van Glabbeek. *Merge and termination in process algebra.* In [16], pages 153–172, 1987.

[8] J. C. M. Baeten and J. W. Klop, editors. *CONCUR '90, Theories of Concurrency: Unification and Extension.* Number 458 in Lecture Notes in Computer Science. Springer-Verlag, 1990. Proceedings of CONCUR '90, Amsterdam, The Netherlands, August 1990.

[9] J. C. M. Baeten and M. A. Reniers. *Discrete time process algebra with relative timing.* Unpublished lecture notes of Eindhoven University of Technology Course 2L920 "Process Algebra".

[10] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[11] S. H. J. Bos and M. A. Reniers. *The I^2C-bus in discrete-time process algebra.* Science of Computer Programming, **29**(1–2):235–258, 1997.

[12] ITU-T. *ITU-T Recommendation Z.100: Specification and Description Language (SDL).* ITU-T, Geneva, 1988.

[13] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC).* ITU-TS, Geneva, 1993.

[14] ITU-TS. *ITU-TS Draft Recommendation Z.120: Message Sequence Chart 1996 (MSC96).* ITU-TS, Geneva, 1996.

[15] C. P. J. KOYMANS AND J. L. M. VRANCKEN. *Extending process algebra with the empty process ε.* Technical Report LGPS 1, Utrecht University, Department of Philosophy, 1985.

[16] K. V. NORI, editor. *Foundations of Software Technology and Theoretical Computer Science.* Number 287 in Lecture Notes in Computer Science. Springer-Verlag, 1987. Proceedings of the 7th conference on the foundations of software technology and theoretical computer science, Pune, India, December 1987.

[17] H. REICHEL, editor. *FCT '95, International Conference on Fundamentals of Computation Theory.* Number 965 in Lecture Notes in Computer Science. Springer-Verlag, 1995. Proceedings of FCT '95, 10th International Conference, Dresden, Germany, August 1995.

[18] M. A. RENIERS. *Message Sequence Chart: Syntax and Semantics.* Ph.D. thesis, Eindhoven University of Technology, 1998. In preparation; title tentative.

[19] M. A. RENIERS AND J. J. VEREIJKEN. *Completeness in discrete-time process algebra.* Technical Report CSR 96/15, Eindhoven University of Technology, Computing Science Department, 1996. Parts also appeared as Chapters 4 and 5 of [20].

[20] J. J. VEREIJKEN. *Discrete-Time Process Algebra.* Ph.D. thesis, Eindhoven University of Technology, 1997.

[21] J. L. M. VRANCKEN. *Studies in Process Algebra, Algebraic Specifications and Parallelism.* Ph.D. thesis, University of Amsterdam, 1991.

[22] J. L. M. VRANCKEN. *The algebra of communicating processes with empty process.* Theoretical Computer Science, **177**(2):287–328, 1997. Also appeared as Chapter 1 of [21].

Paul and the Others : 1 + 155

J.W. de Bakker
CWI - SEN

1. The Scene

1.1 Paul

On September 1, 1972, Paul Klint was hired by the Mathematisch Centrum as 'wetenschappelijk assistent'. He was promoted to 'wetenschappelijk medewerker' on December 1, 1973. In the spring of 1979, his appointment was made permanent. On March 30, 1982, he defended his Ph.D. thesis 'From Spring to Summer : Design, Definition and Implementation of Programming Languages for String Manipulation and Pattern Matching' at the -then- Technische Hogeschool Eindhoven. Promotores were prof.dr. F.E.J. Kruseman Aretz and prof. H. Whitfield. During 1982 he was part-time seconded at the Ministry of Education and Sciences, where he participated in the preparation of the 'Informatica-stimuleringsplan'. From June 1, 1983 till September 30, 1984, he was (acting) deputy department head. From March 1985 onward he was leader of the group Extensible Programming Environments, and since January 1997 he is leader of the theme Interactive Software Development and Renovation.
On November 1, 1985, Paul Klint was appointed as -part-time- full professor of Computer Science at the University of Amsterdam (specialization 'Gespreide Systemen'). Since then, he has acted as promotor for more than 20 Ph.D.'s.

1.2 The Others

Ever since his arrival on the scene at the MC/CWI *, Paul has taken a strong and informed interest in his environment : in his colleagues in the department, in the fate of our institute as a whole, and in the development of computer science and its applications in the Netherlands and abroad.

It so happens that the 25 year period of Paul's stay at the CWI almost coincides with the life time of the CWI's Department of Informatics/Software Technology. Below, we list the precise development of the various ancestors of the present cluster SEN:

- Till December 1972: Computation Department (then called the 'Rekenafdeling', in existence since 1946, headed by prof. A. van Wijngaarden, and covering both Computer Science and Numerical Mathematics)
- January 1973 till February 1985: Department of Informatics
- March 1985 till December 1996: Department of Software Technology (in March 1985, two new departments, on Algorithmics and Architecture, and on Interactive Systems, were formed as well)
- January 1997 - : Cluster Software Engineering.

---

*) In 1983 the Mathematisch Centrum changed its name to CWI ('Centrum voor Wiskunde en Informatica'); below, we solely use the name 'CWI'.

On September 1, 1972, 14 computer scientists were members of the Computation Department, section Informatics. Since then, the respective departments listed above have hired Paul Klint and 141 further researchers. Thus, altogether 155 persons constitute the group of 'others' he has worked with as his departmental colleagues. Naturally, the majority of these have by now left us. My aim with this article is to provide a complete account of their destinations. * For Paul, this may provoke some pondering about the destiny of his colleagues who have moved on to new territories. For the author - who has had the privilege of being reponsible for the successive departments since January 1, 1973 - it triggers many memories of the numerous staff members he has seen coming and going. For the CWI, it provides input for the continuous process of reflection on its mission. The 'production' of scientists is one of our main goals; we hope that the data to follow may help in determining to what extent we have been successful in this endeavour.

2. Academic Positions

2.1 Permanent

P. Asveld	(UT)
R. Back	(Abo Akademi, SF)
J. Baeten	(TUE)
J. Bergstra	(UvA/RUU)
M. Bezem	(RUU)
F. de Boer	(RUU)
R. Bol	(Uppsala U., S)
A. de Bruin	(EUR)
W. Böhm	(Colorado State U., USA)
N. van Diepen	(KUN)
J. Ebergen	(Waterloo U., CDN)
A. Eliëns	(VUA)
W. Fokkink	(Swansea U., UK)
L. van der Gaag	(RUU)
D. Grune	(VUA)
J. Jacquet	(Namur U., B)
J. Kok	(RUL)
P. Lucas	(RUU)
E. Marchiori	(Venice U., I)
A. Middeldorp	(Tsukuba U., J)
S. Mullender	(UT)
A. Ponse	(UvA)
J. Rekers	(RUL)
P. Rodenburg	(UvA)
W. de Roever	(Kiel U., D)
M. Sint	(OU)
A. Tanenbaum	(VUA)

*) I thank ms. A. van den Berg for her assistance with collecting the data.

J. Tucker	(Swansea U., UK)
F. Vaandrager	(KUN)
J. van Vliet	(VUA)
J. Zucker	(McMaster U., CDN)

Persons who have a present (joint) position at the CWI have not been listed here.
Out of the 31 scientists listed above, 12 have a position as full professor (8 in The Netherlands, 4 abroad), and 19 have some other academic rank (12 in The Netherlands, 7 elsewhere). The geographic distribution of the positions is the following: 4 VUA, 4 RUU, 3 UvA, 2 RUL, 2 KUN, 2 UT, 1 EUR, 1 TUE, 1 OU, 11 abroad.

2.2 Temporary

G. Barthe	(Chalmers U., S)
M. Bonsangue	(RUL)
F. van Breugel	(McGill U., CDN)
R. van Glabbeek	(Stanford U., USA)
B. Jacobs	(KUN)
J. Jaspars	(UvA)
E. Marchiori	(CWI/RUL)
W. Meyer Viol	(Imperial College, UK)
M. de Rijke	(Warwick U., UK)
D. Turi	(Edinburgh U., UK)
F.J. de Vries	(ETL, Tsukuba, J)

In accordance with the present shortage of vacancies at Dutch computer science departments, the majority of this group has found employment abroad. As to the dutch members, two have SION postdoc positions, one a KNAW fellowship, and one has an academic teaching position.

2.3 Academic exit positions

Here we list persons who took up some academic job upon exit from the CWI. For most of them, their present position is not known to us - though we believe that the majority of them have left the academic world.

H. Boom	(Canadian university)
S. van Egmond	(VUA)
T. Fernando	(Stuttgart U., D)
J. van der Grinten	(RUL)
F. Heeman	(VUA)
G. Kok	(ZWO/UvA)
J. Mulder	(UvA)
V. Stebletsova	(NWO/RUU)
G. ten Velden	(VUA)
W. van Waning	(UvA)
J. Warmer	(VUA)

3. Non-academic positions

3.1 Positions in industry

H. Bakker	(BSO/Origin)
D. Bosscher	(ID Research)
P. Bouvry	(FICS, B)
J. Ganzevoort	(ACE)
T. Hagen	(OCE)
P. Hendriks	(SERC)
A. Janssen	(Mabel Systems)
H. Jonkers	(Philips Research)
J. Kamperman	(ID Research)
S. Klusener	(CAP Gemini)
H. Korver	(ID Research)
L. Kossen	(AI Technology)
A. Lenstra	(AT&T Research/Citibank, USA)
E. van der Meulen	(Mees Pierson)
J. Spee	(KPN Research)
G. van Rossum	(Corporation for National Research Institutes, USA)
K.E. Shahroudi	(Woodward Governor)
F. Teusink	(BSO/Origin)
F. Tip	(IBM Research, USA)
A. Veen	(Consultant)
J. Warmerdam	(KPN Research)
P. Weijland	(KPN)
C. Zuidema	(IBM)

In the first half of the considered period -say till 1985 - only five persons left the CWI for industry. After 1985, this number increased sharply to 18. Out of these, 7 worked directly for Paul Klint, and several more experienced close interaction with him.

3.2 Persons on secondment from industry

M. van Dijk	(BSO)
W. Koorn	(BSO)
M. Logger	(BSO)
A. Verhoog	(BSO)

These were all involved in some way in one of Paul's ESPRIT projects.

3.3 Semi-government/others

L. Ammeraal	(Higher Professional Education)
E. Kuypers	(National Aerospace Laboratory)
J. van Vaalen	(Higher Professional Education)
R. van Vliet	(Dutch Library for the Blind)

We note that this group is remarkably small. E.g., one might have expected more CWI staff members to have found positions as HBO teachers.

4. Visitors

Below, we list a variety of visitors, almost all from abroad. Somewhat arbitrarily, we define this category as
- duration of stay between 6 and 24 months
- paid by non-CWI sources (e.g. a national scholarship, industrial grant, sabbatical, EU fellowship, ...; occasionally, some CWI support was given as well).

H. Barendregt	(NL, KUN)
A. Bouali	(F)
C. Brovedani	(I)
J. Carr III	(USA)
A. Corradini	(I)
S. Eker	(UK)
S. Etalle	(I)
M. Gabbrielli	(I)
P. Gianantonio	(I)
H. Goeman	(NL, RUL)
E. Horita	(J)
Y. Korablin	(USSR)
M. Marchiori	(I)
D. Naidich	(CIS/USA)
E. Olderog	(D)
C. Palamidessi	(I)
D. Plump	(D)
D. Rosenthal	(UK)
W. Rounds	(USA)
W. Savitch	(USA)
A. Schaerf	(I)
Y. Toyama	(J)
J. Villadsen	(DK)
H. Walinska	(PL)
H. Wiklicky	(A)

Some statistics as to country of origin:

Italy	8
USA	3
UK	2
Germany	2
Japan	2
NL	2

and several countries with 1 representative. We observe the - for SEN members well known - phenomenon that we have by far the closest ties with Italy. We conjecture that this stems from a combination of personal/historical factors, and from the fact that the Italian government strongly promotes foreign visits for its young researchers.

5. Present CWI staff members

Of course, many persons who entered the CWI in the 25 years from 1972 till 1997 are still employed by us. We list them below:

5.1 Present SEN members (per 010997)

F. Arbab
J. de Bakker
I. Bethke
C. Blom
F. Burger
A. van Deursen
T. Dinesh
H. Elbers
C. Everaars
D. Griffioen
J.F. Groote
J. Harkes
J. Heering
M. Hoogenboom
C. van Kemenade
J.W. Klop
J. Kniesmeijer
T. Kuipers
W. de Leeuw
R. van Liere
B. Lisser
B. Luttik
E. Montfroy
J. Mulder
H. La Poutré
J. Romijn
J. Rutten
A. Scutellà
J. Springintveld
R. van Stee
P. Walters
J. van Wamel
J. Warners
M. van Wezel

Of these, 11 have permanent positions, and 2 have been seconded by CAP Gemini.

5.2 Present CWI - non SEN members (per 010997)

K. Apt
J. van Eijck
A. Groenink
P. ten Hagen
F. Kuijk
H. Noot
L. Meertens
S. Pemberton
M. de Ruiter
F. van Raamsdonk
P. Vitányi

6. Unknown

This category is very small: only one researcher who once entered SEN or its predecessor departments since September 1972 has left with an unknown destination;

J. Rukkers, 1987-1988

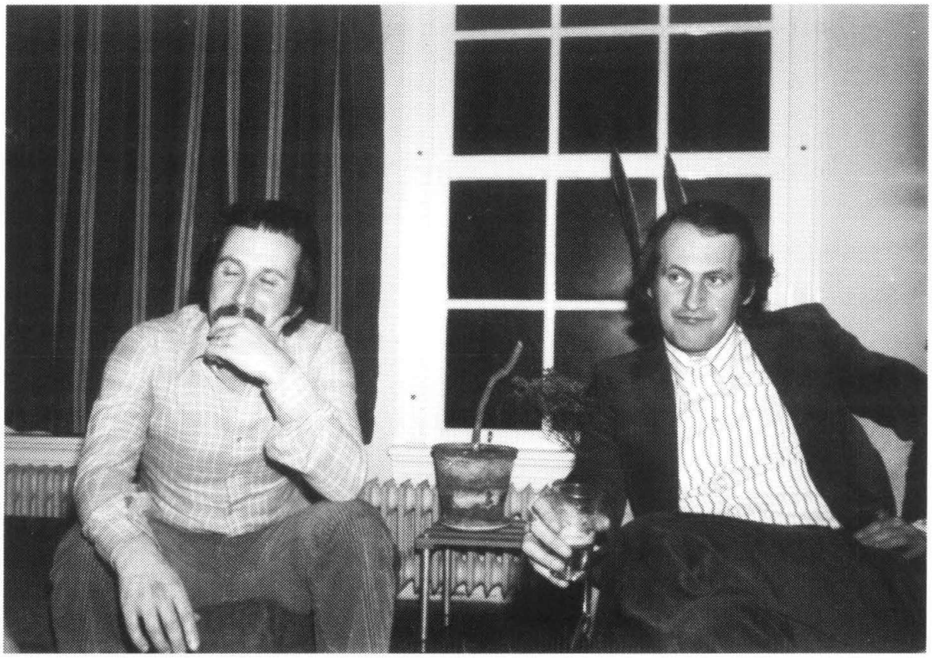
7. Conclusions

Out of the 155 'others', 45 are still employed by the CWI. From the 110 persons who have left our institute, about two thirds have some - permanent or temporary - university job, and about one third has a position elsewhere (mostly in industry).

Moreover, we see
- the number of persons going to industry is rapidly increasing
- permanent positions at Dutch universities have become scarce
- 18 out of the 155 'others' are full professor
- throughout, a sizable proportion of the SEN (and its predecessors) staff has been filled by various kinds of visitors
- for all but one of the 155 persons it is known where (s)he went upon leaving the CWI.

So much about the 155, and now about the 1:
- in numerous ways, as project leader, group leader, theme leader, deputy department head, scholar, drummer, promotor, 'our man in The Hague', ..., Paul Klint has played a central role in the lifes of many of the above, thus contributing significantly to the achievements of our institute.

On behalf of SEN and all its forerunners, I express my warmest thanks to Paul for all he has done for us.



Process Algebra Primitives for File Transfer*

Jan A. Bergstra[1,2] Alban Ponse[1]

[1] *University of Amsterdam, Programming Research Group*
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

[2] *Utrecht University, Department of Philosophy*
P.O. Box 80126, 3508 TC Utrecht, The Netherlands

E-mail: `janb@wins.uva.nl` - `alban@wins.uva.nl`

Abstract

Action primitives for sequential file transfer are presented in the style of the early input action of [BB94].

1 Introduction

In modeling distributed systems one might associate infinite data types with 'local computation', and finite data types with information to be 'distributed'. For example, transferring a natural `N` from one location to another, say with help of an action `send(N)`, should not be regarded primitive in case `N` is a string of very large length over alphabet $\{0, 1, ..., 9\}$ (e.g., a representation of 9^{9^9} by juxtaposition of digits). For specification of (concurrent) transfer of files of arbitrary finite length, we propose new process algebra primitives. These are based on the finite alphabet from which 'files' are constructed, be it `ASCII`, $\{0, 1, ..., 9\}$, bits, or bytes.

In the next section we recall process algebra with early input. Then, in Section 3, we introduce our primitives for file transfer. In Section 4 we shortly discuss a notion of file manipulation primitives. The paper is ended with a short reflection on file transfer and manipulation in the realm of the ToolBus coordination architecture.

*Dedicated to Paul Klint. His activities, professional both technical and managerial as well as personal have been appreciated very much by both authors, and have proven effective and useful for both of them.

2 Early Input, Process Prefix, and SOS

In process algebra applications it is often useful to have specialised actions for various roles. With D a (finite) set of data items and I a collection of port names the following notations have become standard ([BK86]): for $d \in D$, $i \in I$

$s_i(d)$: send d along port i,

$r_i(d)$: receive d along port i,

$s_i(d)$: communicate d along port i.

The communication $s_i(d) \mid r_i(d) = c_i(d)$ describes the connection between these actions.

Let x, y, z be variables ranging over D. Process expressions with parameters in D are often denoted $P(x)$, $Q(x,y)$ etc. As an example, consider

$$P(x,y,z) = (s_i(x) \parallel s_j(y)) \cdot s_k(z)$$

and

$$Q(x,y) = r_i(x)^* s_j(d_0) \qquad (d_0 \in D, y \text{ need not occur in } Q(x,y)),$$

where the $\_{}^*\_$ operation is the binary Kleene star, defined by

$$P^*Q = P(P^*Q) + Q,$$

([Kle56], see also [BBP94]).

The expression $[e/x]P$ denotes the result of substituting e for x in a head normal form of P,

$$[e/x](\textstyle\sum_{i\in K} a_i P_i + \sum_{j\in L} b_j) = \sum_{i\in K}[e/x]a_i \cdot [e/x]P_i + \sum_{j\in L}[e/x]b_j.$$

With $\sum_{d\in D} r_i(d) \cdot [d/x]P$ we obtain a process which reads a value $d \in D$ along port i and thereupon transforms itself into $[d/x]P$. This kind of process occurs very frequently indeed, whence [BB94] proposed the notation

$$er_i^D(x); P = \textstyle\sum_{d\in D} r_i(d) \cdot [d/x]P.$$

Here er_i^D is an *early read* action, and $\_\,;\,\_$ is the *action prefix operation.* Of course, $[e/x]er_i(x); P = er_i(x); P$, as x is in this case not free for substitution.

Action prefix is extended to *process prefix* satisfying for action set A:

$$
\begin{array}{rcll}
(x+y);z & = & x;z+y;z & \\
(a\cdot x);y & = & a;(x;y) & (a\in A)\\
a;x & = & a\cdot x & (a\in A\setminus\{\text{early read actions}\}).
\end{array}
$$

A nice bonus of the early input notation is that parallel input is easily described.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\left(er_1^D(x) \parallel er_2^D(y)\right); P(x,y)}\\
= & \left(er_1^D(x)\cdot er_2^D(y)\ + er_2^D(y)\cdot er_1^D(x) + \delta\right); P(x,y)\\
= & \left(er_1^D(x)\cdot er_2^D(y)\right); P(x,y)\ +\\
 & \left(er_2^D(y)\cdot er_1^D(x)\right); P(x,y)\\
= & er_1^D(x); (er_2^D(y); P(x,y))\ +\\
 & er_2^D(y); (er_1^D(x); P(x,y))\\
= & \sum_{d\in D} r_1(d)\cdot \sum_{e\in D} \cdot r_2(e)\cdot P(d,e)\ +\\
 & \sum_{e\in D} r_2(e)\cdot \sum_{d\in D} \cdot r_1(d)\cdot P(d,e).
\end{array}
$$

Indeed parallel input can be expressed without early input actions, but at cost of a combinatorial explosion. Expressions like

$$\left(er_1^D(x_1) \parallel \ldots \parallel er_n^D(x_n)\right); P(x_1,\ldots,x_n)$$

are instrumental for the description of data flow networks and grid protocols (see [BHP97], inspired by [TT94]). Avoiding early input actions at cost of expanding the merge is hardly a practical option.

Finally, we display in Table 1 transition rules that can be used to assign a transition system to process expressions. In this table, P and Q, and their primed versions range over process expressions with parameters in data set D.

3 Primitives for File Transfer

We now turn attention to larger pieces of data which cannot be transferred in one step. We consider D^*, writing Λ for the empty sequence, $\_ +\!\!+ \_$ for strong concatenation, $\_ \frown \_$ for left insertion of a D-element in a sequence, and $\_ \smallfrown \_$ for append (right insertion). We use X as a variable ranging over D^*.

Table 1: Transition rules for process algebra with early input.

	$a \xrightarrow{a} \surd \quad (a \in A \setminus \{\text{early read actions}\})$	
$er_i^D(x); P$	$er_i^D(x); P \xrightarrow{r_i(d)} [d/x]P \quad (d \in D)$	
$+, \cdot, \lfloor\!\lfloor, \Vert, {}^*$	$\dfrac{P \xrightarrow{a} \surd}{\begin{array}{c} P+Q \xrightarrow{a} \surd \\ Q+P \xrightarrow{a} \surd \\ P \cdot Q \xrightarrow{a} Q \\ P \lfloor\!\lfloor Q \xrightarrow{a} Q \\ P \Vert Q \xrightarrow{a} Q \\ Q \Vert P \xrightarrow{a} Q \\ P^*Q \xrightarrow{a} P^*Q \\ Q^*P \xrightarrow{a} \surd \end{array}}$	$\dfrac{P \xrightarrow{a} P'}{\begin{array}{c} P+Q \xrightarrow{a} P' \\ Q+P \xrightarrow{a} P' \\ P \cdot Q \xrightarrow{a} P'Q \\ P \lfloor\!\lfloor Q \xrightarrow{a} P' \Vert Q \\ P \Vert Q \xrightarrow{a} P' \Vert Q \\ Q \Vert P \xrightarrow{a} Q \Vert P' \\ P^*Q \xrightarrow{a} P'(P^*Q) \\ Q^*P \xrightarrow{a} P' \end{array}}$
$\mid, \Vert$ $a \mid b = c \in A$	$\dfrac{P \xrightarrow{a} \surd \quad Q \xrightarrow{b} \surd}{\begin{array}{c} P \mid Q \xrightarrow{c} \surd \\ P \Vert Q \xrightarrow{c} \surd \end{array}}$	$\dfrac{P \xrightarrow{a} \surd \quad Q \xrightarrow{b} Q'}{\begin{array}{c} P \mid Q \xrightarrow{c} Q' \\ P \Vert Q \xrightarrow{c} Q' \end{array}}$
	$\dfrac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} \surd}{\begin{array}{c} P \mid Q \xrightarrow{c} P' \\ P \Vert Q \xrightarrow{c} P' \end{array}}$	$\dfrac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{\begin{array}{c} P \mid Q \xrightarrow{c} P' \Vert Q' \\ P \Vert Q \xrightarrow{c} P' \Vert Q' \end{array}}$
∂_H	$\dfrac{P \xrightarrow{a} \surd}{\partial_H(P) \xrightarrow{a} \surd}$ if $a \notin H$	$\dfrac{P \xrightarrow{a} P'}{\partial_H(P) \xrightarrow{a} \partial_H(P')}$ if $a \notin H$

Sending or receiving a file $\sigma \in D^*$ will require one action per item in σ. For *file output* we propose the following notation:

ofs_i^D : open file send along port i, an action that initiates sending a file over D along port i,

$Sf_i^D(\sigma)$: a process that represents the actual sending of the elements of file σ over D, specified by:

$$\begin{aligned} Sf_i^D(\Lambda) &= cfs_i^D, \\ Sf_i^D(d^\frown\sigma) &= s_i(d) \cdot Sf_i^D(\sigma), \end{aligned}$$

cfs_i^D : close file send along port i, an action indicating that end of file (EOF) has been reached.

For *file input* we use

ofr_i^D : open file receive along port i, an action that marks the start of a file transmission,

cfr_i^D : close file receive along port i, an action indicating that end of file (EOF) has been reached.

Then, similar to the early input case we propose the following primitives for *early file input*:

$$\begin{aligned} ofr_i^D(X); P &= ofr_i^D \cdot [\Lambda/X]P, \text{ i.e., bind } X \text{ in } P \text{ for file input,} \\ efr_i^D(X); P &= \textstyle\sum_{d \in D} r_i(d) \cdot [X^\frown d/X]P, \\ & \qquad \text{i.e., append in } P \text{ the next item to } X, \\ cfr_i^D(X); P &= cfr_i^D \cdot P. \end{aligned}$$

Here the substitution $[X^\frown d/X]$ must be handled with care: it does not affect $efr_i^D(X)$ but substitutes $X^\frown d$ for X in all other occurrences. So, unlike the case of the early input $er_i^D(x); P$ where x can be considered bound, X is not bound in $efr_i^D(X); P$.

As a useful macro we have

$$Efr_i^D(X) \stackrel{\text{def}}{=} efr_i^D(X)^* cfr_i.$$

Writing $\overline{P(X)} = Efr_i^D(X); P(X)$ we can derive

$$\begin{aligned} \overline{P(X)} \;=\; & cfr_i^D \cdot P(X) + \textstyle\sum_{d \in D} r_i(d) \cdot (\\ & \quad cfr_i^D \cdot [X^\frown d/X]P + \textstyle\sum_{e \in D} r_i(e) \cdot (\\ & \qquad cfr_i^D \cdot [X^\frown d^\frown e/X]P) \; + \ldots \\ &)). \end{aligned}$$

As an example, consider the following derivation:

$$
\begin{aligned}
&[X^\frown d_0/X]\overline{P(X)}\\
&\quad = [X^\frown d_0/X]\left(cfr_i^D \cdot P(X) + \textstyle\sum_{d\in D} r_i(d)\cdot [X^\frown d/X]\overline{P(X)}\right)\\
&\quad = cfr_i^D \cdot P(X^\frown d_0) + \textstyle\sum_{d\in D} r_i(d)\cdot [X^\frown d_0/X][X^\frown d/X]\overline{P(X)}\\
&\quad = cfr_i^D \cdot P(X^\frown d_0) + \textstyle\sum_{d\in D} r_i(d)\cdot [X^\frown d_0{}^\frown d/X]\overline{P(X)}.
\end{aligned}
$$

Now $ofr_i^D(X)$ acts as a binder, and in the most useful combination

$$\left(ofr_i^D(X)\cdot Efr_i^D(X)\right); P(X)$$

X is bound indeed. Let H be such that communications along channel i are enforced:

$$H \;=\; \{ofs_i^D, ofr_i^D, cfs_i^D, cfr_i^D\} \;\cup\; \{s_i(d), r_i(d) \mid d \in D\}.$$

Let furthermore $\sigma = d_1{}^\frown d_2 ...{}^\frown d_n$, then we obtain

$$
\begin{aligned}
&\partial_H(ofs_i^D \cdot Sf_i(\sigma) \parallel (ofr_i^D(X)\cdot Efr_i^D(X)); P(X))\\
&\quad = ofc_i^D \cdot \partial_H(Sf_i(\sigma) \parallel [\Lambda/X]\overline{P(X)})\\
&\quad = ofc_i^D \cdot c_i(d_1)\cdot ... \cdot c_i(d_n)\cdot cfc_i^D \cdot P(d_1{}^\frown d_2 ...{}^\frown d_n).
\end{aligned}
$$

For the specific case $\sigma = d_1{}^\frown d_2 \in D^*$ and $P(X) = ofs_j^D \cdot Sf_j^D(X^R)$, we derive this identity in full detail in Table 2. Here σ^R is the *reversal* of σ, defined by $\Lambda^R = \Lambda$, $(\sigma^\frown d)^R = d^\frown \sigma^R$.

For these file transfer primitives we give the transition rules displayed in Table 3. For $Efr_i^D(X); P$ we can derive the following transitions:

$$
\begin{aligned}
&Efr_i^D(X); P \xrightarrow{ofr_i^D} [\Lambda/X]P\\
&Efr_i^D(X); P \xrightarrow{r_i(d)} [X^\frown d/X]Efr_i^D(X); P \qquad (d \in D).
\end{aligned}
$$

Table 2: Derivation of a file transfer.

$$
\begin{array}{l}
\partial_H(ofs_i^D \cdot Sf_i(d_1{}^\frown d_2) \parallel (ofr_i^D(X) \cdot Efr_i^D(X);[ofs_j^D \cdot Sf_j^D(X^R)]) \\
\quad = \quad ofc_i^D \cdot \partial_H(Sf_i(d_1{}^\frown d_2) \parallel [\Lambda/X]Efr_i^D(X);[ofs_j^D \cdot Sf_j^D(X^R)]) \\
\quad = \quad ofc_i^D \cdot \\
\qquad \partial_H\left(Sf_i(d_1{}^\frown d_2) \parallel \left[\begin{array}{l} cfr_i^D \cdot ofs_j^D \cdot Sf_j^D(\Lambda) \\ + \\ \sum_{d\in D} r_i(d) \cdot [d/X]Efr_i^D(X);[ofs_j^D \cdot Sf_j^D(X^R)] \end{array}\right]\right) \\
\quad = \quad ofc_i^D \cdot c_i(d_1) \cdot \partial_H(Sf_i(d_2) \parallel [d_1/X]Efr_i^D(X);[ofs_j^D \cdot Sf_j^D(X^R)]) \\
\quad = \quad ofc_i^D \cdot c_i(d_1) \cdot \\
\qquad \partial_H\left(Sf_i(d_2) \parallel \left[\begin{array}{l} cfr_i^D \cdot ofs_j^D \cdot Sf_j^D(d_1) \\ + \\ \sum_{d\in D} r_i(d) \cdot [d_1{}^\frown d/X]Efr_i^D(X);[ofs_j^D \cdot Sf_j^D(X^R)] \end{array}\right]\right) \\
\quad = \quad ofc_i^D \cdot c_i(d_1) \cdot c_i(d_2) \cdot \partial_H(cfs_i^D \parallel [d_1{}^\frown d_2/X]Efr_i^D(X);[ofs_j^D \cdot Sf_j^D(X^R)]) \\
\quad = \quad ofc_i^D \cdot c_i(d_1) \cdot c_i(d_2) \cdot \\
\qquad \partial_H\left(cfs_i^D \parallel \left[\begin{array}{l} cfr_i^D \cdot ofs_j^D \cdot Sf_j^D(d_2{}^\frown d_1) \\ + \\ \sum_{d\in D} r_i(d) \cdot [d_1{}^\frown d_2{}^\frown d/X]Efr_i^D(X);[ofs_j^D \cdot Sf_j^D(X^R)] \end{array}\right]\right) \\
\quad = \quad ofc_i^D \cdot c_i(d_1) \cdot c_i(d_2) \cdot cfc_i^D \cdot ofs_j^D \cdot Sf_j^D(d_2{}^\frown d_1).
\end{array}
$$

Table 3: Transition rules for process algebra with early file input.

$ofr_i^D(X);P$	$ofr_i^D(X);P \xrightarrow{ofr_i^D} [\Lambda/X]P$
$efr_i^D(x);P$	$efr_i^D(X);P \xrightarrow{r_i(d)} [X{}^\frown d/X]P \qquad (d \in D)$

4 File Manipulation Primitives

File transfer primitives will naturally lead to file manipulation primitives.

Let $F : V \to W_{\mathtt{D}}$ ($W_{\mathtt{D}} = W \cup \{\mathtt{D}\}$), with $F(v) = \mathtt{D}$ representing the case that the computation $F(v)$ diverges. The special constant $\mathtt{D}$ is taken from [BBR95]. We write:

$$\begin{array}{lll} Y_{F,j}(v) = \iota^2 \cdot s_j(w) & \text{if} & F(v) = w \in W, \\ Y_{F,j}(v) = \iota^\omega & \text{if} & F(v) = \mathtt{D}. \end{array}$$

Here Y stands for "yield", and ι is an action that represents internal activity (ι satisfies the law $x \cdot \iota = x \cdot \iota \cdot \iota$).

Now $er_k(X); Y_{F,j}(X)$ represents the function F as a process. For X of sort D^* we use $Efr_k^D(X); Y_{F,j}(X)$.

For a nondeterministic operation (relation) R it is not so easy to find a general form for representing it as a process. We have to deal with a computation process that may diverge as well as deliver different values (along different paths). In special cases however something can be said. We will discuss an example involving the arbitrary (random) permutation of lists in advance of the computation of output. For instance the action prefix $Aperm^D(X)$ can be defined as follows:

$$Aperm^D(X); P = \sum_{r:perm(X)} \iota^2 \cdot [r(X)/X]P.$$

Here $perm(X)$ is the set of permutations of the list X. Clearly $|perm(X)| = length(X)!$.

If F computes a value in W from a sequence X of D's, then

$$\left(ofr_i^D(X) \cdot Efr_i^D(X) \cdot Aperm^D(X) \cdot \iota\right); Y_{F,j}(X)$$

is a process that opens file X, and reads its values until end of file along port i. Thereafter it generates an arbitrary permutation, evaluates function F on that permutation, and outputs the result along port j.

Of course this operation has many variations, e.g., producing even permutations ($Aperm^D_{even}(X)$), or cyclic permutations ($Aperm^D_{cyclic}(X)$) only.

5 Digression

In case it is known that a file to be received has a bound on its length, we can use much more specific operations, both for modeling its representation, and for modeling its transfer. We define *bounded repetition* $x^n y$ by

$$\begin{aligned} x^0 y &= y, \\ x^{n+1} y &= x \cdot (x^n y), \end{aligned}$$

and *bounded iteration* (or bounded binary Kleene star) $x^{*^n} y$ by

$$\begin{aligned} x^{*^0} y &= y, \\ x^{*^{n+1}} y &= x \cdot (x^{*^n} y) + y. \end{aligned}$$

For a file that has exactly length n one may use the prefix

$$\left(efr_i^D (X)^n cfr_i^D \right) ; P.$$

In the case that one knows that a file to be received has at most n elements, the prefix

$$\left(efr_i^D (X)^{*^n} cfr_i^D \right) ; P$$

will work. Though such modeling will be closer to reality, it may still be worthwhile to consider $\_ ^* \_$ and $\_ ^* \delta$ as the 'limit' cases of $x^{*^n} y$ and $x^n y$, respectively. Proving a certain property defined in terms of these more abstract operations might be easier. Still, such a property might give one sufficient information about properties that are expressed with the specific operations discussed above.

Giving a realistic example of the use of our file transfer primitives is not that easy. Clearly in Paul Klint's favorite subject (distributed programming environments, [Kli93]), files of "arbitrary" length play a key role as declarations, programs, variable environments, typechecker outputs etc. are all character strings, say from type `C`. A component like

$$\begin{aligned} &\left(ofr_1^{\mathtt{C}} \cdot Efr_1^{\mathtt{C}}(\mathtt{decls}) \parallel ofr_2^{\mathtt{C}} \cdot Efr_2^{\mathtt{C}}(\mathtt{stats}) \right) ; \\ &\qquad \left(ofs_3^{\mathtt{C}} \cdot Sf_3^{\mathtt{C}}(\mathtt{decls} +\!\!\!+ \mathtt{stats}) \parallel ofs_4^{\mathtt{C}} \cdot Sf_4^{\mathtt{C}}(\mathtt{type\text{-}check}(\mathtt{decls} +\!\!\!+ \mathtt{stats})) \right) \end{aligned}$$

can input in parallel strings like `decls` (declarations) and `stats` (statements), and return an encoded parse tree and a file with type check output.

The ToolBus model using process algebra as given in [BK96a, BK96b] is an approximation of reality rather than a dogma. It so seems that this model would get closer to the facts if the communication between bus and tools were described by means of the file transfer primitives established above, rather than the atomic action used in [BK96a].

References

[BB94] J.C.M. Baeten and J.A. Bergstra. On sequential composition, action prefixes and process prefix. *Journal of Formal Aspects of Computing Science*, 6(3):83–98, 1994.

[BBP94] J.A. Bergstra, I. Bethke, and A. Ponse. Process algebra with iteration and nesting. *Computer Journal*, 37(4):243–258, 1994.

[BBR95] J.A. Bergstra, I. Bethke, and P.H. Rodenburg. A propositional logic with 4 values: true, false, divergent and meaningless. *Journal of Applied Non-Classical Logics*, 5:199–217, 1995.

[BHP97] J.A. Bergstra, J.A. Hillebrand, and A. Ponse. Grid protocols based on synchronous communication, *Science of Computer Programming*, 29:199-233, 1997.

[BK86] J.A. Bergstra and J.W. Klop. Process algebra: specification and verification in bisimulation semantics. In M. Hazewinkel, J.K. Lenstra, and L.G.L.T. Meertens, editors, *Mathematics and Computer Science II*, CWI Monograph 4, pages 61–94. North-Holland, Amsterdam, 1986.

[BK96a] J.A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION), LNCS 1061*, pages 75–88. Springer-Verlag, 1996.

[BK96b] J.A. Bergstra and P. Klint. The discrete time ToolBus. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST), LNCS 1101*, pages 286–305. Springer-Verlag, 1996.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, pages 3–41. Princeton University Press, Princeton NJ, 1956.

[Kli93] P. Klint. A meta-environment for generating programming environments. ACM Transactions on Software Engineering and Methodology, 2(2):176–201, 1993.

[TT94] B.C. Thompson and J.V. Tucker. Equational specification of Synchronous Concurrent Algorithms and architectures (Second Edition). Report CSR 15-94, University of Wales, Swansea, 1994.

Looking back

Inge Bethke   Jan Willem Klop   Roel de Vrijer

$\triangleright \triangleright \triangleright$ *Dedicated to Paul Klint* $\triangleleft \triangleleft \triangleleft$

In general our interest in term rewriting and lambda calculus is directed forward. The confluence property states that diverging reductions can be prolonged to converging ones—in forward direction. The termination property rules out infinite (forward) computations. We can also look at a more microscopic level at reductions and discern how parts of the expressions 'descend' from earlier ones, as particles in a stream are connected in the flow of computation. The formalization of this intuitive notion is well-known since the dawn of lambda calculus and combinatory logic: it is the notion of *descendant* (or residual) with its inverse notion of ancestor.

Usually we are interested in the forward notion of descendants, such as in the Parallel Moves Lemma. But sometimes it is rewarding to look back and concentrate on ancestors rather than descendants. In particular we are interested in *tracing* the symbols in a reduced term back to their origins in the initial term. In recent terminology this is what has been called *origin tracking* and in the group of Paul Klint this technique has proved to have many useful practical applications such as error recovery and program slicing.

In the present note, which will remain of an informal character, we will apply this technique of origin tracking in a theoretical context. Specifically, we will describe how it can be used to obtain a fundamental theorem of Huet and Lévy. Their original proof is rather intricate, employing the notion of standard reductions for orthogonal rewrite systems. We feel that the proof outlined in this note is a simplification of the proof of Huet and Lévy.

1 The theorem of Huet and Lévy

We will adopt the framework of orthogonal first-order term rewriting systems. So we may assume confluence of the reduction relation and uniqueness of normal forms: each term can have at most one normal form.

The theorem of Huet and Lévy consists of three parts.

THEOREM.

i. Given a term t having a normal form, t will always possess a needed redex. A redex in t is *needed* when in every reduction from t to its normal form, some descendant of that redex must be contracted.

ii. Repeatedly contracting needed redexes must lead eventually to the normal form, provided the original term has a normal form. In other words, needed reduction is a *normalizing* reduction strategy.

iii. Needed reduction is not only normalizing, but even 'hyper-normalizing': there does not exist an infinite reduction of t containing infinitely many steps in which a needed redex is contracted. In other words, even the relaxed notion of needed reduction which allows between needed reduction steps any finite number of arbitrary reductions is normalizing.

EXAMPLE. Consider the well-known orthogonal TRS for addition and multiplication on natural numbers generated by 0 and S. The reduction rules are given in the following table. Now the redex $A(0,0)$ in the term $M(A(0,0),0)$ is

ρ_1	$A(x,0)$	$\to$	x
ρ_2	$A(x,S(y))$	$\to$	$S(A(x,y))$
ρ_3	$M(x,0)$	$\to$	0
ρ_4	$M(x,S(y))$	$\to$	$A(M(x,y),x)$

Table 1: The TRS for addition and multiplication

not needed. It is erased in the reduction to normal form consisting of the single reduction step

$$M(A(0,0),0) \to_{\rho_3} 0.$$

Without putting further restrictions on the orthogonal term rewriting systems that we work with this result is nice but not necessarily very useful, since we cannot always determine what are the needed redexes. The notion of 'needed redex' is formally undecidable. However, Huet and Lévy gave reasonable restrictions that ensured the decidability.

Here we will not discuss this matter. What we are aiming at is a proof of this general theorem by means of origin tracking.

2 The needed prefix

So, consider a term t and a reduction $t \equiv t_0 \to t_1 \to \cdots \to t_n$, where t_n is a normal form, and let us try to determine the part of t_0 that has been 'necessary' in manufacturing t_n. The idea is to look at each symbol s in t_n, and to determine what symbols $s', s'', \ldots$ in t_{n-1} were 'responsible' for the occurrence or appearance of s in t_n. The symbols s', s'' can also be viewed as the 'causes' or *origins* of s. In turn, we trace back the symbols $s', s'', \ldots$ to the previous term t_{n-2}, and so on. In the end we arrive at a bunch of symbol occurrences in t_0 that are the original causes of the symbol s in t_n. Doing this for all symbols in t_n and taking all the 'origins' together, we have the needed part of t_0. At least, it would be nice if this were indeed the case. Our statement might be merely wishful thinking.

Actually we will find that the situation can be summarized as follows.

> The origins in the original term $t \equiv t_0$ of all symbols in t_n will make up a prefix of t.[1]

It will be called the ***needed prefix*** of t, since all redexes having their pattern in the needed prefix are in fact needed.

Moreover, the needed prefix will contain at least one redex (or rather a redex pattern).

Finally, everything in the non-needed, dark part of t is *garbage*; it can be replaced by anything without affecting or altering the normal form t_n.

Here is the picture.

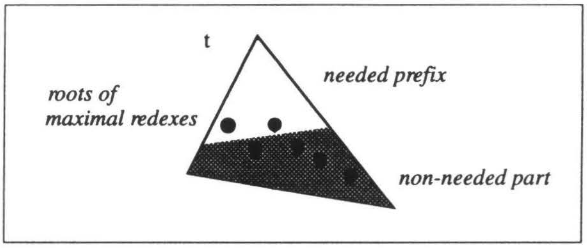


Figure 1: Needed prefix

3 Tracing back

How to define the tracing or tracking relation to find the s', s'' in t_{n-1} that are the origins of s in t_n? By way of example we have visualized in Figure 2 the trac(k)ing relation between symbols in the reduction step

$$S(F(G(0,1),H)) \to S(R(S(1),1)),$$

obtained from the reduction rule

$$\rho: \ F(G(x,y),H) \to R(S(y),y).$$

We now describe the intended tracing relation more precisely. Let $t(\vec{x})$ and $s(\vec{x})$ be terms involving the variables $\vec{x} = x_1, \ldots, x_n$ and let $\rho: \ t(\vec{x}) \to s(\vec{x})$ be a rewrite rule. We call the context $t(\vec{\ })$ obtained by replacing in t the variables $\vec{x}$ by n holes the *redex pattern* of the rewrite rule ρ, and $s(\vec{\ })$ the *contractum pattern* of ρ.

EXAMPLE. So in the rule $\rho: \ F(G(x,y),H) \to R(S(y),y)$ we have that the context $F(G(\Box,\Box),H)$ is the redex pattern and $R(S(\Box),\Box)$ the contractum pattern.

[1] This fact has also been studied by van Oostrom. He noted that it is related to earlier results that were obtained in the context of the Automath project, especially van Daalen's 'square bracket lemma', and to the folklore 'Barendregt's lemma'.

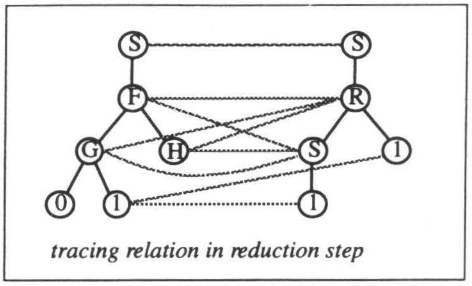


Figure 2: Tracing symbols

Moreover, let $C[t(\vec{x}^\sigma)] \to C[s(\vec{x}^\sigma)]$, be a rewrite step generated by ρ. (Here $C[\,]$ is a context and σ is a substitution.) There are three cases for the position of a symbol to be traced:

i. A symbol in the context $C[\ ]$ of the left-hand side of the rewrite step traces to the same symbol in the right-hand side $C[\,]$. So in Figure 2 the two top S's are connected.

ii. A symbol in x_i^σ in the left-hand side of the rewrite step traces to the same symbol in all the copies of x_i^σ in the right-hand side. So in the figure the 1 in the left-hand side is connected to both 1's in the right-hand side; the 0 is not connected to anything, it is erased.

iii. A symbol in the redex pattern traces to all the symbols of the contractum pattern. In the figure each of the symbols F, G, H is connected to both R and the lower S in the right-hand side.

4 Collapsing rules

There is however a problem with 'collapsing' reduction rules.[2] A reduction rule $t \to r$ is called *collapsing* if the right-hand side r is a mere variable.

EXAMPLES. The rule

$$A(x, 0) \to x$$

from Table 1 is collapsing. And so is the famous rule

$$Kxy \to x$$

from Combinatory Logic.

The definition above of the tracing relation does not provide for this situation, since in a collapsing rule there is no contractum pattern. It is the trivial or

[2]Cf. Stelling 7 in Middeldorp's Ph.D. thesis: "Herschrijfregels waarvan de rechterkant uit een enkele variabele bestaat vormen een groot obstakel in het werk van een termherschrijver".

empty context $\Box$. So where to attach the traces leaving the symbols in the redex pattern?

This a nasty technical problem and we are not aware of a solution that is both simple and natural. There are some tricks to eliminate the problem. One is to forbid collapsing rules altogether, and when they do occur to 'code them away'. This can be done be replacing e.g. the rule $A(x,0) \to x$ by $A(x,0) \to e(x)$, where e is a unary 'dummy' symbol. Then one has to add infinitely many new reduction rules, 'saturating' all left-hand sides with the symbol e.

Other solutions involve not only labeling symbols (nodes in the term tree), as we shall do in the next section, but also branches in the term tree.

In this note we will not enter this matter. For the present we simply rule out collapsing rules.

5 Labels

Actually, we do not find the definition of tracing as described in Section 3 above entirely satisfactory. We would prefer a more 'algebraic' formulation over the verbal or pictorial definitions above. Such an algebraic formulation, in the syntax of term rewriting systems itself, is indeed possible. It is inspired by Lévy's labeled lambda calculus.[3]

Let us first introduce the labels that will be used. They are formed from atomic labels $a, b, c, \ldots$, using the operations of concatenation and underlining. That is, if α is a label, then $\underline{\alpha}$ is a label and if α, β are labels, then $\alpha\beta$ is one. E.g. $\underline{a\underline{\underline{bc}}}ad$ is a composite label. In our notation labels will be attached to symbols as superscripts, but their actual status is that of unary function symbols.

We are now going to decorate rewrite rules with labels.

> Consider the rule from the earlier picture, $F(G(x,y),H) \to R(S(y),y)$. For every label α, β, γ we will have the labeled version:
>
> $$F^{\alpha}(G^{\beta}(x,y),H^{\gamma}) \to R^{\underline{\alpha\beta\gamma}},(S^{\underline{\alpha\beta\gamma}}(y),y).$$
>
> So, every symbol in the redex pattern has some label. All these labels are swept together (say in order of appearance) and underlined. This new label is then attached to all symbols in the contractum pattern.

Now if R is an orthogonal TRS, then R^L will be the labeled TRS consisting of all labeled versions of the rewrite rules of R. We note that R^L is again an orthogonal TRS, because an overlap of the labeled rules will yield at once an overlap of the unlabeled rules after omitting the labels.

We will use the notation t^I for a term t in R with labeling I. The labeling I can be perceived as a map from the symbol occurrences of t to the set of labels. So t^I is a term in R^L. Labelings will be denoted by $I, J, \ldots$.

[3]The labelings that we employ were introduced in Klop's Ph.D. thesis.

We can now give a precise definition of the tracing relation between symbols in a rewrite step $t \to s$. Provide t with an *initial* labeling, that is a labeling where each symbol of t gets an atomic label such that different symbol occurrences get a different label. Then we 'lift' the step to the labeled TRS R^L, obtaining a labeled step $t^I \to s^J$.

Now a symbol p^a in t^I traces to all symbols q^α in s^J such that $a \in \alpha$. This tracing relation is then projected down again to the original unlabeled rewrite step $t \to s$. If the symbols p in t, q in s correspond qua position to the labeled symbols p^a and q^α, then p and q are connected by a trace. We also say that p traces forward to q, or that q traces back to p.

We use the notation $p \triangleright q$, and $q \triangleleft p$, as also used on audio- or video-recorders to denote forward or backward winding. If $t \to s$ and p, q are symbols in t, s respectively, then $q\triangleleft$ is the set of symbols in t to which q traces back. Likewise if Q is a set of symbols in s, $Q\triangleleft$ is the union of the origins in t of all $q \in Q$.

EXAMPLE. Let us demonstrate by an example that the present 'algebraic' definition of $\triangleright$ by means of labels indeed yields the same notion as the one described in the more verbose way in Section 3.

Consider again the rule ρ: $F(G(x,y),H) \to R(S(y),y)$, yielding the rewrite step in Figure 2:

$$t \equiv S(F(G(0,1),H)) \to S(R(S(1),1)) \equiv s.$$

Provide the left-hand side t with an initial labeling:

$$S^a(F^b(G^c(0^d,1^e),H^f)).$$

We now apply the following labeled version of ρ:

$$F^b(G^c(x,y),H^f) \to R^{\underline{bcf}}(S^{\underline{bcf}}(y),y)$$

yielding the labeled step

$$t^I \equiv S^a(F^b(G^c(0^d,1^e),H^f)) \to S^a(R^{\underline{bcf}}(S^{\underline{bcf}}(1^e),1^e)) \equiv s^J.$$

Inspection of the labels clearly shows what pairs of symbols (p,q) in the corresponding unlabeled reduction step are in the relation $\triangleright$ and it is easily checked that this is just the relation described in Section 3.

Up to now we have only defined the tracing relation $\triangleright$ for symbols in begin and end of a single reduction step $t \to s$. We would like to do this also for many-step reductions $t \twoheadrightarrow s$, or more explicitly, $t \equiv t_0 \to \cdots \to t_n \equiv s$. This is very simple: we extend $\triangleright$ by transitivity in the obvious way. There is however another way as follows. Give t an initial labeling t^I and lift the reduction $t \twoheadrightarrow s$ to the labeled reduction $t^I \twoheadrightarrow s^J$. Now define as before that a symbol p in t traces to q in s if and only if its label (in t^I) is included in the label of q (in s^J).

So the difference is that in the former definition tracing is defined by repeated initialization of the labels: in each step the labels are 'refreshed' to an initial labeling. Fortunately we can without much effort prove that both ways yield the same. In other words, repeated initialization is superfluous.

This has an important consequence. Given a reduction $t \twoheadrightarrow s$, with s a normal form, let us trace back symbol q in s to its origins in t. Now the question is whether the set of origins depends on the actual reduction from t to s.

Having the second definition of $\lhd$ in mind (direct comparison of an initial labeling of t with the resulting labeling of s) we can now state that the set of origins is independent from the actual intermediate reduction. To see this we first observe that if s is a normal form (in R), then each labeled version s^J is a normal form in R^L. Now orthogonality of R^L implies uniqueness of normal forms. Hence if $t^I \twoheadrightarrow s^J$ on the one hand, and $t^I \twoheadrightarrow s^{J'}$ on the other hand, we have $s^J \equiv s^{J'}$, in particular $J \equiv J'$.

Therefore $q\lhd$ in t is for both reductions the same as it only depends on the initial labeling I and the final labeling J.

Let us finally note that traces can stop in forward direction. That is, if $t \rightarrow s$ and p is a symbol in t, the set $p\rhd$ may be empty. But in backward direction traces do not stop. Everything has an origin—symbols are not created out of the blue.

6 Ordinary descendants and simple labels

With $\rhd$ and $\lhd$ we could give a neat definition of the notions of descendant and ancestor. But in fact this would yield a refinement of the classical, standard descendant relation. Let us compare the two approaches.

Again we use a labeled system to define ordinary descendants. However, now only simple labels are allowed: $\epsilon, a, b, c, \ldots$. Here $a, b, c, \ldots$ are proper labels, they are single letters, and ϵ is the empty label.

Again we decorate rules with labels. Rule ρ gives now rise to all labeled versions

$$F^{\alpha}(G^{\beta}(x,y), H^{\gamma}) \rightarrow S(R(y), y).$$

Now the trace relation $\blacktriangleright$ is defined analogous to the definition of $\rhd$ with this difference: $p \blacktriangleright q$ if and only if p and q have the same proper labeling.

The relation $\blacktriangleright$ characterizes the classical descendant relation. As a matter of fact, usually the notion of descendant is seen as a relation between subterm rather than symbol occurrences. But since subterms and their roots are in 1-1 correspondence, defining the relation on symbols, as we do here, amounts to same thing.

We have $p \blacktriangleright q \Rightarrow p \rhd q$, but not vice versa. For occurrences of variables $x, y, z, \ldots$, and the symbol Ω that will be used later, the two notions are identical.

Note that a redex has no ordinary descendants after its contraction. Further note that if $p \blacktriangleright q$ and p is a redex root, then so is q and vice versa. For $p \rhd q$ the analogous statement does not hold.

7 The prefix property

Consider again the reduction $t_0 \to t_1 \to \cdots \to t_n$ with t_n in normal form. We will prove that $t_n \lhd$ in t_0, that is all the origins of t_n traced back to t_0 constitute a prefix π of t_0. A *prefix* of a term is a set of occurrences that is upward closed. (So in the pictures, an upper 'half' of the triangle if terms are written as trees.) We will denote prefixes of terms by π, etc.

NOTATION. If p, q are symbol occurrences, then $p \leq q$ means that p is above q. We call $\leq$ the prefix ordering.

Now we have the following proposition.

PROPOSITION. The prefix order $\leq$ and the tracing order $\lhd$ commute. That is:

$$s' \leq s \lhd t \Rightarrow \exists t' \; s' \lhd t' \leq t.$$

PROOF. The proposition is easily proved by some casuistics. Consider a step where redex r is contracted. If s' is above s in the redex pattern, we just take any t' in the contractum pattern (if there is). Then $s' \lhd t'$.

If s' is above the redex pattern, than actually it is in the context of r and we take the corresponding t' (at the same position).

From the proposition we have at once that prefixes are preserved in tracing back: if $t \to s$, and π is a prefix of s then, $\pi\lhd$ is a prefix of t.

In the sequel we will need the following property of prefixes. A prefix is *redex-pattern closed*, or for short, has the *rpc* property, if it contains redex patterns only in their entirety, and not 'half' of a redex pattern. In other words, if the prefix contains the root of a redex, it must contain the whole redex pattern.

Now we can strengthen our previous result that prefixes are preserved in tracing back.

PROPOSITION. All prefixes that we find by tracing back the normal form t_n, have the rpc property.

That t_n itself, being its own prefix, has the rpc property, is trivial as it contains no redexes at all. The further proof is again by simple casuistics, and will be omitted here.

Consider a step $t \to s$, and a prefix π of s. Then, as we have seen, $\pi\lhd$ is a prefix of t. Now we may ask whether $\pi \lhd \rhd$ is again π. In general this will not be the case. But if the prefixes have the rpc property it will the case. So we have the following nice situation:

> All terms in the reduction graph of t_0 are partitioned in a 'white' prefix above and a 'dark' remainder. The white prefixes are made up from the origins of all the symbols in t_n. If t, s are reducts of t_0 and $t \twoheadrightarrow s$, their white prefixes $\pi(t)$ and $\pi(s)$ are exactly related by $\rhd$ and $\lhd$.

Recall Figure 1, where the dark and white area's were indicated for the original term t_0.

8 Properties of the prefix

In the sequel we will employ Ω-terms, i.e. terms where the constant Ω may occur. We will use Ω as demarcation of prefixes, by appending them at the cut-points of the prefix. Now we have the following proposition.

PROPOSITION. Let t_0 and t_n be as above, and consider a step $t \to s$ in the reduction graph of t_0. Let $\pi(t)$ and $\pi(s)$ be the prefixes of t and s constituted by the origins of the normal form t_n. Let r be the redex contracted in $t \to s$. Then either $\pi(t) \equiv \pi(s)$, or $\pi(t) \to \pi(s)$ in such a way that the Ω's in $\pi(t)$ have the Ω's in $\pi(s)$ as descendants.

PROOF. The proof is by considering where the redex r is situated in t. If it is in the dark part, its contraction does not affect the prefix at all (the prefix is then in the context of r). If r is in the white prefix of t, then contracting the 'same' redex r (possibly with some of its arguments cut-off) yields the step between $\pi(t)$ and $\pi(s)$.

We have several consequences of the last proposition.

i. There must be a redex (pattern) in the prefix of t_0. By the previous proposition we have $\pi(t_0) \twoheadrightarrow \pi(t_n) \equiv t_n$. Now if there was no redex in $\pi(t_0)$, it was in fact a normal form, and hence $\pi(t_0) \equiv t_n$. But t_n does not contain the symbol Ω, contradiction.

ii. The dark part, under the white prefix, is indeed garbage. Consider the reduction $\pi(t_0) \twoheadrightarrow t_n$. We can refine the Ω's in this reduction to arbitrary terms, while preserving the reduction. So we find that the refined $\pi(t_0)$ reduces to the refined t_n, which is t_n again.

iii. A redex r in the dark part is not needed. Intuitively: r is below some Ω (Ω is like a box containing r); all Ω's are erased in the reduction to t_n; hence also r is erased on the way to t_n and so by definition non-needed.

It remains to be shown that a redex r in the prefix $\pi(t_0)$ is indeed needed. Suppose not, then in some t_k all descendants (in the standard sense) must have vanished, not by contraction, but by erasure. Now consider the root of r, call it p. Then for some symbol q in t_k we must have $p \rhd q$; otherwise p was not in the prefix $\pi(t_0)$. However, the $\rhd$ traces of p follow precisely the $\blacktriangleright$ traces of r, and because the latter all stop at or before t_k by erasure, also the $\rhd$ traces of the top p will stop at or before t_k by erasure. Contradiction.

9 Needed reduction is (hyper) normalizing

We will now show that repeated contraction of needed redexes must terminate, in the normal form, even when between needed contractions we contract some non-needed redexes. To this end we assign a norm $\|t\|$ to each term t in the reduction graph of t_0. First we define the norm $|\alpha|$ of a label: this is just the number of its symbols, counting an underlining as one symbol. Now $||t||$ is the sum of the $|\alpha|$ for every α in the prefix $\pi(t)$. Now (i) for a needed contraction $t \to s$ we have $||t|| < ||s||$, while (ii) for a non-needed contraction we have $||t|| \leq ||s||$. The proper increase in (i) is due to the fact that the labels attached in the contractum pattern are underlined. That in case (ii) no decrease is possible, is due to the fact that the non-needed redexes are in the dark part below the white prefix—so they cannot erase symbols and labels in the white prefix.

From (i) and (ii) we immediately have termination as announced, since the norms are bounded by $||t_n||$, the norm of the normal form.

It is worthwhile to remark that we also have as an immediate corollary that parallel outermost reductions are normalizing. This is seen by first noting that there must be a needed outermost redex, since neededness is preserved upward. If redex r is needed, and r' is a redex containing r as a subterm, then r' is needed. So one of the outermost redexes must be needed. Parallel outermost reduction therefore must be normalizing by the termination theorem just mentioned.

Acknowledgement

We thank Vincent van Oostrom for many helpful discussions about the subject of this note, especially for pointing out the definition and use of the increasing norm.

References

D.T. van Daalen, The language theory of Automath. Ph.D. thesis, Eindhoven University of Technology, 1980. (In: Selected papers on Automath, R.P. Nederpelt, J.H. Geuvers, R.C. de Vrijer, eds. North Holland, 1994.)

A. van Deursen, P. Klint, F. Tip, Origin tracking. Journal of Symbolic Computation, 15, p. 523-545, 1993.

A. van Deursen, P. Klint, F. Tip, Origin tracking and its applications. Ch. 7 in: Language prototyping - an algebraic specification approach, eds.: A. van Deursen, J. Heering, P. Klint, World Scientific Publishing Co., 1996, p. 249-294.

G. Huet, J.-J. Lévy, Call-by-need computations in non-ambiguous linear term rewriting systems. In: Computational Logic, Essays in Honour of Alan Robinson, MIT Press 1991.

(Computations in orthogonal rewriting systems I and II. In J.-L. Lassez and G. Plotkin, eds., p. 395-443, Computational Logic, Essays in Honour of Alan Robinson, MIT Press 1991.)

J.W. Klop, Combinatory reduction sytems. Ph.D. thesis, Utrecht University, 1980. (Mathematical Centre Tracts 127, CWI, Amsterdam 1980.)

J.-J. Lévy, An algebraic interpretation of λ-calculus and a labelled λ-calculus. In: C. Böhm, ed., λ-calculus and Computer Science, Springer LNCS 37, p. 147-165, 1975.

V. van Oostrom, Finite family developments. In: Proc. RTA-97, H. Comon ed., LNCS 1232, Springer, p. 308-322, 1997.

A. Middeldorp, Modular properties of term rewriting systems. Ph.D. thesis, Vrije Universiteit Amsterdam, 1990.

The Missing Links

Mark van den Brand
University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
markvdb@wins.uva.nl

Abstract

Using ASF+SDF as implementation language for the development of components of the ASF+SDF Meta-Environment depends on the availability of ASF compilers or a methodology of transforming ASF specifications into executable programs. By giving an overview of the various activities to obtain a *new* ASF+SDF Meta-Environment it is demonstrated that both the compiler and the methodology is still missing. Developing compilers for ASF revealed the absence of a formal semantics of ASF as well.

1 Unparsing

My first "assignment", as a member of the Programming Research Group five years ago, was to improve the mapping from abstract syntax trees to text, thus improving the unparsing facilities in the ASF+SDF Meta-Environment [10]. Based on the work of Karin Vos [13] a box language and a mapping from this box language to text was specified in ASF+SDF. Furthermore, a mapping from VTP to the box language was programmed in Lelisp. In order to incorporate this new component in the Meta-Environment it was necessary to compile the "BOX2TEXT" specification in one way or another. Pum Walters and Jasper Kamperman just had finished the first version of the ASF2C compiler [9] about the same time and this compiler was used to compile the BOX2TEXT specification. This succeeded, although a huge C program was generated, and the old unparser was replaced by a compiled specification.

Based on this small success a more ambitious project was launched, viz., the development of an unparser generator [6] specified in ASF+SDF as well. The unparser generator was and is used to generate unparsers for the languages C, COBOL, μCRL, PSF, and SEAL. In order to use these unparsers either to replace the default unparser in the ASF+SDF Meta-Environment or in stand-alone environments it was necessary to compile them as well. When compiling the generated C code of the COBOL unparser the limits of the C compiler were almost reached.

So, there was need for improved compilation technology.

2 Transforming Specifications into Programs

Parallel to the development of the unparser generator the first ideas concerning an (incremental) reimplementation of the ASF+SDF Meta-Environment were born. These ideas where concretised on the one hand by replacing the user interface and the text editor part of GSE [11] by Motif and emacs, respectively. On the other hand the specification of the functionality of various components of the Meta-Environment was started. The first track proved to be unsuccessful but lead to the development of the TOOLBUS [1, 2], a software coordination architecture. The second track triggered the development of the AsFix formalism and later on the ATerm formalism to represent the (abstract) syntax trees to be manipulated in the Meta-Environment. Both ATerms and AsFix are formally specified in ASF+SDF.

A number of people, including Wilco Koorn, Pieter Olivier, and Tobias Kuipers, have been working on an algebraic specification of the structure editing facilities of GSE which can be considered as the kick off of the renovation of the Meta-Environment. By the time Tobias Kuipers finished his masters thesis [12], the incremental renovation track was abandoned and the redesign and reimplementation of the new Meta-Environment was in full process. One of the prerequisites for developing the new Meta-Environment is the availability of an ASF compiler that can generate executables that can be connected to the TOOLBUS. Because the development of the AsFix2RNx-compiler (see Section 3) was not yet completed it was decided to transform the specification of the structure editor into a JAVA program. This structure editor used ATerms and AsFix for representing and manipulating syntax trees.

The construction and manipulation of ATerms are supported by a C library as well as a JAVA library. These libraries are very memory efficient because of the sophisticated term sharing mechanism. On top of these ATerms libraries AsFix libraries are developed when writing the JAVA programs for the structure editor and C programs for other components of new Meta-Environment. Several components of the new Meta-Environment were first specified and later on implemented in C, for more details see [3]. Implementing these components based on specifications proved to be rather straightforward, in particular because of the availability of C libraries for ATerms and AsFix.

A case study of transforming an ASF+SDF specification into a TOOLBUS application can be found in [7].

> *However, a structured way of transforming an* ASF+SDF *specification into a C or JAVA program was and is not available.*

3 Compiling ASF+SDF Specifications

The success of a number of industrial ASF+SDF projects, e.g., developing a renovation factory for COBOL [4] and an integrated environment for RISLA [5], depends on the available compiler technology. There have been various projects to develop compilers for ASF+SDF. An overview of the various translation routes is depicted in Figure 1. The compiler should offer a number of extra facilities:

- It should be able to connect the compiler to the TOOLBUS, in order to obtain a better integration of the new developed Meta-Environment and the compiler itself.
- The generated code should be connectable to the TOOLBUS, in order to construct stand-alone environments in a fast way.
- It should be possible to write libraries in conventional, efficient languages for "standard" specifications such as the Integers, etc., to improve the overall performance of the generated code.

In the next paragraphs a number of the arrows of Figure 1 will be discussed. The discussed arrows are compilation steps which have been implemented.

ASF2C The first ASF+SDF compiler is the ASF2C compiler. It offers the possibility to compile specifications. However, incorporating hand written C code which implements standard libraries is not straightforward and it is not clear how the compiler and the generated code can be connected to the TOOLBUS.

ARM Based on these results the ARM technology has been developed [8]. It consists of a restricted number of (rewrite) instructions and a very fast interpreter. This interpreter is written in ANSI-C and thus very portable. The coupling with the TOOLBUS proved to be feasible.

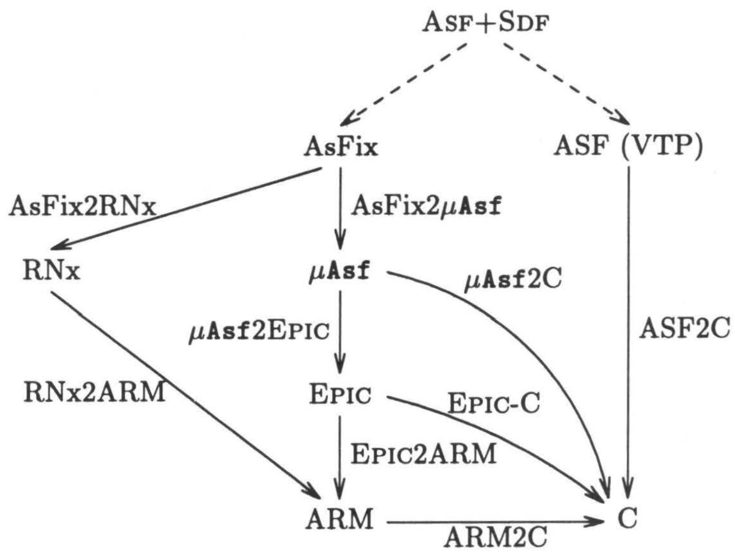


Figure 1: Possible translation routes.

RNx2ARM and EPIC2ARM On top of this ARM technology languages like RNX and later on EPIC are developed [14]. A compiler for translating RNX into ARM has been developed as well as a compiler for EPIC to ARM. These compilers makes it possible to write specifications in a convenient notation.

AsFix2RNX Given the fact that AsFix has become the crucial ingredient in the developments around ASF+SDF and the new Meta-Environment, it was and is logical to use AsFix as a starting point for developing a new compiler. Given RNX as back end an AsFix2RNX compiler was developed. This compiler is entirely specified in ASF+SDF itself and has been bootstrapped.

AsFix2EPIC However, a (serious) bug in the translation of RNX to ARM and the fact that RNX was replaced by EPIC led to the development of an AsFix2EPIC compiler. The most important differences between RNX and EPIC are the absence of both list matching and conditional equations in EPIC. This makes a straightforward translation from AsFix to EPIC impossible.

μAsf The AsFix2EPIC compiler is developed by introducing a family of small languages, **μAsf** languages, each member of this family has specific properties, e.g., list matching, left linearity, conditions. The translation consists of transforming from one language to another (simpler) language. Each transformation step deals with a specific property of ASF, e.g., conditions are removed, rules are made left linear, etc. In order to implement each of these transformation steps it is necessary to know what the exact semantics of each of the ASF properties is. Falling back to the ASF interpreter available in the ASF+SDF Meta-Environment is not a satisfactory way to get insight in the exact semantics of ASF.

> *This translation project reveales that there is a strong need for having a formal semantics of* ASF.

References

[1] J. A. Bergstra and P. Klint. The Discrete Time ToolBus. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996. Full version: Technical Report P9502, Programming Research Group, University of Amsterdam.

[2] J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION '96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88. Springer-Verlag, 1996. Full version: Technical Report P9408, Programming Research Group, University of Amsterdam.

[3] M.G.J. van den Brand, T. Kuipers, L. Moonen, and P.A. Olivier. Implementation of a prototype for the new Meta-Environment. Technical report, University of Amsterdam, Programming Research Group, 1997. This proceedings.

[4] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. Technical Report P9705, University of Amsterdam, Programming Research Group, 1997.

[5] M. G. J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. A. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[6] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.

[7] J.A. Hillebrand. *Experiments in specification re-engineering.* PhD thesis, University of Amsterdam, 1996.

[8] J.F.Th. Kamperman. *Compilation of Term Rewriting Systems.* PhD thesis, University of Amsterdam, 1996.

[9] J.F.Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. In H.A. Wijshoff, editor, *Computing Science in the Netherlands*, pages 193–204, 1993.

[10] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[11] J. W. C. Koorn. *Generating Uniform User-Interfaces for Interactive Programming Environments.* PhD thesis, University of Amsterdam, 1994.

[12] T. Kuipers. Language independent structure editing using the toolbus. Master's thesis, University of Amsterdam, Programming Research Group, 1996.

[13] K.J. Vos. Pretty for an easy touch of beauty. Master's thesis, Programming Research Group, University of Amsterdam, 1990.

[14] H. R. Walters and J. F. Th. Kamperman. Epic: An equational language — Abstract machine and supporting tools. In H. Ganzinger, editor, *Rewriting Techniques and Applications (RTA '96)*, volume 1103 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996. Full version of the language definition: Technical Report CS-R9604, Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

The Leap Year Problem

Arie van Deursen

CWI, Amsterdam

Dedicated to Paul Klint
At the occasion of his 25th CWI-anniversary

De laatste paar maanden hebben Paul en ik ons onder meer bezig gehouden met het beruchte jaar-2000-probleem. Voor een wetenschapper valt hieraan heel wat moois te onderzoeken. Een empirisch onderzoeker kan proberen vast te stellen welk percentage sources bij een grote organisatie — met 100 miljoen regels code en meer — gemiddeld "zoek raakt" in de loop der tijd. Meer theoretische onderzoekers kunnen hun semantische ei kwijt: hoe weet je of een stukje code inderdaad een correcte datumoperatie beschrijft? Praktijkgerichte wetenschappers kunnen hun jongensdroom vervullen en prachtige tools bouwen die door geraffineerde trucs de jaar-2000 fouten automatisch verhelpen. Maatschappelijk bewuste onderzoekers, ten slotte, kunnen met hun opgebouwde kennis het grote publiek voorlichten in televisieshows, of advocaten van raad voorzien in het aankomend juridisch milleniumsteekspel.

Paul voelt zich in elk van deze wetenschappelijke rollen thuis, en heeft zich dan ook intensief beziggehouden met het jaar-2000-probleem. Tijdens ons werk zijn Paul en ik heel wat mooie stukjes programmatuur met al dan niet foutieve datumoperaties tegengekomen. Hieronder een selectie met enkele juweeltjes op het gebied van schrikkeljaarberekeningen.

1 Motivation

Before January 1st, 2000, you will have to check all your software. You will have to see where it relies on dates, and whether your dates use a four-digit ("1997") or a two-digit ("97") representation. Those programs using the two-digit option will very often contain incorrect date manipulations, for example for year comparisons ($97 < 0$?), subtractions ($03 - 97 = 94$?), day-of-the-week determination (Monday January 1st 2000?), etc.[1]

But there is more than these two-digit date errors. The year 2000 is a leap year. Some programs know about this. They either simply check whether the year is divisible by 4, or they do it more thoroughly by also checking that the year is not divisible by 100, unless it is divisible also by 400. In other words, neither 1900 nor 2100 are leap years, but 2000 is.

A significant number of programs, however, incorrectly treats the year 2000 as a non-leap year. This may be caused by the use of two digit dates, i.e., "00" is treated as 1900 rather than 2000. In other cases, the programmer may have simply had the wrong algorithm in his mind. One common error is the assumption that all centuries are non-leap; i.e., the programmer mistakenly forgets the "exception to the exception". The other common error is the assumption that the year 2000 just can't be a leap year.[2]

This "leap year problem" is not a toy problem. An example of the cost that might be involved is the $1,000,000 damage caused by the fact that all control computers of a New Zealand aluminum smelter simultaneously went down as they could not deal with the 366th day of 1996 [8]. Similar crashes are to be expected in the year 2000, at a much larger scale.

In this paper, we will try to obtain a better understanding of the nature of this leap year problem. We will do this by studying a series of real life code fragments.

There are several reasons for studying this leap year problem in considerable detail. First of all, it is a very serious problem,

[1]Not only old applications, but even recent ones like the well-known TOOLBUS appear to suffer from this problem.

[2]The following quote from `comp.sys.sgi` illustrates the confusion: *Ok, my info was incorrect - I was taught in school that years divisible by 400 were leap years except for those divisible by 1000 (confused yet - I sure was!).*

as illustrated by the New Zealand aluminum smelter disaster. Second, current year 2000 tools, even the most recent so-called "second generation" ones, have only limited support (no automatic remediation, for example) for dealing with leap years. Last but not least, the leap year problem is a subproblem of the full millennium challenge. Since the scientific method works best if the problem at hand is as small as possible, this makes the leap year problem more suitable for scientific study than the full year 2000 problem. Moreover, the smaller size may make the problem easier to comprehend, and may therefore help to increase awareness of the — still much underestimated — year 2000 problem.

2 Fragment Overview

In the sections to come, we list leap year fragments in Pascal, C, assembly, and COBOL. They come from various sources, such as year 2000 related literature [6, 7], a request posted on Usenet, and inspections of real life code.[3]

Of particular interest are the COBOL fragments, which come from various sources. A large number of fragments is taken from a 50-page list of date-related code examples as occurring in a set of COBOL programs comprising 10^7 LOC, a list which is discussed in more detail in [3].

Moreover, two 10^5 LOC systems were studied in their entirety. It turned out that in each of these systems, four different procedures for checking leap years were implemented, some correct, some incorrect. These figures are a reason for major worry. Not only do they illustrate a serious lack of code reuse; extrapolating them suggests that a large organisation with say 10^8 LOC (which is not even very large) might maintain 4000 leap year procedures[4]!

The goal of listing the various fragments is to understand what problems would be involved in automatically checking whether an arbitrary piece of code performs a leap year check, whether that check is implemented correctly, and, if it is not, to what extent it is possible to propose automatic corrections for typical incorrect

[3]More fragments are still welcome: you can send an email to `arie@cwi.nl`.

[4]The 4000 figure is too hard to believe. But it would be interesting to see what the true number is.

cases.

3 C Code Fragments

A straightforward way to check whether a given year is a leap year is by means of the following function. It relies on the C % operator to determine the remainder of a division.

Fragment 1

```
int isleap(year) int year; {
return( year%4 == 0 && year%100 != 0 || year%400 == 0 )
}
```

The same can be defined as a macro, here as a function returning the number of days in a year:

Fragment 2

```
#define dysize(A)              \
 ( ( ( (A) % 4 == 0 &&         \
        (A) % 100 != 0 ) || \
      (A) % 400 == 0)          \
    ? 366:365 )
```

The next fragment is essentially the same, but transformed in accordance with the De Morgan laws:

Fragment 3

```
leap_year =
  (year % 4 == 0 &&
  !( year % 100 == 0 &&  !(year % 400 == 0)));
```

The following, from the Linux `cal` application, defines a macro that even takes the calendar reorganization of 1752 into account:

Fragment 4

```
#define leap_year(yr)                          \
 ( (yr) <= 1752 ?  !((yr) % 4) :               \
     ( !((yr) % 4) && ((yr) % 100) ) ||        \
     !((yr) % 400) )
```

Evidently, there are many different ways to combine the various divisions into the correct logical expression. As an example, J. Stockton compiled a list of fragments, all in Pascal, illustrating the various ways of combining the logical operators in leap year computations. The original version is available through `http://www.merlyn.demon.co.uk/programs/leapyear.pas`. It includes versions using only `xor`, `<>`, `=`, `and not`, etc. Moreover, it includes several variations which only perform the modulo operations when strictly necessary, using explicit if-then else logic. An example is:

Fragment 5
```
if Y mod 400 = 0  then Leap := true
else if Y mod 100 = 0 then Leap := false
  else if Y mod 4 = 0 then Leap := true
    else Leap := false ;
```

The C `return` statement introduces further possibilities in conditional expressions:

Fragment 6
```
typedef unsigned char Boolean;
enum _boolEnum { false=0, true=1};

Boolean IsLeapYear(int y) {
  if(y % 400 == 0) return true;
  if(y % 100 == 0) return false;
  if(y % 4 == 0) return true;
  return false;
}
```

4 Correct COBOL fragments

In COBOL, it is not so simple to define a function returning a Boolean result. Therefore, most leap year fragments use explicit IF-THEN statements to guide which remainders are to be computed, followed by either the leap year related code or a move of a flag into a certain variable, as shown in the fragment below. The `PIC` clauses of that fragment are variable declarations. They define a `DATE` record, consisting of three fields, `DAY`, `MONTH` (both two digits wide, indicated by 99), and YEAR, which consumes four digits. This YEAR field is subdivided in two CC century and two non-century YY digits[5].

Fragment 7

```
01 DATE.
   02 DAY         PIC   99.
   02 MONTH       PIC   99.
   02 YEAR        PIC   9999
   02 CCYY REDEFINES YEAR.
      03 CC       PIC   99.
      03 YY       PIC   99.
01 LEAP           PIC   X.

MOVE 'F' TO LEAP.
DIVIDE YEAR BY 4 GIVING Q REMAINDER R-1.
IF R-1 = 0
  IF YY = 0
    DIVIDE YEAR BY 400 GIVING Q REMAINDER R-2
    IF R-2 = 0
      MOVE 'T' TO LEAP
    END-IF.
  ELSE
    MOVE 'T' TO LEAP
  END-IF
END-IF
```

The fragment above does not explicitly divide by 100. Instead, it checks whether the YY part of the year equals zero. The fragment below does essentially the same, yet in a completely different order:

Fragment 8

```
IF YY EQUAL ZERO
  DIVIDE CC BY 4 GIVING Q REMAINDER R
ELSE
  DIVIDE YY BY 4 GIVING Q REMAINDER R
END-IF
```

[5]Observe that COBOL requires an explicit declaration of the number of digits to be used for a certain field. This is a major cause of the year 2000 problem.

```
IF R EQUAL ZERO
  MOVE 29 TO MONTH-DAYS(2)
ELSE
  MOVE 28 TO MONTH-DAYS(2)
END-IF
```

The above examples contain the full check, including the division by 400. From the year 2000 perspective, procedures that just divide by 4 are generally considered correct, since their first incorrect exposure is in the year 2100:

Fragment 9

```
DIVIDE YEAR BY 4 GIVING Q REMAINDER R.
IF R = 0
  [ leap year code ]
END-IF
```

Unlike C or Pascal, COBOL does not have a modulo operator. One can either use the explicit DIVIDE BY clause, or use a compute statement as follows:

Fragment 10

```
COMPUTE Q = YEAR / 4.
COMPUTE R = YEAR - (Q * 4).
```

The following example uses a special way of computing the remainder: it performs a division by four, and stores the result in a variable which only can hold two digits behind the decimal point (the `PIC V9(02)`): if this result behind the decimal point equals zero, division by 4 was possible:

Fragment 11

```
  05 TMP   PIC V9(02).

COMPUTE TMP = YY / 4.
IF TMP EQUAL ZERO
  MOVE 29 TO DD(2)
ELSE
  MOVE 28 TO DD(2)
END-IF.
```

The following example not only checks whether the current year is leap, but also whether the next year is a leap year:

Fragment 12

```
DIVIDE YEAR BY 4 GIVING Q REMAINDER R.
IF R EQUAL 0
  MOVE 'T' TO LEAP-THIS-YEAR
ELSE
  IF R EQUAL 1
    MOVE 'T' TO LEAP-LAST-YEAR
  END-IF
END-IF
```

The next example uses GOTOs to get the appropriate leap year logic. Tools for automatically recognizing such fragments may take advantage of algorithms for automatically removing such gotos.

Fragment 13

```
  MOVE 28 TO DM(2).
  DIVIDE CCYY BY 4 GIVING Q REMAINDER R-1.
  IF R-1 NOT EQUAL ZERO
    GO TO L-100
  END-IF.
  DIVIDE CCYY BY 400 GIVING Q REMAINDER R-1.
  DIVIDE CCYY BY 100 GIVING Q REMAINDER R-2.
  IF R-1 NOT EQUAL ZERO AND R-2 EQUAL ZERO
    GO TO L-100
  END-IF.
  MOVE 29 TO DM(2).
L-100.
  [ ... ]
```

The fragment below comes from an assembly routine for computing the days of the week based on an "eternal" calendar which runs from 1901 until 1999. The program itself will be incorrect from 2000 onwards (it will consider 02 to be 1902 rather 2002). The leap year computation fragment, however, does not take century years into account, and therefore happens to behave correctly from 1901 until 2099.

Fragment 14

```
*       CSECT
*       XM095   ETERNAL CALENDAR 1901 - 1999
*       INPUT PARAMETERS
*             P1: DATE  TT,MM,JJ   6 DIGITS
* This calendar assumes that
*       1. january 1901 was a tuesday

        LH      R1,YYB           -- YEAR
        BCTR    R1,0             -- MINUS 1
        MH      R1,=H'365'       -- GET DAYS
        ST      R1,YTT           -- SAVE IT

        XR      R8,R8            -- CLEAR
        LH      R9,YYB           -- YEAR
        LA      R1,4             -- DIVISOR
        DR      R8,R1            -- YEAR / 4
        LTR     R8,R8
        BNZ     D1               -- NOT LEAP YEAR, JUMP
        CLC     MMB,=H'2'
        BH      D1               -- MONTH > FEBR. OK
        BCTR    R9,0             -- SUBTRACT 1
D1      EQU     *
        ST      R9,STT           -- SAVE CORRECTION
```

Sneed [7] describes how this program can be re-engineered automatically into COBOL, in order to improve maintainability. The result is the following fragment:

Fragment 15

```
        COMPUTE YYT = ( YYB - 1 ) * 365
        MOVE YYB TO R9 R8
        COMPUTE R9 = R9 / 4
        COMPUTE R8 = R8 - 4 * R9
        IF R8 = 0 AND NBM <= 2
          SUBTRACT 1 FROM R9
        END-IF
        MOVE R9 TO STT
```

This fragment is easier to read than the assembly equivalent, yet still very similar. Observe that the two compute statements are akin to the remainder test of Fragment 10, yet not the same: some form of dataflow analysis will be needed to recognize that these two fragments are indeed the same.

5 Incorrect COBOL fragments

The fragments shown so far use a wide variety of language constructs, yet they all provide a correct implementation of a leap year check. The fragments still to come, however, are incorrect. Many of the leap year procedures found in real life code contain the error to make all centuries non-leap years:

Fragment 16

```
DIVIDE YEAR BY 4 GIVING Q REMAINDER R-1.
DIVIDE YEAR BY 100 GIVING Q REMAINDER R-2.
IF R-1 = 0 AND R-2 NOT = 0
THEN
  [ leap year code ]
END-IF
```

Essentially the same, yet in a completely different order, is the following:

Fragment 17

```
IF YY EQUAL ZERO
  MOVE 28 TO MONTH-DAYS(2)
ELSE
  DIVIDE YEAR BY 4 GIVING Q REMAINDER R
  IF R EQUAL ZERO
    MOVE 29 TO MONTH-DAYS(2)
  ELSE
    MOVE 28 TO MONTH-DAYS(2)
  END-IF
END-IF
```

The next example splits the division and the test across different paragraphs, using moreover an 88 predicate field (which is only true if the value of TMP is equal to 0). Observe that the result is that 2000 is not a leap year (incorrect).

Fragment 18

```
01 TMP              PIC 99.
   88 LEAP-YEAR     VALUE ZERO.

   PERFORM CHECK-LEAP
   IF LEAP-YEAR
     ADD 186 TO DAY-COUNT
   ELSE ADD 185 TO DAY-COUNT
   END-IF.

CHECK-LEAP.
   IF YY = ZERO
     MOVE 1 TO TMP
   ELSE
     DIVIDE YEAR BY 4
       GIVING Q REMAINDER TMP
   END-IF.
```

Some algorithms explicitly check that the year in question is not the year 2000:

Fragment 19

```
DIVIDE YEAR BY 4 GIVING Q REMAINDER R.
IF R = 0 AND YEAR NOT = 2000
THEN
  [ leap year code ]
END-IF
```

The utlimate short term solution is the "algorithm" below:

Fragment 20

```
IF YY = 92 OR 96
  MOVE 29 TO MD(2)
END-IF
```

6 Conclusions

The fragments show that the leap year problem is indeed a problem. Not all leap year procedures are incorrect, but a significant fraction of them is, perhaps up to 20%. Moreover, we have seen that many companies do not have a single library function for determining leap years, but rather have several procedures for each application. In other words, many leap year checking procedures exist, some of them are correct, some are incorrect.

The list of fragments raises the question to what extent automated tool support for recognizing leap years is conceivable. A candidate route is through the use of *program plan recognition* technology, as explored by [2]. This approach advocates capturing typical computations, such as determining a remainder, checking for zero, checking for divisibility (by 4), checking leap years, etc., in a library of so-called *plans*. A *plan recognizer* can efficiently check whether a given program contains code written according to one of these plans.

From the list of fragments discussed here, an initial set of plans can be composed. The better the plan recognition technology, the fewer plans will be needed to represent all fragments. For example, if the recognizer is aware of the laws of De Morgan, it will not need separate plans for both Fragments 2 and 3.

The list of leap year fragments helps to distill what challenges should be addressed by such tools:

- Reduction of syntactic variety, for example by selecting certain "canonical" constructs only.
- Reduction of dataflow variety, for example by eliminating transitive dependencies.
- Reduction of control flow variety, for example by goto elimination.
- Searching for patterns combining syntactic, dataflow, and control flow constraints;
- Matching modulo a set of equations, for example expressing associativity and commutativity of logical AND.

References

[1] A. van Deursen and P. Klint. Het jaar-2000-probleem. *Computerrecht*, December 1997.

[2] A. van Deursen, S. Woods, and A. Quilici. Program plan recognition for year 2000 tools. In *Proceedings 4th Working Conference on Reverse Engineering; WCRE'97*. IEEE Computer Society, 1997.

[3] A. van Deursen, P. Klint, and A. Sellink. Validating year 2000 compliance. Technical Report SEN-R9712, CWI, 1997. Also appeared as Chapter 2 of the Resolver Release I project deliverable.

[4] The millennium bug: Please panic early. *The Economist*, pages 23–27, October 4th – 10th, 1997.

[5] C. Jones. The global economic impact of the year 2000 software problem. URL: `http://www.spr.com/`, 1997. Software Productivity Research, Inc.

[6] B. Ragland. *The Year 2000 Problem Solver: A Five Step Disaster Prevention Plan*. McGraw-Hill, 1997.

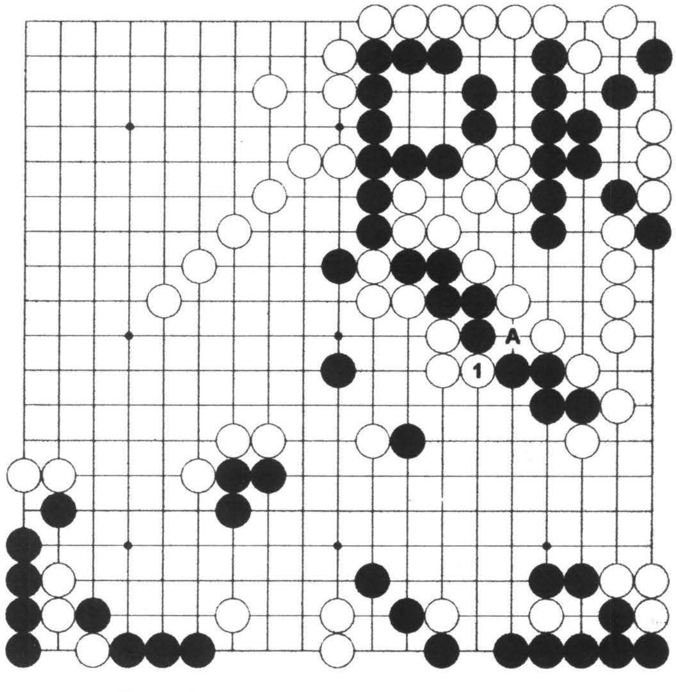
[7] H. M. Sneed. Encapsulating legacy software for use in client/server systems. In *Proceedings Third Working Conference on Reverse Engineering; WCRE'96*. IEEE Computer Society, 1996.

[8] J. Towler. Leap-year software bug gives "million-dollar glitch". *The Risks Digest*, 18(74), 1997. URL: `http://catless.ncl.ac.uk/Risks/18.74.html#subj5`.

Een Paul-Klint-Go-probleem

Niek van Diepen, Katholieke Universiteit Nijmegen

Een jaar of vijfentwintig geleden was Paul Klint een verdienstelijk go-speler. Hij heeft klasse 25 bereikt, internationaal beter bekend als 3e kyu. Voor de niet-ingewijden: indertijd moet hij zo ongeveer de Nederlandse top-100 hebben gehaald. Hij zal niet de enige go-speler op het (toen nog) Mathematisch Centrum zijn geweest. Zo is Andries Brouwer gepromoveerd met een go-stelling als 'tiende' stelling.



figuur 1: zwart aan zet

In bovenstaande figuur is de groep rechtsboven zwaar in de problemen. Er is slechts ruimte voor een oog. Wit heeft zojuist 1 gespeeld, en dreigt daarmee vijf zwarte stenen te slaan. Daarmee zijn alle ontsnappingsroutes naar rechtsonder afgesneden. Een ontsnapping naar linksonder is ook niet mogelijk: de witte stellingen rondom zijn zeer sterk. Maar zwart heeft nog een kans.

Kan zwart verbinden op A? De oplossing vindt u elders in dit boek.

Oplossing Paul-Klint-Go-probleem

Het begin van de zwarte ontsnappingstruc ziet u in diagram 1. Na zwart 2 moet wit wel de ladder verder spelen, want B en C, resp. D en E zijn miai: speelt zwart de ene, dan moet wit de andere spelen om zwarts tweede oog te voorkomen. Dus zwart kan B en D met voorhand spelen, en daarmee de knipstenen altijd vangen.

Merk op dat 3 op 4, 5 op 6, en 7 op 8 niet werken. Na 10 kan wit ook de ladder niet op 12 voortzetten, zodat 11 gedwongen is. De ladder verandert van richting! Ook wit 15 is gedwongen, al lijkt na 24 de ladder uit: wit staat immers atari. Maar hij kan dekken en zelf gelijk atari geven. Als zwart dan verbindt moet wit een steentje offeren op 27, voor hij op 29 weer atari geeft.

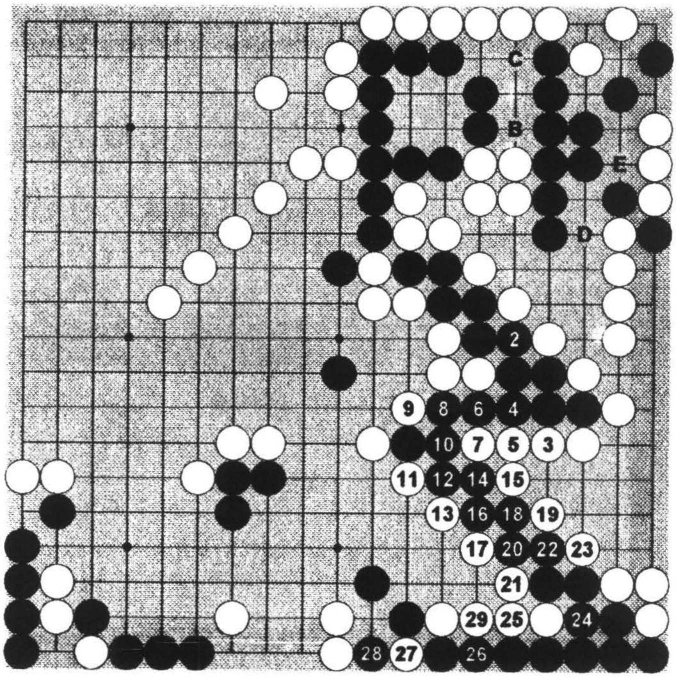


diagram 1: 2-30 (30 dekt op 27)

De ladder maakt nog een draai in diagram 2. Zwart kan na 55 uiteraard niet op 57 slaan, maar hij kan wel op 56 slaan en verbinden. Na 60 staat wit weer atari.

Maar in diagram 3 blijkt dat wit net steeds op tijd kan dekken en zelf atari geven. 75 kan niet op 76 gespeeld worden vanwege de zwarte steen in het centrum. Na 88 is de witte droom uit: de zwarte groep leeft.

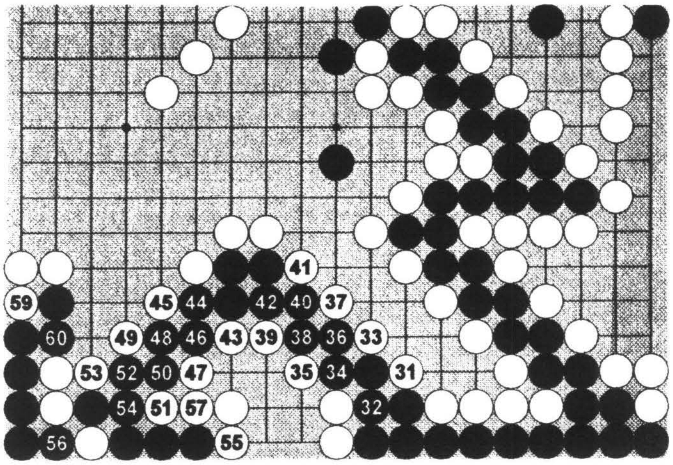


diagram 2: 31-60 (58 dekt rechts naast 56)

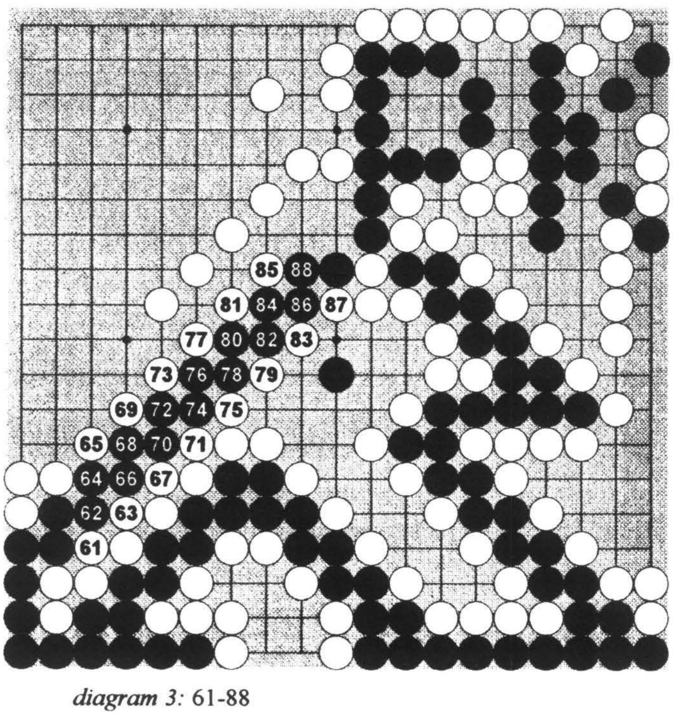


diagram 3: 61-88

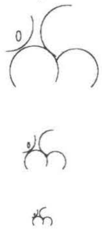
- $'O\dot{g}m \cdots\ 'O\dot{g}m'$ -

I go to him with anxiety, self-doubt, try to explain the problem and in the end all i hear is a pause before he goes $'O\dot{g}m \cdots 'O\dot{g}m'$. Wonder what that was supposed to mean? It could mean anything from 'you don't get it do you?' to 'I don't know either.. in fact i have the same problem.. glad to hear you are in the doo too'. So then why $'O\dot{g}m \cdots\ 'O\dot{g}m'$? What is he thinking? Guess it could be anything in the gamut between the two. There is more than what meets the eye---his other self is being a leader. The researcher-leader-combi flares. Can eh? the leader loathes the researcher and the researcher the leader. Logic dictates 'come out'; 'say it, say you don't get it do you'; 'say i am in doo too'. The researcher-leader-combi flares. I believe the leader leads.. and he utters $'O\dot{g}m \cdots\ 'O\dot{g}m'$. Anyway, leader reckons: its too late to say that now.. to say 'you dint get it do you?' And the reckoning goes on to: if you do without, it ain't in it and i ain't gonna 'its okay'. You see? $'O\dot{g}m \cdots\ 'O\dot{g}m'$. O$\dot{m}$ tat sat---calling Supreme. 'Jeez, what have you done now?' 'Oh God' and other God___ words come to mind.
$'O\dot{g}m \cdots\ 'O\dot{g}m'$

The pattern of the context is as predictable as the little black book. The little book that pops out of the hip pocket when in Holland you ask friends if they would like to go along to a movie with you. The next thing you know, you have a date to go to the movies, in 5 minutes, in about 5 weeks. Caveat emptor: you are Dutch and engulfed in it and cant see the pattern---not the same. Now, you see what the pattern is?
$'O\dot{g}m \cdots\ 'O\dot{g}m'$

Samples are so cute. I want to imitate. I know I can do better and his was a hack :) Times have changed. The great columns that uphold the legacy code are to go. The code of the legacy dictates evolution, but sometimes its the revolution that delivers the colossal structures without columns. 'nice', he says, 'isn't it?'
$'O\dot{g}m \cdots\ 'O\dot{g}m'$

I never restructure right; never ever in retrospect; never never in hindsight. You structure so it evolves; we are structured so we evolve. The sound of creation; the sound within; the sound in-spite;



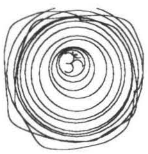
the omkara.
'Oġm··· 'Oġm'

Its a private language; any interpretation of it only destroys the definition!
'Oġm··· 'Oġm' :)

You see I could not simply spell'em, without their other dimensions, 'hmm 'hmm'!!

So Paul.. heres to your repertoire...

'Oġm··· 'Oġm'



--'t din

The Origins of Error Messages

T. B. Dinesh

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands
dinesh@cwi.nl

Frank Tip

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
USA
tip@watson.ibm.com

Abstract

Paul Klint has been intrigued for a long time by the apparently simple problem of producing error-messages with line-numbers that indicate where a program error occurred. His work in this area ranges from an approach for concisely encoding line-numbers in abstract machine instructions [30] to a method for automatically tracking positional information in term rewriting systems [14]. During this time, he supervised several Ph.D. students who worked in this area, and he contributed both directly and indirectly to a long chain of publications concerned with this problem in one way or another [13, 17, 36, 11, 15, 24, 10, 9, 23, 38, 37, 16, 12, 19, 40, 20, 21]. The present paper is the latest addition to this chain.

We present an approach where the location associated with an error message e is defined as a *slice* P_e of the program P being type checked. We show that this approach yields highly accurate positional information: P_e is a program that contains precisely those program constructs in P that caused error e. Semantically, we have the interesting property that type checking P_e is guaranteed to produce the same error e. Our approach is completely language-independent, and has been implemented for a significant subset of Pascal. We report on several experiments with the implementation, and compare our results to previous approaches for localizing errors based on variations of origin tracking.

1 Introduction

Type checkers are tools for determining the constructs in a program that do not conform to a language's type system. Type checkers are usually incorporated in interactive programming environments where they provide programmers with rapid feedback on the nature and locations of type errors. The effectiveness of a type checker crucially depends on two factors:

- The "informativeness" of the type errors reported by the tool.
- The quality of the positional information provided for type errors.

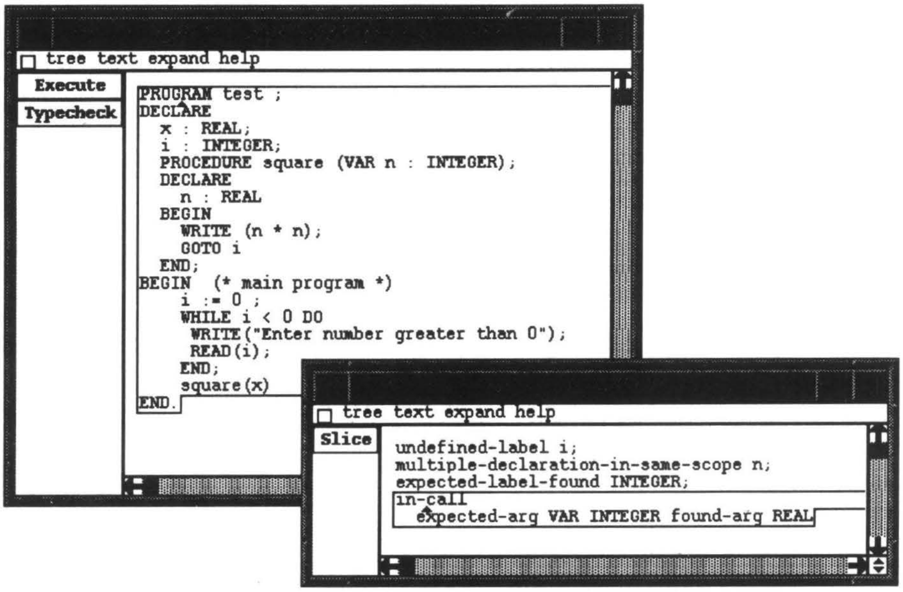


Figure 1: The CLaX environment. The top window is a program editor with two buttons attached to it for invoking a type checker and an interpreter, respectively. The bottom window shows a list of four type errors reported by the type checker. After selecting an error message in the bottom window, the **Slice** button can be pressed to obtain the associated slice.

We believe that the second factor is especially important. For example, consider an assignment statement $x = y$ where x and y are of two incompatible types. What is the source of the error? Specifically, one might ask whether the assignment construct itself is "causing" the error, or if the declarations of x and y, where the incompatible types are introduced, constitute the real "source" of the error. As another example, consider a situation where a label is defined twice inside some procedure. Ideally, the location of this error would comprise *both* occurrences of the label.

We pursue a semantically well-founded approach to answer the question of what the location of a type error should be. In this approach, the behavior of a type checker is algebraically specified by way of a set of conditional equations [2], which are interpreted as a conditional term rewriting system (CTRS) [32]. These rewriting rules express the type checking process by transforming a program's abstract syntax tree (AST) into a list of error messages. We use dynamic dependence tracking [24, 37] to determine a *slice* of the original program as the positional information associated with an error message. This approach has the following advantages:

- The tracking of positional information is completely language-independent and automated; no information needs to be maintained at the specification level.

- Unlike previous approaches [15, 12], no constraints are imposed on the style in which the type checker specification is written. Error locations are always available, regardless of the specification style being used.

- The approach is semantically well-founded. If type checking a program P yields an error message e, then the location P_e associated e is a projection of P that, when type checked, will produce the same error message e. For details about semantic properties of slices, the reader is referred to [24, 37].

Although positional information is always available for any error message, the *accuracy*[1] of these locations depends on the degree to which the specified type checker's behavior is deterministic. This issue will be explored in Section 4.

We have implemented a prototype type checking system using the ASF+SDF Meta-environment [31, 12], a programming environment generator that implements algebraic specifications by way of term rewriting. Dependence tracking was previously implemented in the ASF+SDF system's term rewriting engine for the purpose of supporting dynamic slicing in generated debugging environments [38]. Figure 1 shows a snapshot of a type checking environment for the language CLaX, a Pascal-like language. The most interesting features of CLaX are: nested scopes, overloaded operators, arrays, goto statements, and procedures with reference and value parameters. The top window of Figure 1 is a program editor, which has two buttons labeled '**TypeCheck**' and '**Execute**' attached to it, for invoking the type checker and the interpreter, respectively. The bottom window shows a list of four error messages reported by the type checker for this program.

1. The first error, `undefined-label i`, indicates that the program contains a reference to a label `i`, but there is no statement with label `i` in the same scope.

2. The second error message, `multiple- declaration-in-same-scope n`, points out that an identifier `n` is declared more than once in the same scope.

3. The third error, `expected-label-found INTEGER`, indicates that the program contains an identifier that has been declared as an integer, but which is used as a label.

4. The fourth error, `in-call expected-arg VAR INTEGER found-arg REAL`, points out a type error in a procedure call. In particular, that a procedure is called with a argument type REAL when it was expecting an argument of type INTEGER.

Note that these error messages do not provide any information as to *where* the type violations occurred in the program text.

[1] Accuracy indicates the quality of the slice obtained. Generally, "small" slices, which contain few program constructs, are desirable because they convey the most insightful information.

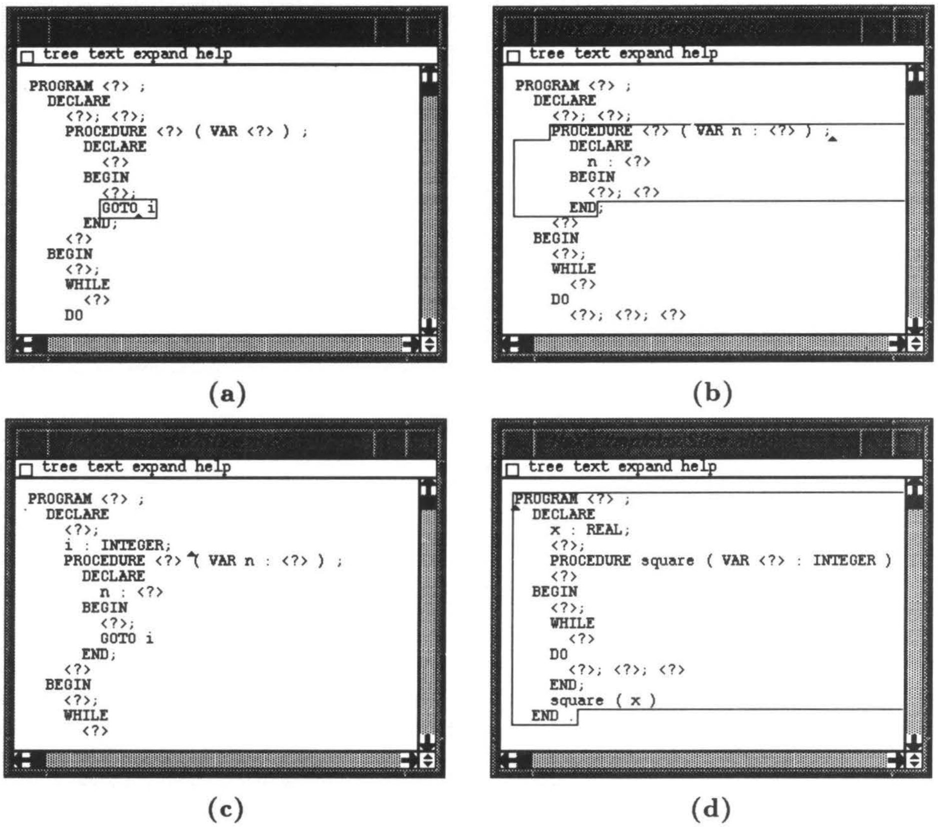


Figure 2: Slices reported by the CLaX environment for each of the type errors of Figure 1.

However, positional information may be obtained by selecting an error message and clicking on the '**Slice**' button. In Figure 2(a)–(d), the slices obtained for each of the four error messages of Figure 1 are shown[2]. Each slice is a view of the program's source indicating the program parts that contribute to the selected error. Placeholders, indicated by '<?>' in the figure, indicate program components that do not contribute to the error under consideration. The semantics of "not contributing towards a certain error message" may be characterized informally as follows: If a placeholder in the slice with respect to an error *e* is replaced with a program component of the same kind[3], type checking the resulting program is guaranteed to produce the same error *e*.

1. Figure 2(a) shows the slice for the `undefined- label` error. Clearly, the `GOTO i` statement is the source of the error, because there is no statement with label `i`.

2. Figure 2(b) shows the slice obtained for the `multiple-declaration-in-same-scope` error. The problem here is that `n` is a parameter as well as a local variable of procedure `square`. Note that both declarations of `n` occur in the slice.

3. Figure 2(c) shows the slice obtained for the `expected-label-found INTEGER` error. Note that, in addition to the `GOTO i` statement and the declaration of `i` as an `INTEGER`, all declarations in the inner scope appear in the slice. Informally, this is the case because replacing any of these declarations by declarations for variable `i` may affect the outcome of the type checking process, in the sense that the `expected-label-found INTEGER` error would no longer occur.

4. Figure 2(d) shows the slice obtained for the `in-call expected-arg VAR INTEGER found-arg REAL` error. Observe that the slice precisely indicates the program components responsible for this problem: (i) the call site `square(x)` that gave rise to the problem, (ii) the type, `INTEGER`, of `square`'s formal parameter (note that the name of this parameter is irrelevant), and (iii) the declaration of variable `x` as a `REAL`.

The reader may observe at this point that, in addition to the program constructs responsible for a type error, a slice generally also contains certain structural information such as `BEGIN` and `END` keywords and declaration and statement list separators that are not directly related to an error. The occurrence of this structural information is due to the way slices are computed. If desired, displaying this information could easily be suppressed to a large extent. For example, removal of all `BEGIN`, `END`, and `DECLARE` keywords and list separators from the computed slices would reduce the amount of "noise" considerably. In certain cases, slices may contain `IF` or `WHILE` statements whose condition and

[2] An alternative way for displaying slices would be to highlight the corresponding text areas in the program editor of Figure 1.

[3] Although all placeholders are displayed as '<?>', placeholders are typed. In order to preserve syntactic validity of the program, an expression placeholder may only be replaced by another expression, an unlabeled-statement placeholder may only be replaced by another unlabeled-statement, etc.

[Eq1]	`tc(begin` *Decls Stats* `end) = dist(`*Stats*`, tenv(`*Decls*`))`
[Eq2]	`dist(`$Stat_1$`;`$Stat_2$`,` *Tenv*`) = dist(`$Stat_1$`,` *Tenv*`); dist(`$Stat_2$`,` *Tenv*`)`
[Eq3]	`dist(`*Id* `:=` *Exp*`,` *Tenv*`) = dist(`*Id*`,` *Tenv*`) := dist(`*Exp*`,` *Tenv*`)`
[Eq4]	`dist(`Exp_1 `+` Exp_2`,` *Tenv*`) = dist(`Exp_1`,` *Tenv*`) + dist(`Exp_2`,` *Tenv*`)`
[Eq5]	`dist(`*Id*`,` *Tenv*`) = type-of(`*Id*`,` *Tenv*`)`
[Eq6]	`type-of(`*Id*`, tenv(`T_1^*`;` *Id* `:` *Type*`;` T_2^*`)) =` *Type*
[Eq7]	`natural + natural = natural`
[Eq8]	`natural := natural = "correct"`

Figure 3: Static semantics specification for determining the validity of assignments.

body are omitted from the slice (see, e.g., Figure 2(d)). Such constructs can also be removed from the slice without affecting the semantic content. We consider slice postprocessing to be primarily a user-interface issue, which is outside the scope of this paper.

The remainder of the paper is organized as follows. Section 2 presents our approach for specifying type checkers. In Section 3, the use of term rewriting for executing specifications is discussed. In addition, dependence tracking, the mechanism for computing slices is presented. Section 4 is concerned with the effect of determinism in the specification on slice accuracy. In Section 5, related work is discussed. In particular, the slice notion introduced in the present paper is compared with the traditional notion of a program slice. Conclusions and possible directions for future work are stated in Section 6.

2 Specification of Static Semantics and Type Checking

A *static semantics* specification only determines the validity of a program and is not concerned with pragmatic issues such as the source location where a violation of the static semantics occurred, or even what program construct caused the violation. A *type checker* specification typically uses the static semantics specification as a guideline, and specifies the presentation and source location of type errors in invalid programs. Adding such reporting information to a static semantics specification is a cumbersome and error-prone task, because keeping track of positional information can be nontrivial, especially if multiple program fragments together constitute a type error.

In [17], we introduced an abstract interpretation style for writing static semantics specifications. In a nutshell, this style advocates the following:

- reducing program constructs to their *type*,
- evaluating type expressions at an abstract level, and
- only specifying the type-correct cases.

```
[Er1]   msgs(Stat1;Stat2) = msgs(Stat1);msgs(Stat2)

[Er2]   msgs("correct") = "No errors"

[Er3]   Msg*; "No errors"; Msg*' = Msg*; Msg*'

[Er4]   msgs(T1 := T2) = msgs(T2)
        when simpletype(T2) ≠ true

[Er5]   msgs(T1 := T2) = "Incompatible types in assignment."
        when simpletype(T2) = true

[Er6]   msgs(T1 + T2) = "Operands of + should have the same type."

[Er7]   simpletype(natural) = true

[Er8]   simpletype(string) = true
```

Figure 4: Postprocessing to obtain human-readable messages.

Operationally, the static semantics specification describes a transformation of a program to a set of type-expressions for program constructs that are type-incorrect.

Figure 3 shows a tiny static semantics specification for determining the validity of assignment statements in straight-line flow programs. The reader should be aware that this specification only serves to illustrate the general style of specifying a static semantics and is incomplete; for example, it does not verify if variables are declared more than once. Equation **[Eq1]** defines a top-level function `tc` for checking a program. Informally, **[Eq1]** states that checking a program involves (i) creating an initial type-environment that contains variable-type pairs, and (ii) distributing the type-environment over the program's statements, using an auxiliary function `dist`. For the simple example we study here, the type-environment consists of the declaration section of the program, to which the constructor function `tenv` is applied. Equation **[Eq2]** expresses the distribution of type-environments over lists of statements, and **[Eq3]** and **[Eq4]** the distribution over assignment operators and '+' operators, respectively. **[Eq5]** states how an identifier is reduced to its type, using an auxiliary function `type-of`, which is defined in **[Eq6]**. Note that the variables T_1^* and T_2^* in **[Eq6]** match any sublist of (zero or more) declarations in a declaration section. Equation **[Eq7]** expresses the abstract evaluation of additions, and **[Eq8]** states that the assignment of a natural expression to a natural variable is valid.

As an example, consider checking the following program block:

```
tc(begin x : natural; y : string;
   x := x + x; x := y + x end)
```

Application of **[Eq1]** results in:

```
dist(x := x + x; x := y + x,
     tenv(x : natural; y : string))
```

Application of **[Eq2]** yields:

```
dist(x := x + x,
     tenv(x : natural; y : string));
dist(x := y + x,
     tenv(x : natural; y : string))
```

At this point, [Eq3] can be applied to both components, producing:

```
dist(x, tenv(x : natural; y : string))
:= dist(x + x,
        tenv(x : natural; y : string));
dist(x, tenv(x : natural; y : string))
:= dist(y + x,
        tenv(x : natural; y : string))
```

The left-hand sides of both assignments can be reduced to their types using [Eq5] and [Eq6], resulting in:

```
natural :=
  dist(x + x,
       tenv(x : natural; y : string));
natural :=
  dist(y + x,
       tenv(x : natural; y : string))
```

Using [Eq4] and [Eq5], the right-hand sides of the assignments can be simplified:

```
natural := natural + natural;
natural := string + natural
```

Using equation [Eq7], the first assignment can be simplified:

```
natural := natural;
natural := string + natural
```

Finally, application of [Eq8] yields the final result:

```
"correct";
natural := string + natural
```

The fact that this term contains a subterm that cannot be reduced to "correct" indicates that the program is not type-correct. Note that the non-"correct" subterm already gives a rough indication of the nature of the type violation.

Figure 4 shows a set of equations that define a function msgs that transforms the cryptic messages produced by the specification of Figure 3 into human-readable form. The equations of Figure 4 assume that the term to which they are applied is fully normalized w.r.t. type checking equations of Figure 3.

Equation [Er1] distributes function msgs over all statements in a block. [Er2] transforms the constant correct, which was derived from a type-correct program construct, into a message "No errors". Since we are not interested in generating messages for correct statements, equation [Er3] eliminates "No

errors" from lists of messages. Equations **[Er4]** and **[Er5]** perform the post-processing of expressions that are derived from incorrect assignment statements. Note that these equations are *conditional*: they are only applicable if a certain condition holds. (Here, the condition verifies if the right-hand side of the expression is a simple type, using auxiliary equations **[Er7]** and **[Er8]**.) **[Er4]** postprocesses assignment statements whose right-hand side consists of an irreducible expression; whereas **[Er5]** postprocesses assignments whose left-hand side and right-hand side are incompatible. Equation **[Er6]** postprocesses '+' expressions with incompatible arguments. The reader should observe that the specification of Figure 4 only serves to illustrate the general technique and that it is incomplete; For example, it does not handle nested expressions.

As an example, we will postprocess the term "**correct**"; **natural := string + natural** by applying the equations of Figure 4 to the term:

```
msgs("correct";
     natural := string + natural)
```

Applying **[Er1]** produces:

```
msgs("correct");
msgs(natural := string + natural)
```

Using equation **[Er2]**, we obtain:

```
"No errors";
msgs(natural := string + natural)
```

By applying **[Er3]**, the "**No errors**" message is eliminated:

```
msgs(natural := string + natural)
```

Since the right-hand side of the assignment is not of a simple type (we cannot derive the constant **true** from the term **simpletype(string + natural)**, conditional equation **[Er4]** can be applied, producing:

```
msgs(string + natural)
```

Application of **[Er6]** yields the human readable error message:

```
"Operands of + should have the same type."
```

The CLaX type checker specification that has been used to generate the snapshots of Figures 1 and 2 follows the same basic principles that have been presented in this section. Language features such as gotos, nested scopes, and arrays introduce some additional complexity, but we experienced no fundamental problems. An annotated listing of the CLaX specification will appear in a technical report in the near future [18]. A previous version of the CLaX specification may be found in [17].

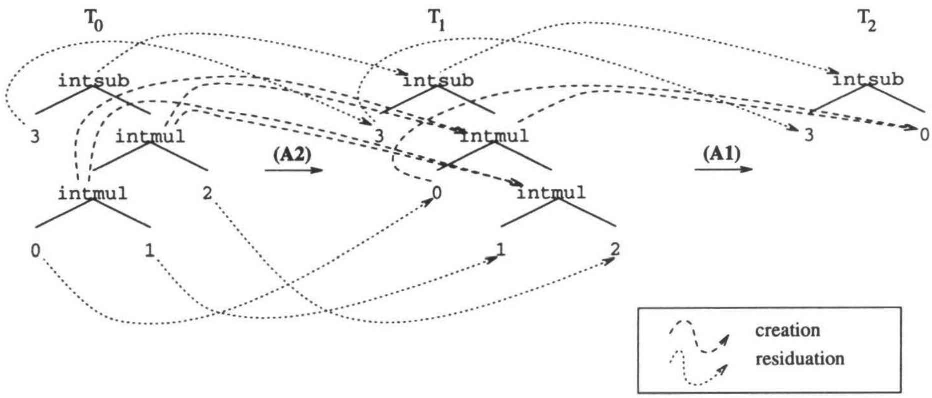


Figure 5: Example of creation and residuation relations.

3 Term Rewriting and Dependence Tracking

In the previous section, specifications were "executed" by repeatedly applying equations to terms—a mechanism that is usually referred to as *term rewriting*. Both theoretical properties of term rewriting systems [32] such as termination behavior, and efficient implementations of rewriting systems [28, 29] have been studied extensively.

Term rewriting [32] can be viewed as a cyclic process where each cycle begins by determining a subterm t and a rule $l = r$ such that t and l *match*. This is the case if a substitution σ can be found that maps every variable X in l to a term $\sigma(X)$ such that $t \equiv \sigma(l)$ (σ distributes over function symbols). For rewrite rules without conditions, the cycle is completed by replacing t by the instantiated right-hand side $\sigma(r)$. A term for which no rule is applicable to any of its subterms is called a *normal form*; the process of rewriting a term to its normal form (if it exists) is referred to as *normalizing*. A conditional rewrite rule [3] (such as **[Er4]** and **[Er5]** in Figure 3) is only applicable if all its conditions succeed; this is determined by instantiating and normalizing the left-hand side and the right-hand side of each condition. Positive (equality) conditions (of the form $t_1 = t_2$) succeed iff the resulting normal forms are syntactically equal, negative (inequality) conditions ($t_1 \neq t_2$) succeed if they are syntactically different.

Thus far, we have described the process of specifying a type checker, and the execution of such specifications by way of term rewriting. In order to obtain positional information, we use a technique called *dependence tracking* that was developed by Field and Tip [24, 37]. For a given sequence of rewriting steps $T_0 \rightarrow \cdots \rightarrow T_n$, dependence tracking computes a slice of the original term, T_0, for each function symbol or subcontext (a notion that will be presented below) of the result term, T_n.

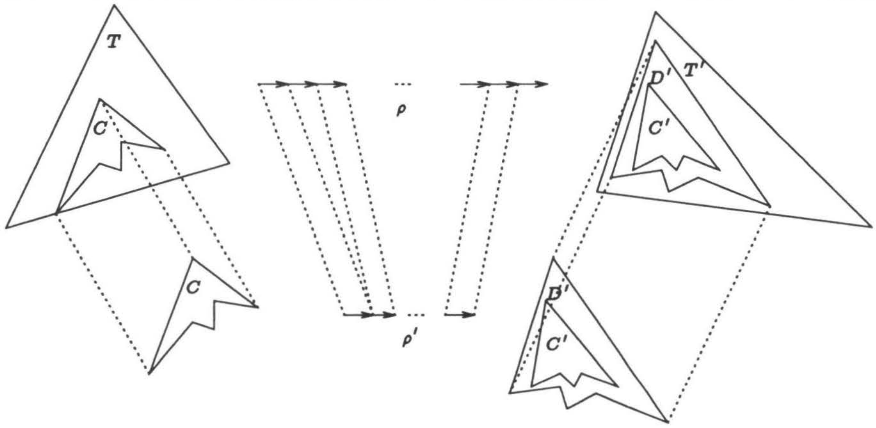


Figure 6: Depiction of the definition of a term slice.

We will use the following simple specification of integer arithmetic (taken from [38]) as an example to illustrate dependence tracking:

```
[A1]  intmul(0, X)                    =  0
[A2]  intmul(intmul(X, Y), Z)         =
           intmul(X, intmul(Y, Z))
```

By applying these equations, the term intsub(3, intmul(intmul(0, 1), 2)) may be rewritten as follows (subterms affected by rule applications are underlined):

$$
\begin{array}{rl}
T_0 = & \texttt{intsub(3,}\underline{\texttt{intmul(intmul(0, 1), 2)}}\texttt{)} \\
 & \qquad\qquad \longrightarrow_{[\mathrm{A2}]} \\
T_1 = & \texttt{intsub(3,}\underline{\texttt{intmul(0, intmul(1, 2))}}\texttt{)} \\
 & \qquad\qquad \longrightarrow_{[\mathrm{A1}]} \\
T_2 = & \qquad\quad \texttt{intsub(3,0)}
\end{array}
$$

By carefully studying this example, one can observe the following:

- The outer context intsub(3, ●) of T_0 ('●' denotes a missing subterm) is not affected at all, and therefore reappears in T_1 and T_2.
- The occurrence of variables X, Y, and Z in both the left-hand side and the right-hand side of [A2] causes the respective subterms 0, 1, and 2 of the underlined subterm of T_0 to reappear in T_1.
- Variable X only occurs in the left-hand side of [A1]. Consequently, the subterm intmul(1, 2) (of T_1) that is matched against X does not reappear in T_2. In fact, we can make the stronger observation that the subterm matched against X is *irrelevant* for producing the constant 0 in T_2: the "creation" of this subterm 0 only requires the presence of the context intmul(0, ●) in T_1.

The above observations are the cornerstones of the dynamic dependence relation of [24, 37]. Notions of *creation* and *residuation* are defined for single rewrite-steps. The former involves function symbols produced by rewrite rules whereas the latter corresponds to situations where symbols are copied, erased, or not affected by rewrite rules[4]. Figure 5 shows all residuation and creation relations for the example reduction discussed above.

Roughly speaking, the dynamic dependence relation for a sequence of rewriting steps ρ consists of the transitive closure of creation and residuation relations for the individual steps in ρ. In [24, 37], the dynamic dependence relation is defined as a relation on *contexts*, i.e., connected sets of function symbols in a term. The fact that C is a *subcontext* of a term T is denoted $C \sqsubseteq T$. For any sequence of rewrite steps $\rho : T \rightarrow \cdots \rightarrow T'$, a *term slice* with respect to some $C' \sqsubseteq T'$ is defined as the subcontext $C \sqsubseteq T$ that is found by tracing back the dynamic dependence relations from C'. The term slice C satisfies the property that C can be rewritten to a term $D' \sqsupseteq C'$ via a sequence of rewrite steps ρ', where ρ' contains a subset of the rule applications in ρ. This property is illustrated in Figure 6.

Returning to the example, we can determine the term slice with respect to the entire term T_2 by tracing back all creation and residuation relations to T_0. The reader may verify that the term slice with respect to `intsub(3, 0)` consists of the context `intsub(3, intmul(intmul(0, •), •))`.

The bottom window of the CLaX environment of Figure 1 is a textual representation of a term that represents a list of errors. The slices shown in Figure 2(a)–(d) are computed by tracing back the dependence relations from each of the four "error" subterms.

4 The Effect of Determinism on Slice Accuracy

We have argued that our approach for obtaining positional information does not rely on a specific specification style. Nevertheless, experimentation with the CLaX type checker has revealed that the *accuracy* of the computed slices inversely depends on the degree to which the specification is *deterministic*. As a general principle, *more* determinism in a specification leads to *less* accurate slices. To understand why this is the case, consider the nature of dynamic dependence relations. Suppose that type checking a program P involves a sequence of rewrite steps r that ultimately lead to an error e. The slice P_e associated with e has the property that it can be rewritten to a term containing e, using a subset r' of the rewrite-steps in r. If the rewrite steps in r encode a deterministic process such as the explicit traversal of a list of statements, this deterministic behavior will also be exhibited by r', to the extent that it contributed to the creation of e.

As an example, consider rewriting the term:

```
type-of(tenv( x :  integer; y:  string;
```

[4] The notions of creation and residuation become more complicated in the presence of so-called *left-nonlinear* rules and *collapse rules*. This is discussed at greater length in [24, 37].

```
                z :  integer), y)
```

according to the specification of Figure 3. By applying equation [Eq6], this term rewrites to the constant `string`. By tracing back the dynamic dependence relations, we find that the context

```
type-of(tenv(●; y:  string; ●), y)
```

was needed to create this result. Now suppose that instead of equation [Eq6], we use the following two equations for reducing the same term:

[Eq6a] `type-of(`*Id*`, tenv(`*Id*`:`*Type*`;` *D**`)) =`
Type

[Eq6b] `type-of(`*Id*`, tenv(`*Id'*`:`*Type*`;` *D**`)) =`
`type-of(`*Id*`, tenv(` *D**`))`
`when` *Id'* `!=` *Id*

The resulting term would be the same as before: the constant `string`, which is obtained by first applying equation [Eq6b] followed by applying equation [Eq6a]. However, the subcontext needed for creating this result would now consist of:

```
type-of(tenv(x :  ●; y:  string; ●), y)
```

The variable `x` in the first element of the type environment is now included in the slice because the *order* in which the type environment is traversed is made explicit in the specification. Informally stated, the resulting term `string` is now dependent on the fact that the first element of the type environment is not an entry for variable `y`.

The use of list functions and list matching in specifications (i.e., allowing function symbols with a variable number of arguments and variables that match sublists) has the effect of reducing determinism, and therefore improving slice accuracy. We believe that more powerful mechanisms for expressing nondeterminism such as higher-order functions [27] can in principle improve slice accuracy even further.

Experimentation with the CLaX type checker specification of [17] revealed a small number of cases where slices were unnecessarily inaccurate due to overly deterministic behavior. Virtually all of these cases consisted of explicit traversals of lists, with the purpose of finding a specific list element, or verifying whether or not a list contained a certain element more than once. In each of these cases, the use of list functions allowed us to specify the same function nondeterministically with little effort. In a forthcoming technical report [18], we will present a brief overview of a few of the more interesting changes we made to the CLaX specification in order to make it less deterministic.

5 Related Work

The work presented in this paper is closely related to earlier work by the same authors. The CLaX type checker [17] was developed in the context of the COMPARE (compiler generation for parallel machines) project, which was part of the European Union's ESPRIT-II program. We originally used *origin tracking* [14] to associate source locations with type errors. Origin tracking is similar in spirit to dependence tracking in the sense that it establishes relationships between subterms of terms that occur in a rewriting process. The key difference between the two techniques is that origin tracking establishes relationships between *equal* subterms (either syntactically equal, or equal in the algebraic sense), whereas dependence tracking determines for each subterm the context needed to create it. The use of origin tracking for obtaining positional information was further investigated in [15, 16]. Although the results were encouraging (in terms of accuracy of positional information), origin tracking was found to impose restrictions on the style in which the type checker specification was written. Since origin tracking only establishes relationships between equal terms, the error messages generated by the type checker must contain fragments that literally occur in the program source; otherwise, positional information is unavailable. In [15, 16], this problem was circumvented by tokenization, i.e., using an applicative syntax structure and rewriting the specification in such a way that error messages always contain literal fragments of program source, which guarantees the non-emptyness of origins. Modification of the type checker specification resulted in adequate positional information for type errors. By contrast, our approach does not require any modifications to specifications at all. In the previous section, we have described techniques for improving the quality of positional information by avoiding determinism in specifications, but it should be emphasized that such improvements are completely optional.

The dependence tracking relation we use for obtaining positional information was developed by Field and Tip [24, 37] for the purpose of computing program slices. A *program slice* [42, 43, 39] is usually defined as the set of statements in a program P that may affect the values computed at the *slicing criterion*, a designated point of interest in P. Two kinds of program slices are usually distinguished. *Static* program slices are computed using compile-time dependence information, i.e., without making assumptions about a program's inputs. In contrast, *dynamic* program slices are computed for a specific execution of a program. An overview of program slicing techniques can be found in [39].

By applying dependence tracking to different rewriting systems, various types of slices can be obtained. In [23] programs are translated to an intermediate graph representation named PIM [22, 1]. An equational logic defines the optimization/simplification and (symbolic) execution of PIM-graphs. Both the translation to PIM and the equational logic for simplification of PIM-graphs are implemented as rewriting systems, and dependence tracking is used to obtain program slices for selected program values. By selecting different PIM-subsystems, different kinds of slices can be computed, allowing for various cost/accuracy tradeoffs to be made. In [38], dynamic program slices are obtained by applying dependence tracking to a previously written specification for a CLaX-interpreter.

The slice notion presented in the current paper differs from the traditional program slice concept in the following way. In program slicing, the objective is to find a projection of a program that preserves part of its *execution* behavior. By contrast, the slice notion we have used here is a projection of the program for which part of another program property—*type checker* behavior—is preserved. It would be interesting to investigate whether there are other abstract program properties for which a sensible slice notion exists.

Another approach to providing positional information for type errors is pursued by van Deursen [10, 9]. Van Deursen investigates a restricted class of algebraic specifications called Primitive Recursive Schemes (PRSs). In a PRS, there is an explicit distinction between constructor functions that represent language constructs, and other functions that process these constructs. Van Deursen extends the origin tracking notion of [14] by taking this additional structure into account, which enables the computation of more precise origins.

Heering [27] has experimented with higher-order algebraic specifications to specify static semantics. We believe that the approach of this paper would work very well with higher-order specifications, since these allow one to avoid deterministic behavior, which adversely affects slice accuracy. However, this would require extension of the dependence tracking notion of [24, 37] to higher-order rewriting systems.

Fraer [26] uses a variation on origin tracking [6, 5, 7] to trace the origins of assertions in a program verification system. In cases where an assertion cannot be proved, origin tracking enables one to determine the assertions and program components that contributed to the failure of the verification condition.

Flanagan et al. [25] have developed MrSpidey, an interactive debugger for Scheme, which performs a static analysis of the program to determine operations that may lead to run-time errors. In this analysis, a set of abstract values is determined for each program construct, which represents the set of run-times values that may be generated at that point. These abstract values are obtained by deriving a set of constraints from the program in a syntax-directed fashion, which approximate the data flow in the program. In addition, a value flow graph is constructed, which models the flow of values between program points. MrSpidey has an interactive user-interface that allows one to visually inspect values as well as flow-relationships.

6 Conclusions

We have presented a slicing-based approach for determining locations of type errors. Our work assumes a framework in which type checkers are specified algebraically, and executed by way of term rewriting [32]. In this model, a type check function rewrites a program's abstract syntax tree to a list of type errors. Dynamic dependence tracking [24, 37] is used to associate a *slice* [42, 39] of the program with each error message. Unlike previous approaches for automatic determination of error locations [17, 15, 16, 10, 9, 6, 5, 7], ours does not rely on a specific specification style, nor does it require additional specification-level information for tracking locations. The computed slices have an interesting semantic property: The slice P_e associated with error message e is a projection of the original program P that, when type checked, is guaranteed to produce the same type error e.

We have implemented this work in the context of the ASF+SDF Meta-environment [31, 12] for a substantial subset of Pascal. Experimentation with CLaX revealed that the computed slices provide highly insightful information regarding the nature of type violations. We have observed that the amount of determinism in a specification is an important factor that determines the accuracy of the computed slices, and we consider this to be a topic that requires further study. As another direction for future work, we intend to study the applicability of slicing-based error location in the related area of type *inference* [8], in particular for object-oriented languages [35] and for ML [34]. Providing accurate positional information for type inference errors in ML is a difficult problem. Several proposals that rely on adapting or extending the underlying type system or inference algorithm have been presented (see, e.g., [4, 41]). In contrast, we are interested in an approach that requires no changes to type inference algorithm or the type system. The basic idea is to apply dependence tracking to a rewriting-based implementation of an ML type inferencer. Although a slice can be computed for each reported type inference error, it is unclear how accurate such slices will be in practice.

References

[1] Bergstra, J., Dinesh, T., Field, J., and Heering, J. A complete transformational toolkit for compilers. In *Proc. European Symposium on Programming* (Linköping, Sweden, April 1996), vol. 1058 of *Lecture Notes in Computer Science*, Springer-Verlag. Full version: Technical Report CS-R9646, Centrum voor Wiskunde en Informatica (CWI), Amsterdam; To appear in TOPLAS, 1997.

[2] Bergstra, J., Heering, J., and Klint, P., Eds. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[3] Bergstra, J., and Klop, J. Conditional rewrite rules: confluence and termination. *Journal of Computer and System Sciences 32*, 3 (1986), 323–362.

[4] Bernstein, K. L., and Stark, E. W. Debugging type errors (full version). Tech. rep., State University of New York at Stony Brook, Computer Science Department, 1995.

[5] Bertot, Y. Occurrences in debugger specifications. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (1991), pp. 327–337. *SIGPLAN Notices* 26(6).

[6] Bertot, Y. *Une Automatisation du Calcul des Résidus en Sémantique Naturelle.* PhD thesis, INRIA, Sophia-Antipolis, 1991. In French.

[7] BERTOT, Y. Origin functions in lambda-calculus and term rewriting systems. In *Proceedings of the 17th Colloquium on Trees in Algebra and Programming (CAAP '92)* (1992), J.-C. Raoult, Ed., vol. 581 of *LNCS*, Springer-Verlag.

[8] CLÉMENT, D., DESPEYROUX, J., DESPEYROUX, T., AND KAHN, G. A simple applicative language: Mini-ml. In *Proc. 1986 ACM Symposium on Lisp and Functional Programming* (1986), pp. 13–27.

[9] DEURSEN, A. V. *Executable Language Definitions—Case Studies and Origin Tracking Techniques.* PhD thesis, University of Amsterdam, 1994.

[10] DEURSEN, A. V. Origin tracking in primitive recursive schemes. Report CS-R9401, Centrum voor Wiskunde en Informatica (CWI), 1994.

[11] DEURSEN, A. V., AND DINESH, T. Origin tracking for higher-order term rewriting systems. In *Proceedings of the International Workshop on Higher-Order Algebra, Logic and Term Rewriting* (Amsterdam, September 1993), J. Heering, K. Meinke, B. Möller, and T. Nipkow, Eds., vol. 816 of *LNCS.*

[12] DEURSEN, A. V., HEERING, J., AND KLINT, P., Eds. *Language Prototyping—An Algebraic Specification Approach,* vol. 5 of *AMAST Series in Computing.* World Scientific, 1996.

[13] DEURSEN, A. V., KLINT, P., AND TIP, F. Origin tracking. Report CS-R9230, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992.

[14] DEURSEN, A. V., KLINT, P., AND TIP, F. Origin tracking. *Journal of Symbolic Computation 15* (1993), 523–545.

[15] DINESH, T. B. Type checking revisited: Modular error handling. In *Semantics of Specification Languages* (1994), D. J. Andrews, J. F. Groote, and C. A. Middelburg, Eds., Workshops in Computing, Springer-Verlag, pp. 216–231. Utrecht 1993.

[16] DINESH, T. B. Typechecking with modular error handling. In *Language Prototyping: An Algebraic Specification Approach,* A. v. Deursen, J. Heering, and P. Klint, Eds. World Scientific Publishing Co., 1996, pp. 85–104.

[17] DINESH, T. B., AND TIP, F. Animators and error reporters for generated programming environments. Report CS-R9253, Centrum voor Wiskunde en Informatica (CWI), 1992.

[18] DINESH, T. B., AND TIP, F. A slicing-based approach for locating type errors. Tech. rep., CWI/IBM, 1997. Forthcoming.

[19] DINESH, T. B., AND ÜSKÜDARLI, S. *Input and Output for Specified Visual Languages.* Springer, 1997, ch. 12 [33], pp. 321–349.

[20] Dinesh, T. B., and Üsküdarli, S. Pretty-printing of Visual Sentences. In *Proc. 1997 IEEE Symposium Visual Languages* (September 1997).

[21] Dinesh, T. B., and Üsküdarli, S. Share-Where Maintenance in Visual Algebraic Specifications. In *Lecture Notes in Computer Scinece* (December 1997), Springer-Verlag. To appear.

[22] Field, J. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (1992), pp. 98–107. Published as Yale University Technical Report YALEU/DCS/RR–909.

[23] Field, J., Ramalingam, G., and Tip, F. Parametric program slicing. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages* (San Francisco, CA, 1995), pp. 379–392.

[24] Field, J., and Tip, F. Dynamic dependence in term rewriting systems and its application to program slicing. In *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming* (1994), M. Hermenegildo and J. Penjam, Eds., vol. 844, Springer-Verlag, pp. 415–431.

[25] Flanagan, C., Flatt, M., Krishnamuthi, S., Weirich, S., and Felleisen, M. Catching bugs in the web of program invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Philadelphia, PA, 1996), pp. 23–32.

[26] Fraer, R. Tracing the origins of verification conditions. In *Proceedings of AMAST'96* (Munich, Germany, July 1996), vol. 1101, Springer-Verlag LNCS.

[27] Heering, J. Second-order term rewriting specification of static semantics. In *Language Prototyping: An Algebraic Specification Approach*, A. v. Deursen, J. Heering, and P. Klint, Eds. World Scientific Publishing Co., 1996, pp. 295–306.

[28] Kamperman, J. *Compilation of Term Rewriting Systems*. PhD thesis, University of Amsterdam, 1996.

[29] Kamperman, J., and Walters, H. Minimal term rewriting systems. In *Recent trends in data type specification : 11th workshop on specification of abstract data types joint with the 8th COMPASS workshop: Oslo, Norway, 19-23.09.1995 : selected papers* (1996), vol. 1130 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 274–290.

[30] Klint, P. Line numbers made cheap. *Communications of the ACM 22* (1979), 557–559.

[31] KLINT, P. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology 2*, 2 (1993), 176–201.

[32] KLOP, J. Term rewriting systems. In *Handbook of Logic in Computer Science, Volume 2. Background: Computational Structures*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Oxford University Press, 1992, pp. 1–116.

[33] MARRIOTT, K., AND MEYER, B., Eds. *Theory of Visual Languages*. Springer, 1997. To appear.

[34] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.

[35] PALSBERG, J., AND SCHWARTZBACH, M. *Object-Oriented Type Systems*. John Wiley & Sons, 1993.

[36] TIP, F. Animators for generated programming environments. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging* (1993), P. Fritzson, Ed., vol. 749 of *LNCS*, Springer-Verlag, pp. 241–254.

[37] TIP, F. *Generation of Program Analysis Tools*. PhD thesis, University of Amsterdam, 1995.

[38] TIP, F. Generic techniques for source-level debugging and dynamic program slicing. In *Proceedings of the Sixth International Joint Conference on Theory and Practice of Software Development* (Aarhus, Denmark, May 1995), P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, Eds., vol. 915 of *LNCS*, Springer-Verlag, pp. 516–530.

[39] TIP, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (1995), 121–189.

[40] ÜSKÜDARLI, S. M. *Algebraic Specification of Visual Languages*. PhD thesis, University of Amsterdam, March 1997.

[41] WAND, M. Finding the source of type errors. In *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL, 1986), pp. 38–43.

[42] WEISER, M. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[43] WEISER, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.

ASF + SDF voor Marsbewoners

Peter van Emde Boas
ILLC-WINS-Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam
peter@wins.uva.nl

Onze vriend de Marsbewoner is een intelligent bewust schepsel, die zich met grote inzet en belangstelling gestort heeft op het Aardse in de hoop enig inzicht te verkrijgen in de wijze waarop het hier toegaat. Hij bezit een voldoende kennis van talen en onderdelen van de wetenschap om zich binnen de Universitaire gemeenschap te kunnen handhaven. Zijn vermogen om zich te kunnen inleven in systemen (al dan niet formeel) op basis van een oppervlakkige kennisname heeft mij telkenmale versteld doen staan. De wijze waarop hij daarbij regelmatig tot onvermoede maar soms ook incorrecte inzichten geraakt leert mij eerder na te denken over aard en gebruik van deze systemen, dan dat het ooit mijn twijfel heeft opgeroepen inzake de ongekende geestelijke capaciteiten van de Marsman.

Onze onderlinge relatie is de afgelopen jaren tamelijk intiem geworden. Dit uit zich mede in het feit dat ik bij vele gelegenheden de Marsman weet te betrekken bij mijn dagelijkse activiteiten, zonder dat dit bij de overige aanwezigen opvalt (tenslotte is zijn lichaamsomvang zodanig dat hij gemakkelijk in mijn vestzakje past). Op deze wijze is hij in de afgelopen jaren getuige geweest van het merendeel van de Doctoraalexamens in de Informatica, waarbij ik ambsthalve aanwezig moet zijn.

Recentelijk bereikte mij het bericht van het aanstaande 25-jarige jubilaeum van Paul Klint aan het CWI. Het leek mij een goede gelegenheid om uit te zoeken wat het beeld is dat de Marsman van de jubilaris heeft gekregen op grond van zijn waarnemingen. Een korte samenvatting van zijn uitspraken wil ik in deze bijdrage aan de feestbundel voor Paul Klint verwoorden.

Het is de Marsman uiteraard opgevallen dat waar gesproken wordt over Paul Klint al spoedig het thema ASF + SDF ter tafel komt. Omdat het hier een formalisme betreft is de interesse van de Marsman zonder meer aanwezig. De Marsman is echter een lui persoon die geen enkele aanvechting heeft om boeken en technische manualen te bestuderen, want dat vindt hij maar grofstoffelijk geploom. Zijn visie op het onderwerp blijft derhalve beperkt tot al datgene wat hij vanuit mijn vestzakje heeft kunnen horen tijdens de examens (en promoties) en het weinige dat hij vanuit deze krappe positie heeft meegekregen van het vele leesvoer dat ik regelmatig voor de kiezen krijg.

Al overleggende kwamen wij tot het inzicht dat er een aantal centrale vragen zijn met betrekking to ASF + SDF:

1. Wat hebben wij gezien en begrepen van het potentieel van het formalisme
2. Kunnen wij inzien dat het formalisme alles kan
3. Zijn er begrenzingen aan wat het formalisme kan

Voor de beantwoording van de eerste vraag hebben wij de experimentele methode gehanteerd. Wij nemen de lijst van scripties van studenten van Paul Klint die zijn afgestudeerd bij de Amsterdamse Informatica:

Lijst Doctoraalstudenten van Paul Klint; Informatica Universiteit van Amsterdam

Datum	**Naam** **Titel scriptie**	**Variant**
19860827	Joan Gerard Rekers *A parser generator for Finitely Ambiguouos Context Free Grammars*	Programmatuurkunde
19860922	Paulus Jan Veerkamp *Viewing and Hidden Surface Removal; Report on a GKS/3D project*	Programmatuurkunde
19880229	Monique Maria Megens *An Implementation of a Simple Design Description Language*	Programmatuurkunde
19880511	Victoria Webster de Mey *Object-Oriented Programming Concepts and Their Use in Application Development*	Programmatuurkunde
19890228	Caspar Hendrik Sipke Dik *A fast implementation of the Algebraic Specification Formalism*	Programmatuurkunde(CL)
19890830	Lars Hendrik Jacobson *A Token Ring Network in PSF_d & A Tool Interface Language for PSF_d*	Programmatuurkunde
19901031	Karin Jacoba Vos *Pretty, for an easy touch of beauty*	Programmatuurkunde (CL)
19900831	Tanja van Rij *Intelligent prototype for simulation of discrete production processes*	Robotiek en Kunst. Intelligentie
19901121	Walter van der Laan *A program generator for database applications*	Programmatuurkunde
19910403	Frank Tip *The equation debugger*	Programmatuurkunde (CL)
19911202	Wouter Pasman *Incremental Parsing*	Programmatuurkunde
19911211	Hanna Luden *Software Configuration Management*	Programmatuurkunde
19920212	Richard Peter van der Luit *An algebraic specification of the relational database language SQL*	Programmatuurkunde
19920831	Khalil Sima'an *Detecting ambiguity in ε-free context free grammars*	Programmatuurkunde(CL)
19930618	Eelco Visser	Programmatuurkunde (CL)

Combinatory algebraic specification & compilation of list matching

19930712 Jeroen Johannes Entken Programmatuurkunde
A prototype of a simulator for hydraulic systems

19930831 Gerrit Vos Programmatuurkunde
Algebraïsche toonsystemen

19940620 Catharina Zaadnoordijk Programmatuurkunde
An ASF+SDF specification of a query optimizer for a RDBMS

19940620 Liang Hwie Oei Programmatuurkunde
Giving meaning to an IDEAL
Part 1: Syntax and static semantics of interlocking design and application language
Part 2: Pruning the search tree of interlocking design and application language operational semantics

19940829 Pieter Antonius Olivier Programmatuurkunde
Seal versus the Toolbus

19940829 Martinus Res Programmatuurkunde
A generated programming environment for Risla

19950704 Judi Maria Tirza Romijn Programmatuurkunde
Automatic analysis of term rewriting systems

19950718 Seyyed Kamal Koochak Kosari Programmatuurkunde
A peephole optimization for an abstract rewriting machine

19950825 Dennis Karl Adrian Geluk Programmatuurkunde
Program transformation using ASF + SDF

19950825 Saskia Monique Eijkelkamp Programmatuurkunde
Program transformation using ASF + SDF

19950825 Dan Andres Jernberg Programmatuurkunde
Incremental specfications

19950825 Joshua Narteh Sappore-Siaw Programmatuurkunde
Equational debugger; an experimantal debugger for the ASF + SDF term rewriting system

19950831 Fred Osei Bonsu Programmatuurkunde
Graphic generation language; automatic code generation from design

19950831 Ralph Oudejans Programmatuurkunde
Graphic generation language; automatic code generation from design

19950831 Martijn Kaart Programmatuurkunde
Modelling of a system for traffic regulation at signailzed intersections using high-level Petri nets

19951012 Anthonios Thanos Programmatuurkunde
Some recommendations for solving consistency problems in Uniface Six

19951201 Michiel Johan Koens Programmatuurkunde
Generation of forms based editors for domain specific formalisms (CL)

19960425 Edward Mike Kien Tit Tai-A-Fat Programmatuurkunde
Using the DeltaBlue Constraint Solver in a programming environment

19960830 Johannes Gerardus Maria Al Programmatuurkunde
COM in het sociaal verzekeringsdomein; onderzoek naar infrastructuren voor componenten

19960830 Serge Louis Barth Programmatuurkunde
ToBox: a rule-driven front-end for prettyprinting AsFix

19961015 Leonardus Marinus Feanciscus Moonen Programmatuurkunde(CL)
Data flow analysis for reverse engineering

19961120 Bernard Martin Caspers Vrij doctoraal
The CORBA experience; realizing an Object-Oriented architecture for internet client/server systems

19961128 Peter Ivan Manuel Programmatuurkunde
ANSI COBOL III in SDF + an ASF definition of a Y2K tool

19961220 Tobias Kuipers Programmatuurkunde
Language independent structure editing

19970704 Egbert Jan van Buiten Programmatuurkunde
Graphic generation language[II]

19970704 Timme Kees Dekker Programmatuurkunde
Graphic generation language[II]

19970704 Ahmed Abd Elsalam Ahmed Ebrahim Programmatuurkunde
System renovation using cluster analysis

19970820 Maarten Kees van der Graaf Programmatuurkunde
A specification of Box to HTML in ASF+SDF

19970829 Harold Frank Breebaart Programmatuurkunde
Building an editor for defining structured constraint-based graphical objects

Inspectie van deze lijst levert enkele interssante conclusies op. Het feit dat Paul Klint zijn naam in mijn boeken vermeld ziet worden achter 46 studenten (van de 496 afgestudeerden per 19970930), dwz. ruim 9 % van alle afgestudeerden) is een prestatie, zeker gezien het feit dat Paul slechts voor een deel van zijn tijd bij de UvA in dienst is. Proficiat! Pauls bijdrage is nog groter als wij vaststellen dat hij van de 36 Cum Laude studenten er 7 heeft afgelevert, en eveneens 7 van de 51 dames. Het meest opvallende aspect is echter dat het overgrote deel van de scripties handelt over toepassingen van ASF + SDF, dan wel over technologie voor de ontwikkeling en implementatie van programmeertalen die rechtstreeks verband lijken te hebben met het grote project. Bij de toepassingen treffen wij onderwerpen aan als hidraulica, databases, financiele producten, verkeer, Sociale verzekeringen, graphics en visuele systemen, en het Jaar 2000 probleem. Overigens is het aardig te kunnen vermelden dat in deze lijst ook de oudste doctoraalscriptie gewijd aan het thema Object Orientatie aan onze instelling is opgenomen.

Op grond van dit experimentele onderzoek kan de Marsman dus slechts één conclusie trekken: *in ASF + SDF kan alles worden gespecificeerd.*

Voor wat betreft de tweede vraag waren de Marsman en de schrijver het snel eens. Om vast te stellen dat een formalisme alles kan gebruiken wij de klassieke methode: men neme een ander formalisme waarvoor deze vraag reeds positief is beantwoord (Turing Machines of zo iets) en tone aan dat dit formalisme zich in ASF + SDF laat specificeren. Zelfs op grond van ons minimale inzicht konden wij gemakkelijk vaststellen dat dit het soort opgave is waarvan verwacht mag worden dat een gemiddelde student van het college Software Engineering deze moeiteloos zal kunnen maken. Waar wij het echter niet eens over konden worden is de keuze van het meest geschikte formalisme om als uitgangspunt bij deze meesterreductie te worden gebruikt. De grammaticale component suggereert om een type 0 grammatica na te spelen via bv. de conditionele herschrijfregels. Men kan ook denken aan regelrechte simulatie van Turingmachines of Post Tag systemen als een speciaal geval van dit idee. Een alternatief is echter het gebruik van de karakterisering van de recursief opsombare relaties via Diophantische predicaten (Hilbert 10), want die hebben alles te maken met huis, tuin en keuken rekenen en dat is nu eenmaal een wereld die regelmatig om de hoek komt binnen de scripties en afstudeervoordrachten als motiverend voorbeeld.

Het is ons wel opgevallen dat het onderwerp *universaliteit* nooit aan de orde komt in de passages die de studenten aan het ASF + SDF systeem wijden in hun scriptie. Is het te zelf-evident om te worden vermeld of ziet men het helemaal niet als een relevante issue (omdat het slechts een theoretische vraagstelling is waarvan het antwoord niets bijdraagt aan de ontwikkeling van taalomgevingen....). Misschien toch een suggestie om hier in de toekomst enige opmerkingen over te laten maken.

Komen wij tot de laatste vraag: Wat zijn de begrenzingen van de kracht van het formalisme?

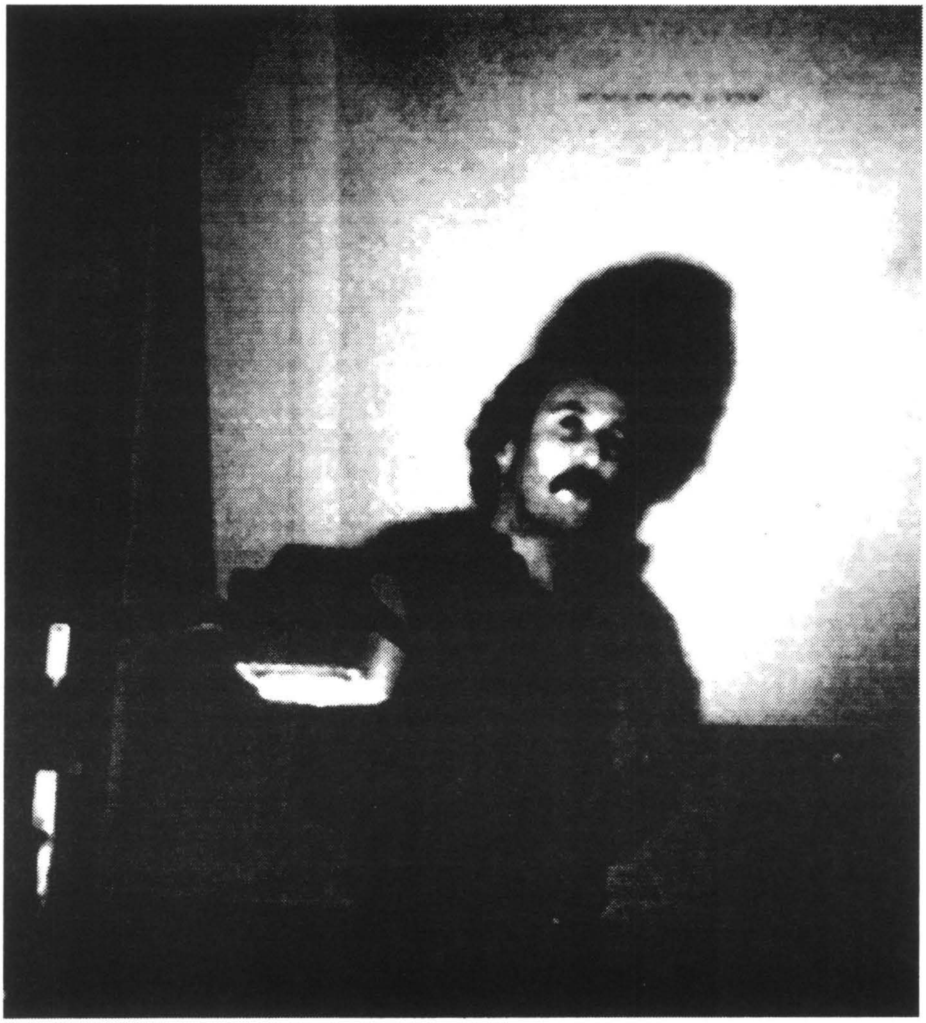
Voor de beantwoording van deze vraag wreekt zich dat de vraag een veel nauwgekeuriger analyse van het object dat in ASF + SDF beschreven wordt vraagt dan in het algemeen wordt aangedragen door de scripties. Mijn vriend de Marsman heeft zich eraan gewend te denken in termen van algebra's bepaald door een theorie van conditionele gelijkheden, en daar de modellen van te beschouwen. Helaas blijkt het feitelijke formalisme nog wel eens van aard te veranderen, waarbij de dynamica van het term-herschrijven een veel grotere rol blijkt te spelen dan de buitenstaander zou vermoeden. Wat ik eerder constateerde over het ontbreken van uitspraken over de universaliteit geldt al evenzeer ten aanzien van de beoogde semantieken. Het zal wel weer een veel te theoretische vraagstelling zijn.

Wij hadden Paul voor deze gelegenheid dan ook gaarne bij wijze van geschenk een net opgeschreven bewijs willen doen toekomen van de voor de hand liggende uitspraak dat de kracht van ASF + SDF niet groter is dan die van andere effectieve formalismen. Een klassieke aanpak gebaseerd op Gödelisering van herschrijfreeksen zou dit resultaat moeiteloos moeten opleveren. Helaas kwamen wij al snel tot het inzicht dat dit bewijs niet opgeschreven kan worden aangezien de bewering onjuist is. De combinatie van conditionele herschrijfregels met prioriteiten stelt de gebruiker in staat een willekeurige recursief opsombare conditie te complementeren en vervolgens door te rekenen. Met de aldus verkregen Σ_2 predicaten kunen wij dezelfde truuk uithalen en aldus willekeurig ver de arithmetische hierarchie inklimmen. Onze informanten beweren dat dit geen nieuw inzicht is: deze jump in expressieve kracht is bij de deskundigen bekend, waarbij bovendien de bodem van deze put van ellende nog lang niet in zicht is.

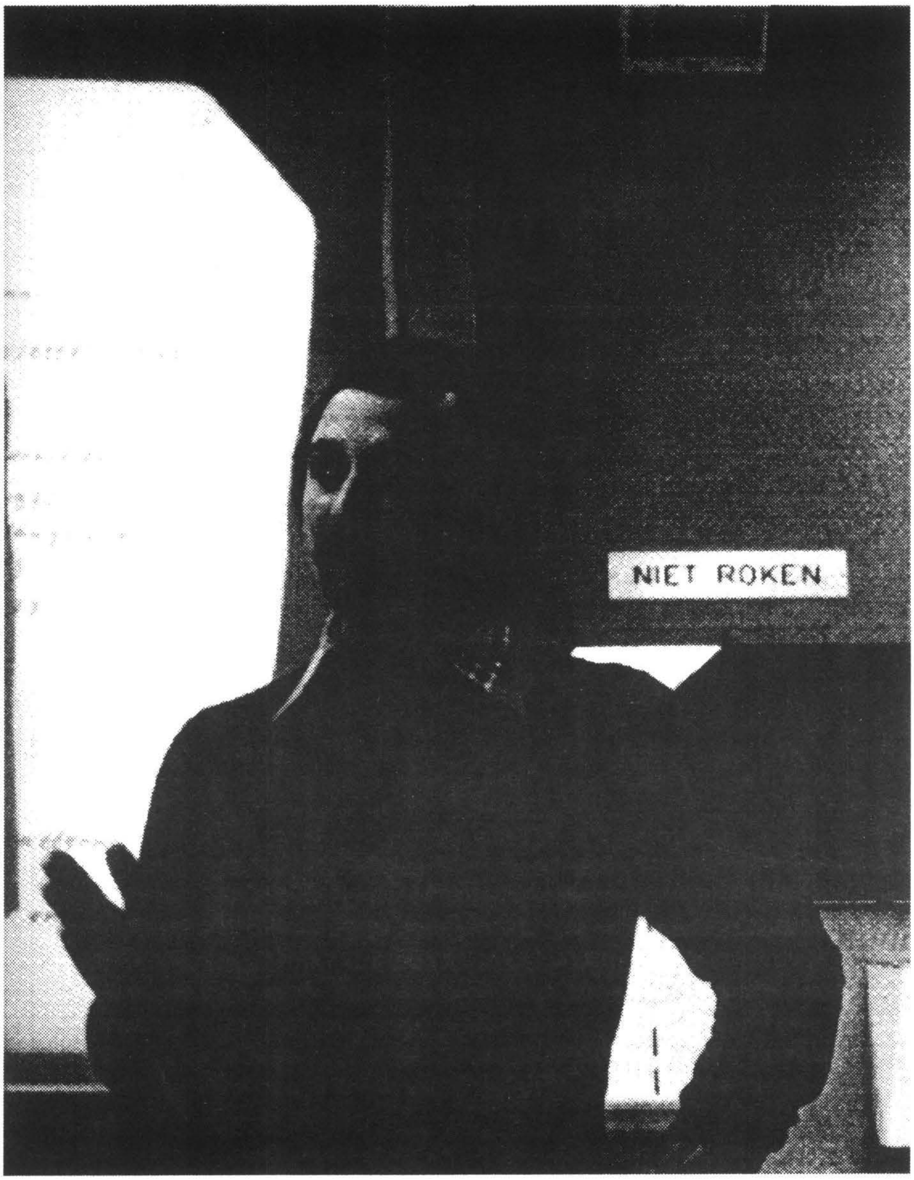
Op Mars zijn de regels behoorlijk streng: het gebruik van niet recursief opsombare formalismen wordt beschouwd als ketterij en kan de bedrijver ervan zowel professioneel als sociaal zwaar isoleren. Wellicht een problematiek om eens nader uit te werken in het kader van een popularisatie van ASF + SDF.

19970925

13e Ned. Wisk. Congres (Erasmus Universiteit Rotterdam—6/7 april 1977).



MC Colloquium Datastructuren (11 november 1977).



Algemeen Colloquium SFW (UvA—6 december 1977).

Why men (and octopuses) cannot juggle a four ball cascade

A. Engels, S. Mauw
Department of Mathematics and Computing Science,
Eindhoven University of Technology,
P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands.
engels@win.tue.nl, sjouke@win.tue.nl

Abstract

The most common juggling pattern is the three ball cascade. A four ball cascade seems to be infeasible, but why? We formalize the notion of a cascade and the decomposition of juggling patterns in order to give an answer to this question.

1 Introduction

It is not difficult to learn how to juggle. One only needs a few hours of practice and some perseverance in order to master the most common pattern, the three ball cascade (see Figure 1). This cascade is the most symmetric and regular pattern that can be thrown with three balls. The two hands move alternately and the balls are also thrown in a fixed alternating order.

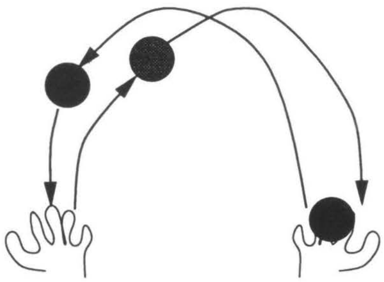


Figure 1: A three ball cascade

After having mastered a three ball cascade, the novice juggler asks himself the question how to juggle with four balls. One would think that juggling four balls is qualitatively the same as juggling three balls, be it with harder constraints for timing and accuracy. But how good his physical skills progress, a four ball cascade does not seem to be possible. People come up with more asymmetric patterns, such as the *shower* (see Figure 1) or with compound patterns such as *two columns* (see Figure 1).

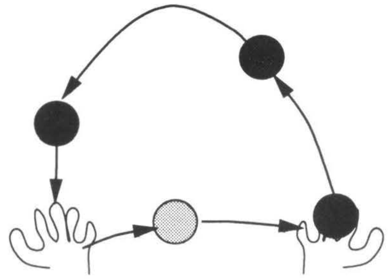


Figure 2: A shower

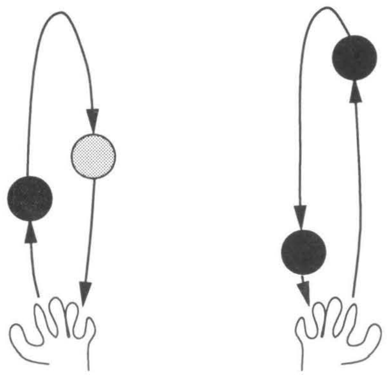


Figure 3: Two columns

Juggling a four ball cascade with two hands seems to be impossible, but why? It is certainly not due to a lack of skills, because experienced jugglers have the same problem. It is possible to juggle a cascade with every odd number of balls (up to 11, which is the current world record), but a cascade with an even number of balls seems impossible. There is probably a more fundamental reason. In this paper we try to formulate an answer.

Thereto, we will give a formal definition of the notion of a cascade and using simple number theory, we will show that two hands cannot throw a four ball cascade.

Because we do not fix the number of hands in our definitions, our result will be more general. The main theorem that we will derive is the following. It is only possible to juggle a cascade if the number of hands and the number of balls are relatively prime, i.e. have no common divisors larger than 1. This theorem has already been stated (without a proof) as a proposition accompanying the thesis of one of the authors (see [2]).

The extension to more than two hands is not purely artificial. In advanced juggling patterns, more than one juggler are involved.

In the remainder of this paper we first define a notation for juggling patterns and the notion of a cascade. By introducing partitions of a juggling pattern, we can make a distinction between a real cascade and a compound cascade.

Thanks are due to all members of the *Department of Pure Juggling* in the Watergraafsmeer (Amsterdam) for the fruitful discussions during the practical and theoretical juggling sessions.

2 notation

It has been recognized by the juggling community that some notation for communicating juggling patterns is needed. A variety of notations are used, ranging from completely informal cartoons to a compact and formal notation such as *site swaps* (see [1]).

Since we intend to use mathematical methods for analyzing juggling patterns, we need a formal notation. Unfortunately, none of the existing notations used lines up neatly with our purposes. Therefore, we introduce a simple and ad-hoc notation for expressing juggling patterns.

In order to capture the essence of a juggling pattern, we have to abstract from physical properties, such as size and speed of the balls, direction of the throws, etc. The only essential information is the order in which the hands throw the balls.

We will describe a juggling pattern as a sequence of pairs $\begin{pmatrix} h_1 \\ b_1 \end{pmatrix} \begin{pmatrix} h_2 \\ b_2 \end{pmatrix} \begin{pmatrix} h_3 \\ b_3 \end{pmatrix} \ldots$, where the h_i are chosen from some finite set of hands H and the b_i from some finite set of balls B. This sequence means that first ball b_1 is thrown from hand h_1, then b_2 from h_2, etc.

Several observations can be made about this notation. First, we did not include the catching of a ball as a separate event. Although the timing of the catches influences the taste of the pattern, we do not think it is essential to the pattern. As a consequence, we are not able to express the difference between *hot potato juggling*, in which the balls stay as short as possible in the hands, and *lazy juggling*, which is the opposite. Secondly, we cannot describe patterns with synchronous throws, which is the case if two throws occur at the same time. Neither is the notation suited to express multiplexing, which means that at certain times more than one ball is in the same hand. Thirdly, we did not include a possibility for explicitly stating that some pattern is executed repeatedly.

However, for many basic patterns, and certainly for a cascade, this notation is all we need.

If $J = \begin{pmatrix} h_1 \\ b_1 \end{pmatrix} \begin{pmatrix} h_2 \\ b_2 \end{pmatrix} \begin{pmatrix} h_3 \\ b_3 \end{pmatrix} \ldots$, we call $h_1 h_2 h_3 \ldots$ its hand pattern and $b_1 b_2 b_3 \ldots$ its ball pattern.

Theorem 1 (periodicity of juggling patterns) Let a juggling pattern J be such that its hand pattern is periodic with period p_H and its ball pattern is periodic with period p_B. Then the pattern itself is periodic with period $\mathrm{scm}(p_H, p_B)$ (where scm stands for smallest common multiple). Furthermore, if the periods p_H and p_B are both prime periods (that is, the patterns are not periodic with any smaller period), then this is the prime period of J.

Proof This follows from the following basic observation. The juggling pattern is periodic with some period p if and only if both the hand and the ball pattern are. ∎

3 The cascade

As stated in the introduction, a cascade is a pattern in which the hands throw alternately and the balls are thrown in a fixed order. This observation leads to the following definition.

Definition 2 (cascade) A cascade is a juggling pattern in which both the hand pattern and the ball pattern are cyclic (that is, the hand pattern is periodic and each hand shows up exactly once in each period, and likewise for the ball pattern).

Under this definition there is exactly one cascade for each given number of hands and balls. But if we look at the 2-hand, 4-ball cascade, we see the following juggling pattern (using the hands L and R for obvious reasons):

$$\binom{L}{1}\binom{R}{2}\binom{L}{3}\binom{R}{4}\binom{L}{1}\binom{R}{2}\binom{L}{3}\binom{R}{4}\cdots$$

Looking at the pattern closely, one notices that it is in fact a superposition of two separate patterns: the L-hand is throwing and catching balls 1 and 3, and the R-hand is throwing and catching balls 2 and 4, without any interaction between the two. The 2-hand, 4-ball cascade is just two 1-hand, 2-ball cascades in parallel (see the *two columns* pattern from Figure 1). Because of this, it does not count as a real cascade. So now the question becomes: Which cascades (and other juggling patterns) are real in the sense that they are not composed from simpler patterns, and which are not? To answer this question, we must first formally define what a real juggling pattern (for reasons which will soon become clear, we will call these prime patterns) is.

4 Closed partitions and Prime patterns

Definition 3 (closed partition) Let $J = \binom{h_1}{b_1}\binom{h_2}{b_2}\ldots$ be a juggling pattern. A closed partition of J is a set $J_1 \ldots J_n$ ($n \in \mathbb{N} \cup \infty$) of subsequences of J, such that:

1. The J_i form a partition of J, that is, each J_i is nonempty, and each pair $\binom{h_j}{b_j}$ is in exactly one of the subsequences J_i.

2. If either $h_i = h_j$ or $b_i = b_j$, then the pairs $\binom{h_i}{b_i}$ and $\binom{h_j}{b_j}$ are in the same subsequence.

A juggling pattern is called prime if and only if its only closed partition is the trivial partition, i.e. $n = 1$ and $J_1 = J$.

We call these pattern prime patterns because they can be used to build up all other patterns, as specified in the following theorem:

Theorem 4 Each juggling pattern J has a unique closed partition $J_1, \ldots, J_n$ such that each J_i is a prime pattern.

Proof If $J_1, \ldots, J_m$ and $J'_1, \ldots, J'_n$ are both closed partitions of J, then a new partition, which partitions all subsequences J_i and J'_i, can be obtained by looking at the subsequences $J_i \cap J'_j$, and discarding those that are empty. Taking such an intersection over all possible closed partitions will give the unique closed partition into prime patterns. ■

Next, our aim is to find some necessary and/or sufficient condition on when a juggling pattern is prime. For this we will use the following basic number theoretic result:

Theorem 5 (Chinese remainder theorem) Let m, n, x and y be 4 numbers such that m and n are relatively prime. Then there exists a number z such that $z \equiv x \mod m$ and $z \equiv y \mod n$.

Proof Can be found in many books on number theory ■

Theorem 6 Let J be a juggling pattern of which the hand pattern is periodic with period p_H and the ball pattern is periodic with period p_B, where p_H and p_B are relatively prime. Then J is a prime pattern.

Proof Let $h \in H$ and $b \in B$ be chosen at random. We will prove that there is a pair $\binom{h}{b}$ in the list J, from which it can easily be shown that J is prime.

Let J be $\binom{h_1}{b_1}\binom{h_2}{b_2}\ldots$, and let m and n be chosen such that $h_m = h$ and $b_n = b$. Now find a number k such that $k \equiv m \mod p_H$ and $k \equiv n \mod p_B$. That such a number exists, follows from the Chinese Remainder Theorem. Because $k \equiv m \mod p_H$ we have $h_k = h_m = h$, likewise $b_k = b$. ■

In this theorem we see that all juggling patterns where the periods of hands and balls are relatively prime are prime patterns, and thus are real patterns, which cannot be subdivided into separate independent patterns. In general the reverse is not true. However, as we shall see in the next Theorem, it is true if we restrict ourselves to cascades only:

Theorem 7 Let J be a cascade with p_H hands and p_B balls. Then J can be partitioned in exactly $\gcd(p_H, p_B)$ prime patterns (where gcd stands for greatest common divisor). Furthermore, each of these prime patters is a cascade with $\frac{p_H}{\gcd(p_H,p_B)}$ hands and $\frac{p_B}{\gcd(p_H,p_B)}$ balls.

Proof We will prove this by actually constructing the prime patterns J_i.

Let $J = \begin{pmatrix} h_1 \\ b_1 \end{pmatrix} \begin{pmatrix} h_2 \\ b_2 \end{pmatrix} \ldots$. We define J_i, for i ranging from 1 to $\gcd(p_H, p_B)$ by $\begin{pmatrix} h_k \\ b_k \end{pmatrix} \in J_i \Leftrightarrow i \mod \gcd(p_H, p_B)$. We claim that this is a closed partition of J, and that each J_i is prime.

It is easy to see that this is indeed a partition of J. To see that it is a closed partition, let $h \in H$ be any hand, and let the numbers m and n be such that $h_n = h_m = h$. Then, as the hand pattern is cyclic with period p_H, $n \equiv m \mod p_H$, and thus $n \equiv m \mod \gcd(p_H, p_B)$ as well, so $\begin{pmatrix} h_m \\ b_m \end{pmatrix}$ and $\begin{pmatrix} h_n \\ b_n \end{pmatrix}$ are in the same subsequence J_i. Of course the proof for hands is analogous, from which it can be seen that this partition is indeed closed.

Now take h and b such, that there are, for some h' and b', pairs $\begin{pmatrix} h \\ b' \end{pmatrix}$ and $\begin{pmatrix} h' \\ b \end{pmatrix}$ in some given J_i. We will prove that there is a pair $\begin{pmatrix} h \\ b \end{pmatrix}$ in J_i, from which we can conclude that J_i is prime.

As there are pairs $\begin{pmatrix} h \\ b' \end{pmatrix}$ and $\begin{pmatrix} h' \\ b \end{pmatrix}$ in J_i, there must be some numbers m and n, such that $h_{m.\gcd(p_H,p_B)+i} = h$ and $b_{n.\gcd(p_H,p_B)+i} = b$. Using the Chinese Remainder Theorem we find that there is a number l such that $l \equiv m \mod \frac{p_H}{\gcd(p_H,p_B)}$ and $l \equiv n \mod \frac{p_B}{\gcd(p_H,p_B)}$ ($\frac{p_H}{\gcd(p_H,p_B)}$ and $\frac{p_B}{\gcd(p_H,p_B)}$ are prime by definition of gcd). From this we can conclude that $l.\gcd(p_H, p_B) \equiv m.\gcd(p_H, p_B) \mod p_H$, so $l.\gcd(p_H, p_B) \equiv m.\gcd(p_H, p_B) + i \mod p_H$, so $h_{l.\gcd(p_H,p_B)+i} = h_{m.\gcd(p_H,p_B)+i} = h$. Likewise $b_{l.\gcd(p_H,p_B)+i} = b_{n.\gcd(p_H,p_B)+i} = b$, so we have found or pair $\begin{pmatrix} h \\ b \end{pmatrix}$ in J, and clearly in J_i as well (either through the definition of J_i, or because the partition of J is closed).

Let the hand pattern going with the juggling pattern J_i be denoted by H_i, and its ball pattern by B_i. It can easily be seen that H_i is a restriction of the hand pattern H to a subset of hands. As such, it can be seen to be cyclic as well. Furthermore, each H_i has an equal part of the set of hands, and as there are $\gcd(p_H, p_B)$ such subsequences, dividing p_H in equal parts, each such H_i contains exactly $\frac{p_H}{\gcd(p_H,p_B)}$ hands. Analogously, the ball pattern B_i is cyclic with $\frac{p_B}{\gcd(p_H,p_B)}$ balls. Thus, the patterns J_i are cascade patterns with $\frac{p_H}{\gcd(p_H,p_B)}$

balls and $\frac{p_B}{\gcd(p_H,p_B)}$ balls. ■

This leads us to the main result.

Corollary 8 A true (that is, prime) cascade with h hands and b balls exists if and only if h and b are relatively prime.

References

[1] P. J. Beek and A. Lewbel. The science of juggling. *Scientific American*, November 1995.

[2] S. Mauw. *PSF - A process specification formalism*. PhD thesis, Programming Research Group, University of Amsterdam, 1991.

Let's Accept Rejects, But Only After Repairs

Jan van Eijck

Abstract

In the quest for a useful syntax for language definition formalisms — the SDF part of ASD+SDF of Klint and his co-workers — Visser has proposed to extend context free rules with reject productions. We propose to modify the definition to ensure modularity of reject grammars. Next, attention is drawn to the close link between reject grammars (under the modified definition), indexed least fixpoint grammars (Rounds) and simple literal movement grammars (Groenink). This link indicates that reject grammars are closely related to the 'mildly context-sensitive' grammar formalisms that have been developed for use in natural language analysis.

1 Reject Grammars

To increase the versatility of the ASD+SDF meta-environment for specification and syntax definition created by Paul Klint and his co-workers [6], Visser [9] proposes to add reject rules $A \rightarrow_r A_1 \cdots A_n$ to CF grammars. The recipe for handling these reject rules is, roughly, the following: if $A \rightarrow_r \alpha$ is a rule of the grammar, and if $\alpha \Rightarrow^* w$, then $A \not\Rightarrow^* w$.

The reason why this is only a rough indication of what is meant is the effect of the rejects rules on the notion of derivability. The familiar $\Rightarrow^*$, the reflexive transitive closure of one-step derivability, does no longer determine membership of the yield of a string of symbols of the grammar.

To precisely define the language generated by a CF reject grammar, a CF grammar with the peculiarity that some of its are marked as reject rules (indicated by $A \rightarrow_r \alpha$), we have to say the following. The derivable strings of a CF^r grammar G are defined as follows:

1. Every string of symbols derives itself.
2. If $A \rightarrow \alpha$ in G, and if α derives β, while for all reject rules $A \rightarrow_r \gamma$ in G it holds that γ does NOT derive β, then A derives β.
3. If $A \rightarrow_r \alpha$ in G, and α derives β, then A does NOT derive β.

Consider example grammar (1).

$$
(1)\quad \begin{array}{lll} S & \rightarrow & \epsilon \mid AS \\ A & \rightarrow & a \mid b \\ S & \rightarrow_r & a \end{array}
$$

According to the recipe given above, grammar (1) generates the set of all strings over a, b except for the strings ending with a. In this particular case, the generated language is regular, but in general the generated language need not even be CF.

It should be noted that some care is needed to ensure that the format is well-defined. Observe that the presence of a reject rule of the form $A \rightarrow_r A$ will immediately lead to a contradiction: according to clause 1, A derives A, but then, according to clause 3, A does not derive A. We can remedy this by restricting clause 1 to strings of terminals, but it turns out that this is not enough, witness the following example.

$$
\text{(2)} \quad \begin{array}{lcl} S & \rightarrow & a \\ A & \rightarrow & S \\ S & \rightarrow_r & A \end{array}
$$

Does example grammar (2) derive a? Suppose the answer is yes. i.e., suppose a is derivable from S. Then $A \rightarrow S$ and S derives a, and moreover, there are no reject rules with A as LHS. Therefore, according to clause 2 of the definition, A derives a. Since $S \rightarrow_r A$ is a reject rule of the grammar, according to clause 3, S does not derive a, and contradiction.

Suppose, on the other hand, that S does not derive a. Since $A \rightarrow S$ is the only rule with LHS A, it follows that A does not derive a either. According to clause 1 of the definition, a derives a. Since $S \rightarrow a$ in G, we can apply clause 2 to conclude that S derives a, and contradiction again.

We get a contradiction either way, even if we restrict clause 1 of the definition to terminal strings. In this example case, the problem is caused by the loop $S \rightarrow_r A \rightarrow S$. Visser proposes to rule out such loops in the formalism, by stipulating that a reject grammar should have no loops through a reject production. If I understand this correctly, Visser wants to rule out grammars where A derives B, $B \rightarrow_r \beta$ and β derives A. This seems to solve the problem, but at the (considerable) cost of destroying the modularity of the formalism. CF grammars are modular in the sense that adding extra rules to a CF grammar will always give a CF grammar. CF reject grammars satisfying the loop constraint are not, for the addition of extra rules may create a vicious loop and thereby destroy the constraint.

2 A Closer Look at the Definition

Let us see if we can find an alternative definition that preserves modularity. For that, we need to take a closer look at the definition of derivation paths for reject grammars. The relations $\Rightarrow$ (for one step derivation), $\Rightarrow_r$ (for one step rejection), and $\Rightarrow^\bullet$ (for being on an accept path) are defined as follows:

Definitions of $\Rightarrow, \Rightarrow_r$

$$
\frac{A \rightarrow \alpha}{\beta A \gamma \Rightarrow \beta \alpha \gamma} \qquad\qquad \frac{A \rightarrow_r \alpha}{\beta A \gamma \Rightarrow_r \beta \alpha \gamma}
$$

Definition of $\Rightarrow^\bullet$ The following rules cover the three clauses from the definition of 'derivation' above. To be on the safe side, we restrict 1. to lists of terminals (strings from V_T^*).

1.

$$\frac{}{\alpha \Rightarrow^\bullet \alpha}\ \alpha \in V_T^*$$

2.

$$\frac{\alpha \Rightarrow \beta \qquad \beta \Rightarrow^\bullet \gamma \qquad \forall\delta:\ \text{if } \alpha \Rightarrow_r \delta \text{ then } \delta \not\Rightarrow^\bullet \gamma}{\alpha \Rightarrow^\bullet \gamma}$$

3.

$$\frac{\alpha \Rightarrow_r \beta \qquad \beta \Rightarrow^\bullet \gamma}{\alpha \not\Rightarrow^\bullet \gamma}$$

To be able to establish negative conclusions, we have to add:

4.

$$\frac{\forall\beta:\ \text{if } \alpha \Rightarrow \beta \text{ then } \beta \not\Rightarrow^\bullet \gamma}{\alpha \not\Rightarrow^\bullet \gamma}$$

The following expresses that we have derived a contradiction:

5.

$$\frac{\alpha \Rightarrow^\bullet \beta \qquad \alpha \not\Rightarrow^\bullet \beta}{\bot}$$

We can now formalize our reasoning about grammar (2) as follows:

$$\frac{\dfrac{S \Rightarrow_r A \qquad \dfrac{A \Rightarrow S \qquad S \Rightarrow^\bullet a \qquad \forall\beta : A \not\Rightarrow_r \beta}{A \Rightarrow^\bullet a}}{S \not\Rightarrow^\bullet a} \qquad S \Rightarrow^\bullet a}{\bot}$$

$$\frac{\dfrac{S \Rightarrow A \qquad a \Rightarrow^\bullet a \qquad \dfrac{\dfrac{\dfrac{S \not\Rightarrow^\bullet a}{\forall\beta:\ \text{if } A \Rightarrow \beta \text{ then } \beta \not\Rightarrow^\bullet a}}{A \not\Rightarrow^\bullet a}}{\forall\beta:\ \text{if } S \Rightarrow_r \beta \text{ then } \beta \not\Rightarrow^\bullet a}}{S \Rightarrow^\bullet a} \qquad S \not\Rightarrow^\bullet a}{\bot}$$

Looking at the rules 1–4, it seems that 3 and 4 deserve closer scrutiny, for they yield negative conclusions. But rule 4 is harmless. It just expresses the fact that $\Rightarrow^\bullet$ is the *smallest* relation satisfying 2 and 3. It argues from a negative premise about $\Rightarrow^\bullet$ to a negative conclusion about $\Rightarrow^\bullet$, so it does not involve a polarity switch. However, the other rule with a negative conclusion is vicious, for rule 3 switches polarity between premise and conclusion. In the next section we will see that this may destroy the monotonicity of the yield function of a grammar.

3 Denotational Semantics for Grammar Rules

To see the effect of rule 3, we consider the behaviour of the parallel single step yield function $[G]$ associated with a grammar G. To define $[G]$, we proceed as follows. Let V_N be the non-terminals of G, and V_T the terminals. Then an assignment for G is a function $i : V_N \to \mathcal{P}(V_T^*)$. Let I be the set of assignments for G. Associate a language with every member of V_T, by mapping every terminal to the singleton containing that terminal: $G_t := \{t\}$. To determine G_A, for $A \in V_N$, we proceed as follows. Every assignment i can be extended in a unique way to a function $i' : (V_N \cup V_T)^* \to \mathcal{P}(V_T^*)$ by means of the stipulations:

- $i'(x) := i(x)$ if $x \in V_N$,
- $i'(x) := G_x$ if $x \in V_T$,
- $i'(x\alpha) := i'(x)i'(\alpha)$.

The final step in the definition uses language concatenation.

Now if G is a standard CF grammar, then one parallel rewriting step according to G can be viewed as a function $[G] : I \to I$, given by $[G](i) = j$, where j is the assignment given by:

$$j(A) \quad := \quad \bigcup\{i'(\alpha) \mid A \to \alpha \text{ a rule of } G\}$$

(3) $\quad S \quad \to \quad ab \mid aSb$

For example, if G is grammar (3), and i is the assignment which maps S to $\emptyset$, then $[G](i)$ is the assignment which maps S to $\{ab\}$.

The assignments have an obvious partial order on them: $i \sqsubseteq j$ iff for all $A \in V_N$, $i(A) \subseteq j(A)$. The minimal assignment in this ordering is the assignment that assigns $\emptyset$ to any non-terminal. Call this assignment $\perp$. If $[G]$ is monotonic, as it always is if G is a CF grammar, we can iterate the process of applying $[G]$, starting with $\perp$, and we are bound to arrive at a fixpoint: an assignment j with the property that $[G](j) = j$. It is easy to see that $[G]$ also is continuous, in the sense that it preserves limits of chains. Therefore the smallest fixpoint of $[G]$ equals $\bigsqcup_{k\in\mathbb{N}}\{[G]^k\perp\}$. If j is the smallest fixpoint of $[G]$ and S is the start symbol of G, then $j(S)$ is the language generated by G. (See Rounds [8] for more information on this perspective on grammar formalisms.)

In the example case of G equal to grammar (3), the fixpoint is only reached at infinity, and we have:

$$\begin{array}{lcl}
([G]\perp)(S) & = & \{ab\} \\
([G]^2\perp)(S) & = & \{ab, aabb\} \\
([G]^3\perp)(S) & = & \{ab, aabb, aaabbb\} \\
 & \vdots & \\
\bigsqcup_{k\in\mathbb{N}}\{([G]^k\perp)\}(S) & = & \{a^nb^n \mid 0 < n < \infty\}
\end{array}$$

Let us apply this to the case of reject grammars. Here we have to slightly redefine the operation $[G]$, in order to take the reject productions into account.

The function $[G] : I \to I$ is now given by $[G](i) = j$, where j is the assignment defined as:

$$
\begin{aligned}
j(A) \quad := \quad & \bigcup\{w \in i'(\alpha) \mid A \to \alpha \text{ a rule of } G, \\
& \qquad \text{and for all } A \to_r \beta \text{ rules of } G, w \notin i'(\beta)\}.
\end{aligned}
$$

If we take G to be grammar (1) we get the following:

$$
\begin{array}{lcllcl}
([G]\bot)(S) & = & \{\epsilon\} & ([G]\bot)(A) & = & \{a,b\} \\
([G]^2\bot)(S) & = & \{b,\epsilon\} & ([G]^2\bot)(A) & = & \{a,b\} \\
([G]^3\bot)(S) & = & \{ab,bb,b\epsilon\} & ([G]^3\bot)(A) & = & \{a,b\} \\
([G]^4\bot)(S) & = & \{aab,bbb,ab,bb,b,\epsilon\} & ([G]^4\bot)(A) & = & \{a,b\} \\
 & \vdots & & & \vdots & \\
\bigsqcup_{k\in\mathbb{N}}\{[G]^k\bot\}(S) & = & \{a,b\}^*b+\epsilon & \bigsqcup_{k\in\mathbb{N}}\{[G]^k\bot\}(A) & = & \{a,b\}
\end{array}
$$

Now let G be grammar (2). We get:

$$
\begin{array}{lcllcl}
([G]\bot)(S) & = & \{a\} & ([G]\bot)(A) & = & \emptyset \\
([G]^2\bot)(S) & = & \emptyset & ([G]^2\bot)(A) & = & \{a\} \\
([G]^3\bot)(S) & = & \{a\} & ([G]^3\bot)(A) & = & \emptyset \\
 & \vdots & & & \vdots &
\end{array}
$$

It turns out that the function $[G]$ for this grammar is not monotonic, so it does not have a fixpoint, so G does not generate a well-defined language.

4 A Monotonic Semantics for Reject Grammars

The trouble with the definition of $\Rightarrow_\bullet$ for reject grammars is that it involves reasoning from premises where $\Rightarrow_\bullet$ occurs negatively to a conclusion where it occurs positively, and *vice versa.* We can avoid this by switching to a simultaneous definition of two relations $\Rightarrow_a^\bullet$ ((for acceptance) and $\Rightarrow_r^\bullet$ (for rejection), while taking care that the definition does not involve polarity switches between premises and conclusion.

1'.

$$\frac{}{\alpha \Rightarrow_a^\bullet \alpha}\ \alpha \in V_T^*$$

2'.

$$\frac{\alpha \Rightarrow \beta \qquad \beta \Rightarrow_a^\bullet \gamma \qquad \forall\delta: \text{ if } \alpha \Rightarrow_r \delta \text{ then } \delta \Rightarrow_r^\bullet \gamma}{\alpha \Rightarrow_a^\bullet \gamma}$$

3'.

$$\frac{\alpha \Rightarrow_r \beta \qquad \beta \Rightarrow_a^\bullet \gamma}{\alpha \Rightarrow_r^\bullet \gamma}$$

4'.

$$\frac{\forall\beta: \text{ if } \alpha \Rightarrow \beta \text{ then } \beta \Rightarrow_r^\bullet \gamma}{\alpha \Rightarrow_r^\bullet \gamma}$$

No further principles for handling negation are necessary, since there are no polarity switches in the rules.

To define a denotational semantics for this format, let an assignment for grammar G with nonterminal set V_N be a pair of functions $i_a : V_N \to V_T^*$, $i_r : V_N \to V_T^*$. Extension to $i'_a : (V_N \cup V_T)^* \to \mathcal{P}(V_T^*)$, $i'_r : (V_N \cup V_T)^* \to \mathcal{P}(V_T^*)$ as before. Let $[G] : I^2 \to I^2$ now be given by $[G](i_a, i_r) = (j_a, j_r)$, where j_a, j_r are the assignments given by:

$$\begin{aligned} j_a(A) &:= \bigcup\{w \in i'_a(\alpha) \mid A \to \alpha \text{ a rule of } G, \\ &\qquad \text{and for all } A \to_r \beta \text{ rules of } G, w \in i'_r(\beta)\}, \\ j_r(A) &:= \bigcup\{w \in i'_a(\alpha) \mid A \to_r \alpha \text{ a rule of } G\} \\ &\qquad \cup \bigcap\{w \in i'_r(\beta) \mid A \to \beta \mid A \to \beta \text{ a rule of } G\}. \end{aligned}$$

Defining $\sqsubseteq$ on I^2 pointwise in terms of $\sqsubseteq$ on I, we see that $(\perp, \perp)$ is the bottom element of I^2 and that the function $[G] : I^2 \to I^2$ is monotonic and continuous. This means $[G]$ has a fixpoint for every reject grammar G, and we can define the language generated by G as $i_a(S)$, where (i_a, i_r) is the least fixpoint of G. We may conclude that our alternative definition of the language works for any reject grammar. What this means is that we have succeeded in giving a modular definition of reject grammars. Moreover, it is not difficult to check that the new definition coincides with the original definition in all non-vicious cases.

5 A Logic Programming Perspective

A slightly different way to look at the rules of a context free grammar is as *constraints* on the interpretation of the grammar, where the interpretation of a (terminal or non-terminal) A is the language G_A. The rule $A \to A_1 \cdots A_n$ gives rise to the constraint $G_A \supseteq G_{A_1} \ldots G_{A_n}$. The interpretation of the grammar G is the 'smallest' assignment to the non-terminals for which all the constraints are satisfied.

To specify the constraints imposed by CF rules, we can use a translation into first order logic, by considering every non-terminal symbol A as a one-placed predicate over strings, and every terminal symbol a as a string. Then the rule $S \to ab$ gives rise to the first order formula $S(ab)$, and the rule $S \to aSb$ to the first order formula $\forall x(S(x) \to S(axb))$ (read this as: 'if x is a string of type S, then axb is a string of type S as well'), and so on.

Leaving out the universal quantifiers and writing $A \to B$ as B :− A, we arrive at the following logic programming style notation for grammar (3):

$$\begin{aligned} &S(axb) \text{ :- } S(x), \\ &S(ab). \end{aligned}$$

Thus, CF rules correspond to definite clauses of a logic programming language with built-in concatenation. Now, how to extend this perspective to reject grammars? According to the original definition, we should translate every accept rule $A \to A_1 \cdots A_n$ as:

$$A(x_1 \cdots x_n) \text{ :- } A_1(x_1), \ldots, A_n(x_n), \neg A^r(x_1 \cdots x_n),$$

and every reject rule $A \to_r B_1 \cdots B_m$ as:

$$A^r(x_1 \cdots x_m) \text{ :- } B_1(x_1), \ldots, B_m(x_m).$$

The logic programming interpretation of this translation will run into trouble in all cases where the negation as failure computation for negated atoms leads to floundering. (See Apt and Bol [1] or Doets [2].) Presumably, these are precisely the cases where the reject language definition gives rise to a vicious circle.

According to the modified definition, we should proceed as follows. First, for every terminal a of the grammar, put the following clauses:

$$\begin{array}{lll} a(x) & \text{:-} & x = a, \\ a^r(x) & \text{:-} & x \neq a. \end{array}$$

Next, add the following epsilon rules:

$$\begin{array}{lll} \epsilon(x) & \text{:-} & x = \epsilon, \\ \epsilon^r(x) & \text{:-} & x \neq \epsilon. \end{array}$$

If A is a non-terminal, assume that the list of accept rules for A looks like this:

$$\begin{array}{lll} A & \to & A_{11} \cdots A_{1n_1} \\ A & \to & A_{21} \cdots A_{2n_2} \\ & \vdots & \\ A & \to & A_{p1} \cdots A_{pn_p}. \end{array}$$

Pick new predicate symbols $A_1^r, \ldots, A_p^r$, and add the following clauses:

$$\begin{array}{rll} A_1^r(x_1 \cdots x_{n_1}) & \text{:-} & A_{11}^r(x_1), \ldots, A_{1n_1}^r(x_{n_1}), \\ A_2^r(x_1 \cdots x_{n_2}) & \text{:-} & A_{21}^r(x_1), \ldots, A_{2n_1}^r(x_{n_2}), \\ & \vdots & \\ A_p^r(x_1 \cdots x_{n_p}) & \text{:-} & A_{p1}^r(x_1), \ldots, A_{pn_1}^r(x_{n_p}), \\ A^r(w) & \text{:-} & A_1^r(w), \ldots, A_p^r(w). \end{array}$$

In case there are no reject rules for A, translate every accept rule $A \to A_1 \cdots A_n$ as:

$$A(x_1 \cdots x_n) \text{ :- } A_1(x_1), \ldots, A_n(x_n).$$

If there are reject rules for A, we may assume the set of reject rules for A to look like this:

$$\begin{array}{lll} A & \to_r & B_{11} \cdots B_{1m_1} \\ A & \to_r & B_{21} \cdots B_{2m_2} \\ & \vdots & \\ A & \to_r & B_{k1} \cdots B_{km_k}, \end{array}$$

In that case, first add the following clauses:

$$
\begin{array}{lcl}
A^r(x_1 \cdots x_{m_1}) & :- & B_{11}(x_1), \ldots, B_{1m_1}(x_{m_1}) \\
A^r(x_1 \cdots x_{m_2}) & :- & B_{21}(x_1), \ldots, B_{2m_2}(x_{m_2}) \\
& \vdots & \\
A^r(x_1 \cdots x_{m_k}) & :- & B_{k1}(x_1), \ldots, B_{km_k}(x_{m_k}).
\end{array}
$$

Next, pick new predicate names $B_1^r, \ldots, B_k^r$, and add:

$$
\begin{array}{lcl}
B_1^r(x_1 \cdots x_{m_1}) & :- & B_{11}^r(x_1), \ldots, B_{1m_1}^r(x_{m_1}) \\
B_2^r(x_1 \cdots x_{m_2}) & :- & B_{21}^r(x_1), \ldots, B_{2m_2}^r(x_{m_2}) \\
& \vdots & \\
B_k^r(x_1 \cdots x_{m_k}) & :- & B_{k1}^r(x_1), \ldots, B_{km_1}^r(x_{m_k})
\end{array}
$$

Finally, translate every accept rule $A \to A_1 \cdots A_n$ as:

$$A(x_1 \cdots x_n) \; :- \; A_1(x_1), \ldots, A_n(x_n), B_1^r(x_1 \cdots x_n), \ldots, B_k^r(x_1 \cdots x_n).$$

Let us check how this recipe works out for example grammars (1) and (2). The logic program corresponding to grammar (1), after removal of some redundancies:

$$
\begin{array}{rcl}
S(\epsilon) & :- & a^r(\epsilon). \\
S(xy) & :- & A(x), S(y), a^r(xy). \\
S^r(a). & & \\
a^r(x) & :- & x \neq a. \\
A(a). & & \\
A(b). & &
\end{array}
$$

The logic program corresponding to grammar (2):

$$
\begin{array}{rcl}
S(x) & :- & a(x), A^r(x). \\
S^r(x) & :- & a^r(x). \\
A(x) & :- & S(x). \\
a(x) & :- & x = a. \\
a^r(x) & :- & x \neq a.
\end{array}
$$

6 Rejects Versus Simple Movement

The clauses resulting from the translation instruction of the last section satisfy some special properties:

- Every variable occurring in the RHS of a clause occurs in the LHS,
- No variable occurs more than once in the LHS of a clause.

Moreover, the following property is easy to achieve by adding extra predicate symbols:

- Arguments in the RHSs of clauses are always single variables.

To impose this third constraint, the only thing we have to do is replace every clause of the form

$$A(x_1 \cdots x_n) \ :- \ A_1(x_1), \ldots, A_n(x_n), B_1^r(x_1 \cdots x_n), \ldots, B_k^r(x_1 \cdots x_n)$$

by a suitable clause set, employing new predicate symbols C_1, C_2:

$$\begin{array}{rcl} C_1(x_1 \cdots x_n) & :- & A_1(x_1), \ldots, A_n(x_n), \\ C_2(w) & :- & B_1^r(w), \ldots, B_k^r(w), \\ A(w) & :- & C_1(w), C_2(w). \end{array}$$

In Groenink [5] a grammar satisfying the three just mentioned constraints is called a *simple literal movement grammar* (SLMG). Groenink proves that the simple literal movement grammars correspond precisely with the index least fixpoint grammars (ILFPs) of Rounds [8], which in turn are precisely the grammars that can be parsed in polynomial time (in $O(n^c)$, where n is the size of the input string, and c is some constant).

Now it turns out that our definite clause translations of reject grammars can be massaged to satisfy a fourth constraint:

- all predicates are unary.

The only predicates that do not satisfy this constraint are the predicates for string equality and string inequality employed in the clauses for the terminal predicates $a, a^r, \epsilon, \epsilon^r$. These can be replaced by the following:

$$\begin{array}{rl} a(a), & \\ a^r(\epsilon), & \\ a^r(bx) & \text{for all } b \in V_T \text{ with } b \neq a, \\ \epsilon(\epsilon), & \\ \epsilon^r(ax) & \text{for all } b \in V_T. \end{array}$$

A final simplification is to replace every clause of the form

$$A(x_1 \cdots x_n) \ :- \ A_1(x_1), \ldots, A_n(x_n)$$

by a set of clauses employing fresh predicate letters $A'_2, \ldots, A'_{n-1}$, as follows:

$$\begin{array}{rcl} A(xy) & :- & A_1(x), A'_2(y), \\ A'_2(xy) & :- & A_2(x), A'_3(y), \\ & \vdots & \\ A'_{n-1}(xy) & :- & A_{n-1}(x), A_n(y), \end{array}$$

and every clause of the form

$$A(w) \ :- \ B_1(w), \ldots, B_m(w)$$

by a set of clauses

$$\begin{array}{rcl} A(w) & :- & B_1(w), B'_2(w), \\ B'_2(w) & :- & B_2(w), B'_3(w), \\ & \vdots & \\ B'_{m-1}(w) & :- & B_{m-1}(w), B_m(w). \end{array}$$

As Groenink remarks, it is not difficult to translate SLMG clauses with monadic predicates into CF^r format, the key being that reject rules can be used to define intersections (Groenink [4]). Indeed, all clauses except for the intersection rules translate into CF rules. An intersection rule $S(w)\ :-\ A(w), B(w)$ translates into the following reject grammar (assuming a terminal set $\{a, b\}$, and taking care to employ fresh rewrite symbols where appropriate):

$$(4) \quad \begin{array}{rcl} S & \rightarrow & A \mid B \\ S & \rightarrow_r & C \\ C & \rightarrow & A' \\ C & \rightarrow & B' \\ A' & \rightarrow_r & A \\ B' & \rightarrow_r & B \\ A' & \rightarrow & \epsilon \mid aA' \mid bA' \\ B' & \rightarrow & \epsilon \mid aB' \mid bB' \end{array}$$

Here A' generates the complement of L_A, B' the complement of L_B. C generates the union of those complements, and S, finally, the complement of this union.

It should be noted that the correspondence between reject grammars and unary SLMGs only holds under the modified definition of reject grammars proposed above.

The correspondence between reject grammars and unary simple literal movement grammars shows that reject languages are among the tractable languages, i.e., the languages that can be parsed in polynomial time. Next on the agenda should be the attempt to put a reasonable bound on the constant c in $O(n^c)$ (see Groenink [5] for some hints), the definition of suitable partitions of the class of CF^r languages, and an attempt to get still more precise bounds for CF^r grammars satisfying appropriate extra constraints. For instance, if a reject grammar has the property that its reject rules cannot occur nested, what does this tell us about the value of c in $O(n^c)$? And what if the grammar is such that its reject chains have a bound k?

An example of a reject grammar generating the non-CF language $a^n b^n c^n$ was given in Van Eijck [3] (see Visser [9]). The example can easily be extended to show that all n-counting languages (languages of the form $a^n b^n \cdots k^n$, for an arbitrary list of terminals $a, \ldots, k$) are in CF^r.

An example of a very simple language that can be generated by a binary SLMG is a^{2^n}:

$$\begin{aligned} S(xy) \quad &:- \quad S(x),\ x = y, \\ S(a). & \end{aligned}$$

This uses the binary equality predicate. It seems a reasonable conjecture that this language cannot be defined without the equality predicate (or another binary predicate that serves essentially the same purpose), but I have no formal proof of this. To prove such a negative claim one needs a pumping lemma for reject grammars (or equivalently, for unary SLMGs), and such a lemma is still lacking. So let's put that also on our research agenda.

At present, to my knowledge, we do not have examples of languages that are in binary SLML but not in unary SLML (or CF^r), nor do we have examples of languages in CS, but not in CF^r. A CS grammar for a^{2^n} is part of the formal language folklore (see Révész [7], exercise 1.5), but we have no proof that $a^{2^n} \notin CF^r$. The difficulty here is that it is not completely clear how the Chomsky hierarchy relates to the complexity hierarchy proposed in Rounds [8] and Groenink [5].

The following closure properties of CF^r languages are easy to establish:

closure properties of CF^r languages	
closed under union	yes
closed under concatenation	yes
closed under Kleene-star	yes
closed under $^+$	yes
closed under homomorphic image	no
closed under reversal	yes
closed under intersection with CF language	yes
closed under intersection	yes
closed under complement	yes.

The reason for the single 'no' in the table is the classical result that a formalism that can generate homomorphic images of intersections of CF languages can generate every r.e. language.

7 Reject Rules for Natural Language Analysis

Groenink shows in his thesis that SLMG (but with binary predicates) is a suitable formalism for tackling some hairy constructs in natural language syntax, such as cross-serial dependencies in Dutch. It remains an open question whether such natural language constructs can also be tackled with reject rules, as proposed by Visser. To settle that question, we have to know whether the inclusion between reject languages and binary SLM languages is proper or not. My conjecture is that it is, and that it will turn out that reject rules are not powerful enough for handling the full complexity of natural language syntax.

Still, I hope to have illustrated that the two PhD theses resulting from the dual SION project Visser/Groenink, the first supervised by Paul Klint and the second by myself, are closely related. Paul and I were quite right, five years ago, when we told the Dutch computer science funding agency SION that parsing of natural languages and parsing of programming languages have a lot in common, and that money to explore those common aspects would be well spent. I am very glad we did get that grant.

References

[1] Krzysztof Apt and Roland Bol. Logic programming and negation: a survey. *Journal of Logic Programming*, 19-20:9–71, 1994.

[2] Kees Doets. *From Logic to Logic Programming.* MIT Press, 1994.

[3] Jan van Eijck. Email correspondence with Eelco Visser. CWI, July 1997.

[4] Annius Groenink. Stellingen. PhD thesis, 1997.

[5] Annius Groenink. *Surface Without Structure.* PhD thesis, Utrecht University, November 1997.

[6] Paul Klint and co workers. *The ASF+SDF Meta-environment User's Guide.* CWI, 1995.

[7] G. E. Révész. *Introduction to Formal Languages.* Dover, 1991. Reprint of 1983 publication by McGraw-Hill.

[8] Bill Rounds. LFP: A logic for linguistic descriptions and an analysis of its complexity. *Computational Linguistics*, 14(4):1–9, 1988.

[9] Eelco Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, September 1997.

Van Teus Hagen, Grubbenvorst, oktober 1997

Beste Paul,

Vijf en twintig jaar aan het instituut verbonden. Een instituut dat eerst de naam *Mathematick Centrum* droeg en later de naam CWI verkoos. Als een jubileum een 25-jarig huwelijk betreft, word je gevraagd om een A4-tje in te leveren. Bij een verbondenheid aan een belangrijk instituut heet dat, merk ik, een "Liber Amicorum". Welnu voor mij maakt het niet veel uit. Uit de vele evenementen en anekdotes zal ik er van elk één hier openbaren. Allereerst de toonzetting:

Onze eerste ontmoeting stamt van voor de tijd bij het Mathematick Centrum: als eerste jaars novieten bij de studenten vereniging Unitas. Voor jouw begon toen de triomf in oktober 1967 tijdens de novieten week als drummer bij een Beatle songs: "all you need is love". Dit lied luidde een tijdperk in welke blijkbaar nog steeds zo gedragen wordt zoals je die toen inzette.

Wat mij betreft begon onze periode bij het Mathematick Centrum eind 1974 met mijn start bij het graphics research team. Voor die tijd had het team een redelijk grote omvang: Paul ten Hagen, Han Noot, Pail Klint en Teus Hagen. Arthur Veen en Marleen Sint volgden kort erop ook als teamleden. Echter toen was de Internmediate Language for Pictures (een soort voorloper van PostScript) al gedefinieerd en je eerste wetenschappelijke presentatie voor groot publiek in Canada reeds een feit. Voor mij was deze taal-research niet het belangrijkste. Dat was jouw initiatief om RSX11-M, het operating systeem voor de PDP11/4, letterlijk in de prullenbak te gooien. De actie werd gevolgd door het opzetten van een onbekend systeem op die DEC machine: de Europese geboorte van UNIX. *Het was jij die Ken Thompson een brief schreef. En met de opmerking "aardig artikel van je in de ACM operating systems review 9(1975)5" (19 november 1976) Ken blijkbaar zo triggerde dat hij je prompt een RK05 cardridge toestuurde, compleet met het systeem en de broncode (sources heet dat nu). Geen AT&T licentie gedoe, geen vergoedingen, gewoon aub. Paul richtte zich echter meer op de taal en op de programmeer omgevingen en dat was toch wat vloeken in een Algol 68 school klimaat.* "Een bestaand iets weggooien en gewoon met veel risico iets nieuws beginnen" is naar mijn ervaring karakteristiek voor jouw. Het andere karakteristieke is dat je iemand weet te stimuleren met de uitstekende kreet "is dat niet iets voor jouw". Voor mij betekende dat UNIX en netwerking.

Een anekdote*: er wordt wat meer kleur in de brouwerij gebracht!*
Tussen het "schoolgebouw" van het Mathematisch Centrum en de Amstel Brouwerij was onder andere de graphics groep gevestigd in een soort houten keet. Wij beiden hadden een kamer bijna op het eind van de gang. Net als alle kamers toendertijd (eind zeventiger jaren) waren de kamers troosteloos van aanblik en uitzicht (een muur). Wij niet getreurd, tijdens het weekend hebben we gewoon even met de verfkwast gezwaaid. Het gevolg was een unieke kamer qua kleur: groen en oker geel was immers in de mode. We hebben toen niet de moed gehad om ook ons uitzicht (een muur) van kleur te voorzien. Tot de afbraak ij de Boerhavestraat is deze kamer de enige fleurige gebleven.
De uitspraak "*dat doen we even*" staat me nog steeds bij. Zelfs tot in je werk kunnen we het woord "gezelligheid" herkennen. Socialiteit staat hoog in het vaandel. Pragmatisch van aanpak. Recht door zee.

Dit zijn maar een paar voorbeelden uit een ver verleden die volgens mij karakteristiek zijn voor het latere tijdperk van jouw bij het CWI.

Teus Hagen

SEMANOL, BASIS/1-12 en ASF+SDF

Het is een even intrigerende als vage vermelding die ik aantref in een online geschiedenis van het Rome Air Development Center, een laboratorium van de Amerikaanse luchtmacht. Op de lijst van RADC activiteiten voor 1973, tussen de ontwikkeling van een autofocus camera en een systeem voor stemidentificatie, staat

> **SEMANOL Development** The Center developed a method of eliminating multiple interpretations of computer higher-order languages. The Semantics Oriented Language (SEMANOL) found several ambiguities within the JOVIAL programming language, and RADC developed plans to apply SEMANOL to COBOL and FORTRAN as well.

NASA's Technical Report Server bevat veel informatie over dit soort projecten en vindt dan ook onmiddellijk een hele reeks rapporten met veelzeggende titels:

1. SEMANOL (76) interpreter documentation, volume 4, Nov 01, 1977
2. SEMANOL (76) reference manual, volume 2, Nov 01, 1977
3. A SEMANOL (76) specification of minimal BASIC, volume 1, May 01, 1977
4. Improvements to SEMANOL, volume 1, Nov 01, 1977
5. SEMANOL (73) interpreter documentation, Jun 30, 1975
6. JOVIAL (J73) computer validator, Jun 01, 1981
7. Formal SEMANOL specification of ADA, Sep 01, 1980
8. A categorization and evaluation of formal and semi-formal definition techniques, Mar 01, 1978
9. Minimal BASIC SEMANOL (76) specification listing, volume 2, May 01, 1977
10. SEMANOL (76) specification of JOVIAL (J3), volume 3, Nov 01, 1977
11. Automated compiler test case generation, Feb 01, 1978
12. HOL investigation, Jan 01, 1979.

De abstracts zijn er ook. Frank C. Belz van TRW Defense and Space Systems Group, klaarblijkelijk een van de contractors, vat het *SEMANOL (76) reference manual* als volgt samen:

> SEMANOL(76) is a metaprogramming language designed specifically for use in writing formal, operational, specifications of the syntax and semantics of contemporary programming languages. The SEMANOL(76) reference manual provides a detailed description of the metaprogramming language. The context-free syntax is given in the SEMANOL(76) notation, while the context-sensitive constraints and semantics are given by prose text. This manual does not explain how SEMANOL(76) ought to be used when writing formal specifications.

En B.D. Guilmain zegt over de vergelijkende studie (8):

> Operational techniques for defining computer programming languages are examined; specifically, SEMANOL, the Vienna Definition Language (VDL), and BASIS/1-12. A survey of the operational methods is given, in which specific examples of SEMANOL, the VDL, and BASIS/1-12 are explained in detail. A cataloging criteria is established and evaluated. The cataloging criteria is then used to categorize and evaluate SEMANOL, the VDL, and BASIS/1-12. The SEMANOL technique was judged as the best technique followed by BASIS/1-12 and the VDL, in that order.

Ik heb nog nooit van BASIS/1-12 gehoord en NASA's Technical Report Server laat het deze keer afweten.

De volledige rapporten zijn bij NASA te bestellen, maar ze zijn duur. Op zoek naar toegankelijker publicaties stuit ik in *Acta Informatica* van 1976 op *SEMANOL (73)—A Metalanguage for Programming the Semantics of Programming Languages* van de hand van Eric R. Anderson (eveneens TRW), de eerder genoemde Frank Belz en Edward K. Blum (USC). Dit artikel neemt de laatste twijfel weg. SEMANOL was inderdaad bedoeld als praktisch hulpmiddel bij taalontwerp en in die zin een voorganger van ASF+SDF, niet alleen in de rol van formalisme voor taaldefinitie maar ook als werkend systeem en tot op zekere hoogte als generator. Talen als BASIC, JOVIAL en ADA werden in SEMANOL beschreven en getest. Het systeem draaide bij RADC onder MULTICS op de Honeywell 6180.

Welke invloed heeft SEMANOL gehad? Bekend is het systeem nooit geworden, dat is wel zeker. Semantiekdefinities in SEMANOL waren primair *programma's* zonder duidelijk mathematisch karakter. SEMANOL hoorde niet tot de mainstream van het semantiekonderzoek in die tijd, dat zich primair op VDL en denotationele semantiek richtte en een overwegend theoretisch karakter had. Er is een andere mogelijkheid. Verwijzing [17] van het *Acta Informatica* artikel is

> E.K. Blum. Formal semantics of programming languages, IRIA Seminar Reviews, May 1974.

Er was blijkbaar een uit 1974 daterende connectie Blum—INRIA (toen nog IRIA), waar in datzelfde jaar de ontwikkeling van de generieke programmeeromgeving MENTOR van start ging. Men moet dus bij INRIA op de hoogte geweest zijn van SEMANOL. Blijft de vraag welke invloed dit op MENTOR gehad heeft.

Er is nog een vraag. Uit de verwijzingen bij het *Acta Informatica* artikel blijkt dat het SEMANOL project rond 1969 begonnen moet zijn, maar wanneer is het gestopt en waarom? Het laatste bij NASA bekende rapport (6) dateert van juni 1981. Dat zal wel een goede indicatie geven van de einddatum van het project. Niet lang daarvoor was nog over ADA gerapporteerd (7):

> This report summarizes the performance and results of a contractual effort to develop a formal operational specification of the DoD common programming language ADA. The formalism used was that of the Semantics Oriented Language, SEMANOL. The design produced essentially covers the entire ADA language, ignoring only the low-level semantics of implementation dependencies. The SEMANOL system and its use in the specification of ADA language features of a type not previously addressed in SEMANOL specifications of other languages is provided. The report is concluded with a list of problems discovered, in the design of the ADA language, as a result of the formal specification.

Het klinkt interessant genoeg, daar kan het niet aan gelegen hebben, maar wat was dan de reden van beëindiging?

Ik geef een eerste versie van dit stukje aan Jaco de Bakker en hij blijkt zich SEMANOL uitstekend te herinneren. Zijn waarneming dat SEMANOL destijds buiten de mainstream van het semantiekonderzoek viel is in het voorgaande verwerkt. Hij duikt een tweede artikel van hetzelfde drietal auteurs op in E.J. Neuhold (ed.) *Formal Description of Programming Concepts* (North-Holland, 1978) en vindt als klap op de vuurpijl in het MC jaarverslag voor 1973 op p. 133 onder "voordrachten door buitenlandse bezoekers" de vermelding

```
prof.dr. E.K. Blum               - SEMANOL: A language to define
(University of Southern            the semantics of programming
California, Los Angeles, USA;      languages (12 april; AI).
tijd. IRIA, Parijs, Frankrijk)
```

Paul was toen net een half jaar op het MC, maar ik geloof niet dat hij die voordracht heeft bijgewoond. Noch Paul noch ik hadden tot voor kort van SEMANOL gehoord. Deze coïncidentie maakt echter eens te meer duidelijk dat de sterk ontwikkelde semantiekcultuur op het CWI in het algemeen gesproken bevorderlijk is geweest voor de ontwikkeling van een systeem als ASF+SDF. Even zeker is, dat het er zonder de brede softwarekennis en op praktische resultaten gerichte energie van Paul nooit van zou zijn gekomen.

3 october 1997 — Jan Heering

Kwaliteit van een hoogleraar
Praktijkervaringen met een kwaliteitsmodel

Paul Hendriks

Bij de bepaling van de kwaliteiten van hoogleraren wordt maar al te vaak uitsluitend aandacht besteed aan hun functioneren. Er wordt gekeken naar de output die zij opleveren in termen van het aantal publicaties in wetenschappelijk erkende tijdschriften. Als al in kwalitatieve zin gemeten wordt aan die output dan is dat meestentijds in de vorm van het aantal publicaties of het aantal bugs na oplevering. Kwaliteit van een medewerker behelst echter meer. Om een gewogen oordeel te kunnen vellen, moeten ook andere eigenschappen zoals de attractiviteit, koppelbaarheid en herstelbaarheid betrokken worden in de overwegingen.

Het ontbreken van een begrippenkader waarin de kwaliteiten van een hoogleraar kunnen worden uitgedrukt is funest. In discussies over diens kwaliteit ontstaan meningsverschillen doordat de deelnemers aan de discussie denken over hetzelfde begrip te spreken terwijl zij bij nadere bestudering toch werken met verschillende interpretaties.

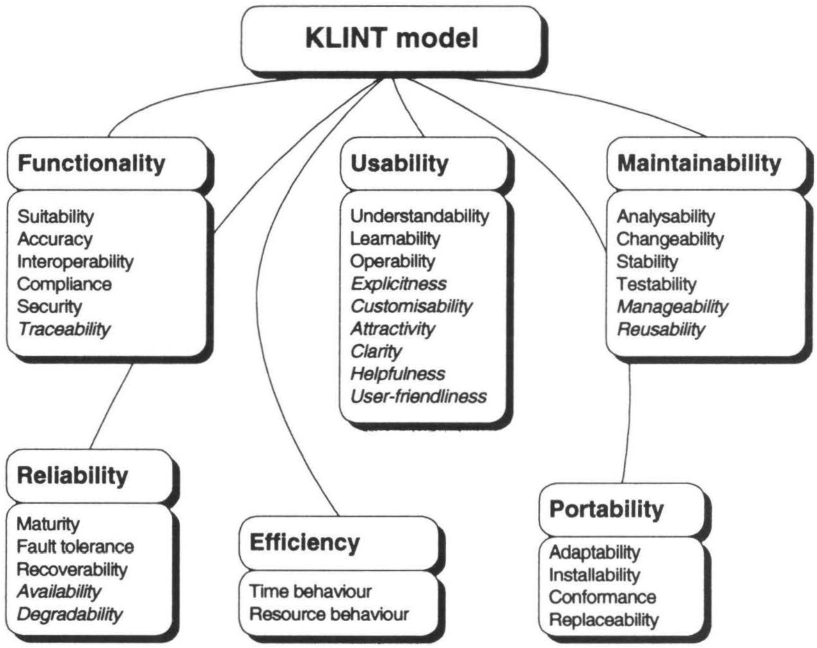
Vanuit de duidelijk geconstateerde behoefte aan een begrippenkader is een praktisch experiment uitgevoerd waarbij een van de kwaliteitsmodellen die is ontwikkeld voor softwareprodukten is toegepast op een van Neerlands hoogleraren die reeds 25 jaar werkzaam is en dus uitermate geschikt is om als proefkonijn in dit experiment te dienen.

Deze eerste praktijkervaring met het kwaliteitsmodel wijst uit dat het zeer goed bruikbaar is. Het model bevat definities van 32 kwaliteitseigenschappen die gebruikt kunnen worden ter specificatie van de eisen die men aan een hoogleraar stelt en ter validatie of aan die eisen wordt voldaan.

In deze bijdrage worden in het kort het gebruikte kwaliteitsmodel en de praktijkervaringen met het model beschreven. De praktijkervaringen moeten de lezer ertoe aanzetten om zelf aan de hand van het model een kwalitatief oordeel te vellen.

Het "KLINT Model"

Het gebruikte model voor specificatie en validatie van kwaliteitseigenschappen van een hoogleraar bestaat uit 32 eigenschappen, die geclassificeerd zijn in zes hoofdeigenschappen. Het model (zie figuur 1) is gebaseerd op de 21 eigenschappen uit de ISO-standaard ISO 9126 [ISO 91] aangevuld met de resultaten van het QUINT-project zoals beschreven in [Zeist et al 96].



Figuur 1 Overzicht van het "KLINT-Model".

Functionality

In het experiment met het KLINT-model blijkt dat de **suitability** wisselvallig beoordeeld wordt. Alhoewel Paul wel eens in een kolbertje gesignaleerd wordt, kan dit toch niet de formele kwalificatie van een "suit" verkrijgen. Als hij zich in een dergelijke uitmonstering vertoont lijkt het alsof hij zich niet helemaal voor de volle honderd procent op zijn gemak voelt.

Daar staat echter tegenover dat Paul uitermate geschikt is als hoogleraar. Hij heeft als promotor reeds menig doctor weten af te leveren. Zijn publicaties in boekvorm of in erkende wetenschappelijke tijdschriften — en ook bladen van minder allooi — laten een duidelijke staat van dienst zien waarop menig collega trots zou zijn. Presentaties en colleges van Paul zijn boeiende vertoningen waar hij gloedvol zijn betoog houdt en zijn publiek bij de les weet te houden.

Met betrekking tot **accuracy** mag gemeld worden dat Paul steeds op accurate wijze iemand op diens fouten weet te wijzen. Ik weet nog hoe mijn eerste poging om een wetenschappelijk rapport in het Engels te produceren gecorrigeerd van Paul terugkwam waarbij de complete pagina zwart zag van de potloodaanwijzingen. Daarbij mag ik nog van geluk spreken dat hij geen rood potlood gebruikt voor zijn verbeteringen. Achteraf gezien blijkt dat ik op die manier ontzettend veel van hem geleerd heb.

Wat betreft **interoperability** (koppelbaarheid) mag geconstateerd worden dat dit bij Paul wel snor zit getuige het feit dat hij al geruime tijd zijn baan bij het CWI weet te combineren met zijn hoogleraarschap aan de UvA. Ook weet hij altijd weer zijn grote variëteit aan contacten met vogels van divers pluimage op een positieve manier te benutten. Hij weet allerlei lijntjes uit te zetten en die daarna te koppelen tot boeiende projecten.

Wat kan gezegd worden over Paul's **compliance** (volgzaamheid)? Enerzijds zou men kunnen zeggen dat een 25-jarig dienstverband toch moet wijzen op een uitermate volgzaam karakter. Anderzijds bestaat het gevoel dat er toch een kleine rebel in hem schuilt. Hij heeft zich wel eens laten ontvallen dat hij graag een eigen bedrijf zou willen starten. Is het zijn volgzaamheid die hem (tot nu toe) daarvan weerhouden heeft?

Paul's **security** is een moeilijk te beoordelen eigenschap. In de dagelijkse omgang is hij altijd heel joviaal en werkt dit mee aan zijn open karakter. Toch is het nog maar de vraag of hij altijd het achterste van zijn tong laat zien.

Paul is soms onnavolgbaar waardoor zijn score ten aanzien van **traceability** negatief beïnvloed wordt.

Efficiency

Paul's effectieve werkwijze zorgt ervoor dat hij goed scoort op **time behaviour**. Anderzijds moet gemeld worden dat hij zelden voor negenen gesignaleerd is (maar welke CWI-(st)er is dat wel?). Het rondje van de koffiejuffrouw was en is voor velen het startsignaal van de werkdag.

Zoals de meesten zullen weten is Paul al sinds jeugdige leeftijd vegetariër. Als gevolg daarvan is zijn **resource behaviour** ten opzichte van vlees niet te evenaren. Toch doet zijn Bourgondische uitstraling vermoeden dat hij dit op de een of andere manier weet te compenseren.

Reliability

Na zoveel jaren trouwe dienst zou men toch wel mogen verwachten dat Paul een zeker volwassenheidsniveau (**maturity**) heeft bereikt. Zijn kinderen weten hem echter jeugdig te houden.

Paul is absoluut niet tolerant ten opzicht van fouten (**fault tolerance**). Zijn jarenlange occupatie met formele specificatieformalismen hebben hem getraind in het speuren en verbeteren van fouten. Zijn eigen geschriften en die van anderen die hij reviewt dienen verschoond te blijven van elke fout. Hij is zeer zeker foutgevoelig.

Omdat bij het onderzoeksteam geen officiële gegevens bekend zijn over Paul's gedrag na uitval als gevolg van dronkenschap, kan door het team geen uitspraak gedaan worden over diens **recoverability**. Tot nu toe zijn nog geen kwalijke gevolgen geconstateerd.

De beschikbaarheid (**availability**) van Paul is natuurlijk uitermate slecht. Zoals het een goed hoogleraar betaamt is hij hartstikke druk met van alles en nog wat. Daardoor is het bijzonder lastig om hem spontaan te pakken te krijgen. Even bij zijn kamer langslopen en kijken of hij er is, loopt meestal uit op een desillusie. Als men echter een afspraak met hem maakt neemt hij de tijd voor je en is hij toch weer beschikbaar.

De eigenschap **degradability** heeft betrekking op de mate waarin een hoogleraar in staat is om na uitval zijn essentiële functionaliteit weer uit te voeren. Een kleine verkoudheid of ander klein ongemak kan Paul niet van zijn werkzaamheden houden. Als hij al geveld wordt door een of ander virus (met kinderen als virusoverbrengers in huis wil zoiets wel eens gebeuren), gaat hij niet uitgebreid uitzieken. Zodra het ook maar enigszins kan zit hij weer in de boeken of achter zijn computer. Kortom, een pluim voor "degradability".

Maintainability

Hoe eenvoudig is het om bij Paul fouten te vinden en te repareren (**analysability**)? Het is bijzonder moeilijk om fouten te ontdekken bij iemand die zo degelijk werk levert als Paul. Er is simpelweg te weinig ervaringsmateriaal voor een gefundeerd oordeel.

Ten aanzien van **changeability** valt niets zinvols op te merken. Bij "adaptability" (zie aldaar) wordt al ingegaan op het aanpassingsvermogen van Paul. Dat is al uitstekend zonder dat daarvoor grote veranderingen in de structuur of programmering noodzakelijk zijn.

Paul is niet snel uit zijn evenwicht te brengen. Zowel fysiek als psychisch scoort hij goed ten aanzien van **stability**.

Met betrekking tot **testability** moet een pleidooi gehouden worden voor meer tests vóórdat een hoogleraar in de praktijk aan het werk wordt gezet. Net als bij software is het uitermate gebruikelijk om bij hoogleraren het testen uit te stellen en de fouten pas te detecteren in de veldtest. Dit kan niet! Men mag de gebruikers niet confronteren met kwalitatief niet getest materiaal. Ook een hoogleraar dient te worden onderworpen aan een uitvoerig aantal tests waaronder uiteraard een uitgebreide stress-test.

Paul is een prima manager. Hij leidt zijn troepen met enthousiasme en hij beheerst zijn vak als hoogleraar als geen ander. Dat zou het vermoeden kunnen wekken dat hij uitermate goed zou scoren op **manageability**. Edoch, is Paul zelf eigenlijk wel beheersbaar en "manage"-baar?

Welke delen van Paul zouden voor hergebruik (**reusability**) in aanmerking komen? Het lijkt er niet op dat hij in enige vorm herbruikbaar is.

Usability

Paul is altijd heel duidelijk (**understandability**) en expliciet (**explicitness**) in zijn uitspraken. Hij heeft een duidelijke mening over allerlei onderwerpen en zal die ook expliciet naar voren brengen.

Paul is iemand die altijd uitermate leergierig is geweest en op de hoogte wilde zijn van de nieuwste ontwikkelingen. Hij is een van de eersten in Nederland die met Unix aan de slag ging. Voor een hoogleraar is leergierigheid uiteraard een vanzelfsprekende eigenschap. Maar zegt dat nu ook wel iets over de leerbaarheid (**learnability**)? Hoe snel is zijn medemens in staat om hem te leren kennen?

De eigenschap **operability** heeft betrekking op de hoeveelheid moeite die het kost om de hoogleraar operationeel te houden. Als de salariskosten van de afgelopen 25 jaar bij elkaar geteld worden, komt daar natuurlijk een aanzienlijke kostenpost uit. Maar daar tegenover staan de baten in termen van gepromoveerden, onderzoeksresultaten, opgebouwde kennis, projecten met andere wetenschappelijke instituten en toepassingen door praktisch onderzoek in samenwerking met bedrijven.

Is Paul aanpasbaar (**customisability**)? Een van de meest letterlijke demonstraties van zijn vermogen om zich aan te passen aan zijn omgeving liet hij zien tijdens een wadlooptocht die door de personeelsvereniging van het CWI georganiseerd was. Vlak voor het einde van de tocht maakte Paul een kleine struikeling zodat hij na een duik in de modder volledig was aangepast aan de omgeving.

Hij doet altijd zijn uiterste best om zijn ideeën zo eenvoudig als mogelijk uit te leggen, Daarbij komen zijn didactische kwaliteiten hem zeer zeker van pas. Als gevolg daarvan mag zijn score op de eigenschap **clarity** meer dan bevredigend genoemd worden.

Om een oordeel te kunnen vellen over Paul's **attractivity** moeten we afgaan op het oordeel van de expert bij uitstek, zijn vrouw en vriendin Anneke. Gezien de bestendigheid van hun relatie moeten we ervan uitgaan dat haar oordeel positief uitvalt.

Paul is steeds hulpvaardig (**helpfulness**) als het gaat om het oplossen van technologische problemen van anderen. Hij is uitermate gebruiksvriendelijk (**user-friendliness**). Ik heb nog nooit een onvertogen woord uit zijn mond vernomen.

Portability

Paul's aanpassingsvermogen (**adaptability**) is fantastisch. Hij weet zich snel aan te passen aan omstandigheden en kan snel reageren op de meest onverwachte ontwikkelingen. Ik kan me nog het bezoek aan een conferentie met tentoonstelling in Brussel herinneren waar een demonstratie gegeven moest worden van GIPE: Generation of Interactive Programming Environments; een van de vele Europese projecten die Paul heeft weten te organiseren. Er was door de Franse partners gezorgd voor eigen machines zodat er niets mis kon gaan. De werkelijkheid bleek anders. Bij installatie bleek een en ander niet op elkaar te kunnen worden aangesloten. Vlak voor de opening van de tentoonstelling lag Paul met een soldeerapparaat onder de tafels om zo alsnog de gewenste verbinding tot stand te brengen. Als dat geen aanpassingsvermogen vergt?

Hoe test je de installeerbaarheid (**installability**) van iemand die al 25 jaar op dezelfde werkplek zit? De enige mogelijkheid die daartoe overblijft is om na te gaan hoe snel hij zich op een hotelkamer thuis voelt bij een van die vele buitenlandse reizen die hem als hoogleraar naar de meest verschillende oorden op deze aardbol brengt. Omdat Paul nog niet op enige vorm van heimwee betrapt is, nemen we voor het gemak maar even aan dat zijn installeerbaarheid in orde is.

De waardering voor **conformance** bleek niet eenvoudig vast te stellen. In hoeverre conformeert hij zich aan standaards? Bij een promotie doet hij naar hartelust mee aan het voorgeschreven protocol. Toch lijkt Paul in de standaarduitrusting van een hoogleraar enerzijds te winnen aan waardigheid, maar blijft er toch een restantje joligheid zichtbaar. Is het drumstel dat ergens nog op zolder staat een overblijfsel van zijn non-conformisme?

Paul's **replaceability** moeten we helaas uitermate negatief beoordelen. Hij is en blijft een unieke persoonlijkheid met een eigen karakter en invulling van zijn werkzaamheden. Hij is gewoonweg onvervangbaar.

Samenvattend

Het gebruik van het KLINT-model in de beoordeling van de kwaliteiten van de hoogleraar Paul Klint laat duidelijk zien dat hij fantastisch scoort. Er is gelukkig ook nog ruimte voor verbetering en bij deze wil ik hem toewensen dat hij nog vele jaren in goede gezondheid zal mogen blijven functioneren.

Voor de ontwerpers van het KLINT-model is het duidelijk dat de behandelde kwaliteitseigenschappen nog niet volledig bepalend zijn voor de "kwaliteit van leven". Het acroniem KLINT staat voor alles wat men iemand zou willen toewensen:

- Kwaliteit,
- Liefde en Gezondheid,
- Intelligentie,
- Natuur en Cultuur,
- Toekomst!

Paul, bedankt voor alles wat je in de afgelopen 25 jaar gedaan hebt en in het bijzonder voor de periode waarin ik deel mocht uitmaken van je team.

Referenties

[ISO 91] ISO 9126, *"Information Technology - software product evaluation - quality characteristics and guidelines for their use"*, 1991.

[Zeist et al 96] B. Van Zeist, P. Hendriks, R. Paulussen, J. Trienekens, *"Kwaliteit van softwareprodukten; Praktijkervaringen met een kwaliteitsmodel"*, Kluwer Bedrijfswetenschappen, ISBN 90-267-2430-6, mei 1996.

Sympa!

De eerste keer dat ik met Paul te maken kreeg vroeg ik hem om een gunst. Ik was toen student informatica en net geswitcht van oude naar nieuwe stijl. Dat betekende dat ik het vak compilerbouw aan de VU reeds had gevolgd, terwijl ik volgens het nieuwe stijl programma ''programmeeromgevingen'' bij Paul moest volgen. Door de overlap in beide vakken had ik daar geen zin in. Tijdens de pauze van het eerste college programmeromgevingen stapte ik op de docent af en legde mijn verhaal voor. Tot mijn grote verbazing kreeg ik direct een vrijstelling. Vanaf dat moment gold voor mij: Paul Klint is de meest sympatieke hoogleraar van Nederland.
"*Is* dat nou zo?" hoor ik er achter aan komen. Ja, maar niet vanwege die vrijstelling zoals ik later begreep toen ik voor Paul een berg tentamens aan het nakijken was......
Moet ik uitleggen waarom wel natuurlijk. Kijk, het is natuurlijk zo dat je iemand die je eerst aan een baan helpt, en later nog eens waarbij je nog promoveert ook, niet bepaald vervelend kan noemen. Zeker niet als je op dat moment een van Neerlands bekendste punk gitaristen bent en er volgens "de norm" vervaarlijk uit ziet. (zou U dat aandurven beste lezer? Hoog sympa gehalte!!!! Heeft ie later nog eens gedaan trouwens, maar dat moet nog goed aflopen) Daar gaat het echter niet om. Om wat je geleerd hebt van hem? Ook niet. Moet ik ook weer even uitleggen. Wat ik met name van Paul geleerd heb is abstraheren tot je een ons weegt, het probleem dan oplossen en vervolgens zien dat het een paar maanden kost om het te implementeren. Bij dat laatste moet je niet te ambitieus zijn, maar dat ben ik nog steeds, dus dat heb ik niet geleerd.

Tegenwoordig werk ik "in de praktijk" en pas ik die abstracties toe waar het kan. Daar heb ik veel aan en ik merk dat anderen het als een gave zien. Ik niet. Het is gewoon doen alsof problemen er niet zijn. Merkwaardiger wijs kan je dat bij een implementatie "klus" niet doen. Een compilatiefout is, na enig abstraheren toch duidelijk te wijten aan het feit dat de compilerbouwers bij de verkeerde persoon in de klas hebben gezeten nietwaar? Het probleem op dat abstractie nivo oplossen levert pas echt een hersenkraker op. Begrijpt U? Nee? Dat bedoel ik nou met een kraker. Heb ik dus toch niks van Paul's visie begrepen. Waarom moesten wij (de GIPErs van destijds) dan alles implementeren? Hoezo sympatiek? Bloed zweet en tranen en je kon er amper een artikel over gepubliceerd krijgen.

Nee, toen ik ooit een artikel over een implementatie klus had geschreven dat echt interessant was (ooit van een gedistribueerde tekst + structuur editor met hergebruik van 200.000 regels code gehoord? nou dan!) moest het mij overkomen dat er een menselijke editor was die het manuscript zoek had gemaakt. Dat was ook een hoogleraar. Sindsdien vind ik deze man minder sympatiek dan Paul.
Dat zal het dus moeten zijn: relatief. Je vind iemand aardig omdat je anderen niet aardig vind. Lijkt me duidelijk. Zo zijn er hoogleraren die hun promovendi ....of die in de faculteitsraad altijd...... Zeg nou zelf waarde lezer (hoogleraren uitgesloten), kent U er een van een groter symp kaliber?

Tot slot nog iets over 25 jaar in hetzelfde vak en het ter gelegenheid daarvan uitkomen van een boek als dit. Op een of andere manier kan ik niet om de gedachte heen van ``en nou nog niet klaar?'' Moet je hier (ik bedoel mijn werk) mee aankomen. Amerikaans management meneer! Eerst resultaten die we kunnen verkopen en dan praten we verder. Voor de niet ingewijden: het vervolg gesprek gaat dan vaak over de kwaliteit van het geleverde product. Zo niet, dan gaat het over de tegenvallende winsten. Anders gezegd, er wordt ook hier heel wat af geabstraheerd (= doen of problemen er niet zijn, weet U nog). Wellicht gaat het dit bedrijf daarom zo goed. En mij ook. Ik hoop jou ook.

Groet van Wilco.

Ubiquitous Computational Steering

Robert van Liere
Jurriaan D. Mulder

Center for Mathematics and Computer Science CWI,
P.O. Box 94097, 1090 GB Amsterdam, Netherlands

Abstract

Throughout the years, we have had many pleasent discussions with Paul on a wide range of topics. One of our favourites – and certainly often recurring – is the impact of computer technology has on our daily lives. In this article we look into a crystal ball and give our views on the largely unexplored area of ubiquitous computational steering.

1 Introduction.

Consumers can already buy 3D graphics accelerators for their home computers, and Intel estimates that by the year 2000 80 % of all PCs will have 3D graphics.[1] A key question is: the PC is ubiquitous, 3D graphics is about to become ubiquitous, can scientific visualization be made ubiquitous too ? In particular, some interesting questions that should be addressed are:

- What 3D graphics capabilities are there now on PCs and what will there be as graphics becomes ubiquitous?
- What new visualization applications will arise?
- What will be the graphics libraries and development environment?
- How will traditional visualization applications be transformed?

[1] We give two loose definitions: "PC" refers to a price point and consumer orientation, not a particular flavor of box (say, Wintel). "Computer graphics" refers to the more applied part of the realm, especially tools and graphics in support of visualization.

- What is the place of traditional workstation graphics? Is there a place for UNIX in the broad consumer environment?
- How will the Web and traditional applications, such as email, be transformed?

Computational steering is a form of scientific visualization in which end users are continuously provided with visual feedback about the state of a simulation, and can change parameters of the simulation on the fly. The benefit of computational steering is that insight to the simulation is enhanced; end-users can easily navigate through parameter spaces, and test various cause/effect scenarios, etc. Will computational steering ever become ubiquitous [1] ?

Current Practice Well known examples of computational steering have been successfully demonstrated in the areas of atmospheric sciences [2], molecular modeling [3], computational fluid dynamics [4], computer aided design and engineering. These examples bring together various aspects of high performance computing, data management, and interactive 3D computer graphics.

Analysing these examples, the following observations can be made:

- *high end technology.* The technology used is based on a small cluster of high performance compute/data servers linked with high speed networks to high-end UNIX graphics workstations.
- *user constituency.* Users are scientists or engineers who are specialists in the field they are studying. In addition, the number of users during a session is usually limited to one or a small group of specialists.
- *development environment.* The available tools are quite low level and require a high learning curve to use. Developers need to know the details of the simulation, data management, and computer graphics in order to develop computational steering applications.
- *user interface.* User interfaces are tailored towards the user constituency; hence very specialized towards the problem at hand. Interface are usually fixed and do not allow users to experiment with different views on their data.
- *3D interaction.* 3D interaction with 2D input devices is still very cumbersome. This has lead to studying new interaction techniques based on 3D widgets, or using more sophisticated display and interaction devices (eg. VR).

Ubiquitous Computational Steering. In ubiquitous computational steering a large number of users will connect/disconnect 3D PCs to ongoing simulations running remotely on compute servers. We believe that there are many opportunities for ubiquitous computational steering. Some examples are: interactive weather broadcasting, interactive electronic scientific magazines/journals and education books (which go beyond interactive 3D browsing), and the dissemination of knowledge at national laboratories.

Due to the rapid growth of 3D technology, we do not believe that the PC itself will be the factor that prohibits bringing computational steering to the consumer (0.5 Gflop and Reality Engine graphics capabilities in a laptop footprint with PC price points are just around the corner). More importantly, the following problems need to be addressed:

- *bandwidth.* This is almost an unsolvable problem, since more bandwidth is necessary as CPU performance increases. Perhaps adequate compression or multi-resolution schemes will provide an outcome.
- *user constituency and user interface* Users may or may not be specialists in the application field. Non-specialists may be interested in different information than specialists. Hence, the type of information that can be controlled must vary per user.
  Also, the type of information that will be presented and the desired representations will vary per user. Ideally, users will be able to customize the 3D interface.

2 Crystal ball.

We postulate our vision as a list of concrete issues:

1. 3D and interactive 3D can provide added value to the consumer market, and goes beyond the possibilities provided by the current 2D consumer market (which is dominated by page description presentation, SGML, PDF, HTML).
2. Ubiquitous computational steering has many potential applications. We believe that a consumer market for "science on demand" could exist. [2]
3. For technology, the available bandwidth will cause bigger problems than the capabilities of the individual PC. The underlying observation is that pro-

[2] Science on demand – analogous to video on demand – is loosely defined as remotely being able to access scientific information. There could be many forms: from purely informative to educational.

cessor speed and graphics power capabilities growth rates are higher than growth rates in bandwidth of wide area networks.

4. Due to the diverse user constituency of the consumer market, the type of information that will be presented to and steered by must be tailored towards the specific needs and desires of the user. In addition, we believe that users must be able to customize their own representations.

5. It is not clear what input and display capabilities the ubiquitous 3D PC will have. 3D interaction and direct manipulation will always be severely compromised if the ubiquitous 3D PC is limited to 2D input and display devices.

In addition, we list our worries:

1. Our institute (a research Center for Mathematics and Computer Science) have installed 120 SGI desktop workstations. Oddly enough, only the visualization and multimedia researchers make use of the 3D graphics capabilities. All other researchers use 2D capabilities and tools, even though they work on 3D problems.

2. Computational steering will need to be as attractive as 3D PC games, otherwise ubiquitous computational steering will not go beyond the entertainment market. This would clearly be a lost opportunity.

References

[1] M. Weiser. Some computer science issues in ubiquitous computing. *CACM*, 36(7):75–84, July 1993.

[2] R. van Liere and J.J. van Wijk. Steering smog prediction. In *Proceedings of the 1997 Conference on High-Performance Computing and Networking*, pages 241–252. Spirnger Verlag, April 1997.

[3] L.D. Bergman, J.S. Richardson, D.C Richardson, and Jr. F.P. Brooks. View - an exploratory molecular visualization system with user-definable interaction sequencesa. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 117–126, 1993.

[4] S. Bryson. Approaches to the successful design and implementation of VR applications. In R.A. Earnshaw, J.A. Vince, and H. Jones, editors, *Virtual Reality Applications*, pages 3–15. Academic Press, 1995.

The M$\star$ Problem

Lambert Meertens

Abstract

We investigate some relatives of the so-called *Millenium Problem* (also known as the *Year 2000 Problem*).

1 The Millenium Problem

Until not that long ago I thought that the *Millenium Problem* was to come up with a cogent argument for the fact that the first day of the Third Millenium of the Common Era is the first of January 2001, not 2000. The relevance of this problem is that many people plan to party to celebrate the arrival of the new millenium — which is fine — but will do it one year early — which is kinda dumb.

You can imagine my surprise when I noticed that whole conferences were organised, exclusively devoted to the Millenium Problem. As it turns out, there is another problem, namely that width-two counters for years will wrap around from `99` to `00`. This is similar to a problem in buying used cars. If the kilometer counter says `001290`, it does not mean that the car has travelled a total distance of only 1290 kilometers. After all, in order to mislead you, the seller may have driven until the counter reached `999999`, and then gone on for another 1291 km.

My solution to what is now understood to be The Problem is simple and also solves the other one. Start partying on January 1st, 2000, and keep going until 2001. During that year, let the techno-nerds (who don't enjoy partying) fix any emerging technical problems. (For a more principled approach, see the last Section.)

2 The MMMCMXCIX+I Problem

The cause of the Millenium Problem is obviousy the Arabic system of numerals (1, 2, 3, 4, and so on). With the earlier Roman system (`I`, `II`, `III`, `IV`, and so on), the problems with a width-two counter for years become apparent before you can say "legacy". A further advantage of Roman numerals is that the programmer can give a provably safe upper bound on the width of a Roman-number style counter. The Romanly longest possible year is `MMMDCCCLXXXVIII`, which is 15 wide. And so what happens for the next year? We simply go to `MMMDCCCLXXXIX`, only 13 positions.

To be true, there is another problem with Roman numerals. Only a finite number of years can be represented. The last representable year is `MMMCMXCIX`. Note, however, that this is inherent in the specification. The inability of a program to handle a later year should therefore not be considered a bug but a feature.

3 The Myriad Problem

The fix chosen for the Millenium Problem is typically to use a fixed width of four for representing years. This gives only a marginal improvement over width two, considering the current best estimates for the age and time-to-demise of the universe. I predict that the problems with the transition 1999 $\rightarrow$ 2000 are small fry compared to the problems we'll see after 9999, a year before the Second Myriad starts. The idea that all presently running systems will have been superseded by other systems by then is naive.

I therefore propose another solution. Let's agree to start over, after 1999, with a New Era, beginning with the year 0 N.E. Just as, going to daylight-saving time, the clock is put back by one hour, we put back the calendar by 2000 years. The ill-begotten New Millenium Celebrations become then timely New Era Celebrations, and the Third Millenium of the New Era will indeed start on January 1, 2000 N.E. More importantly, wrap-around from `99` to `00` is then the correct behaviour. We have another 99 years respite, which we can use to fix the year width. A width of 15 will suffice.

Lieve Paul,

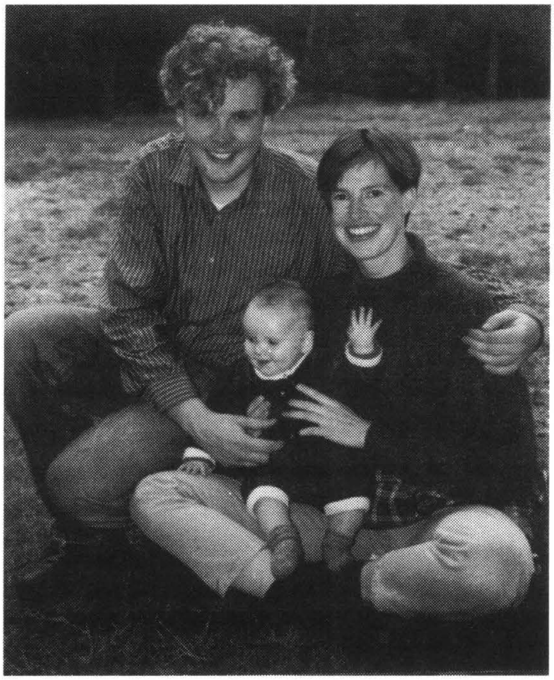
Hartelijk gefeliciteerd met je 25 jarig jubileum !

Waren wij nog rechtgeaarde wetenschappers geweest dan hadden wij het wel geweten. Dan had Emma voor dit feestelijk Liber Amicorum versie X van hoofdstuk Y van haar proefschrift ingeleverd en Casper zijn artikel over de Equation Manager (eerste versie). Inmiddels zijn we afgedwaald naar het bedrijfsleven en schrijven we dit briefje om je te laten weten dat we het erg leuk, nuttig en gezellig vinden dat we nog regelmatig contact met je hebben.

We kunnen er op een moment als dit natuurlijk ook niet omheen dat we elkaar misschien niet dankzij maar toch wel bij jou hebben ontmoet. Tegen vreemden jokt Emma graag dat Casper bij haar is afgestudeerd en Casper dat hij Emma heeft aangezet tot programmeren. Maar jij was degene die werkelijk verantwoordelijk was.

We hebben een goede tijd gehad bij jou in de groep. Emma bedankt je voor de begeleiding bij haar promotie, de goede werksfeer en (zelfs) voor haar docentschap software engineering. Inmiddels weet ze dat de boeken niet overdrijven.

Casper wil je speciaal bedanken voor de vrijheid die hij genoot waardoor hij zich verder kon ontwikkelen en zich buiten de academische wereld kon begeven.



Samen bedanken we je voor de samenwerking die je ons oplegde. Na de succesvolle samenwerking die leidde tot de implementatie van Emma's algoritmen hebben we er zelf nog maar een paar projecten aan vast geknoopt. Gezamenlijke vakanties, een nieuw huis en ons jongste en allerbelangrijkste project: onze dochter Josephine.

Emma van der Meulen
CasperDik
Josephine

Space/Time Efficiency and Redundancy in Terms

Pieter Olivier

Abstract

The ToolBus software architecture uses *terms* as a uniform data representation mechanism to handle data exchange between software components.

The nature of the ToolBus and the applications that we are currently building using the ToolBus poses some special requirements on the underlying term library. It also offers some interesting opportunities for optimizations. In this paper we will discuss the design and implementation of two term libraries, one written in C and one written in Java.

We will show how the design of these term libraries is not only influenced by the operations we need to perform and their efficiency, but also by the redundancy that is typically present in large terms and by the pecularities of the programming languages used to implement the libraries.

1 Introduction

The first ToolBus implementation[BK96] was based on a term library that was adequate for most applications. As the applications we developed with the ToolBus became more complex, several drawbacks of the underlying term library became visible:

- Performance degrades drastically when handling big terms ('big' means more than 1.000.000 nodes).
- The memory footprint of the toolbus interpreter and tools when handling these big terms is unacceptable large.
- When collecting garbage, all the nodes (both free and occupied) where touched by the garbage collector, which has a desastreus effect on both the data cache performance and on disk swapping.
- Transporting a term between ToolBus and tools involves unparsing the term, sending the string representation through some kind of stream, and then parsing it again on the other side.

These drawbacks prompted us to design and implement a new term library that is more efficient in both space and time than the original library.

2 Language independent design decisions

We will first discuss a number of design issues which are to improve the pefor-mance independent of the specific implementation language of the term library.

2.1 Exploiting sharing

Empirical study has shown that most large terms encountered in practice contain a lot of redundancy in the form of common subterms. We decided to exploit this redundancy in order to reduce the memory usage.

The key idea behind the new design is to only build new terms, when a term with the same function symbol and the same arguments does not exist already. When such a term does exist, that term is reused instead. In other words, when two terms have the same value, they are in fact the same term.

This principle has a major implication. Because (sub)terms can be shared by other terms without the creator knowing it, modifications of existing terms are no longer allowed because a modification can have unwanted side-effects. Therefore, all operations on terms must have functional behaviour. Terms effectively become *immutable* after they are created.

2.2 The representation of (argument) lists

Lists are frequently used when constructing terms, and the internal representation influences the efficiency of operations on these lists. So another important design decision is how to represent (argument) lists. There are basically two alternatives:

1. **A simple array of terms**: This is a very memory efficient solution. However, operations like taking the tail of a list or adding a single element to a list become very expensive when the list is long. We also have to take into account that memory management becomes a problem, because we need to handle a lot of different sized memory chunks.

2. **A linked list of terms**: There is some memory overhead needed to administer the list, but most common operations can be done in $O(1)$. Low level memory management becomes much simpler and more efficient, because we only have to allocate and deallocate term nodes of a single size.

Experiments with the first alternative showed that the overhead introduced for common operations like adding an element or retrieving the tail of the list proved to be too high, so we choose the second alternative.

2.3 Term patterns

Constructing and destructing terms by means of matching are frequently used operations when dealing with ToolBus terms. Both types of operations require *term patterns.* A term pattern is a term with holes (holes) in it. A term can be matched against a term pattern, with subterms matching with holes. These holes can be typed, so they only match with subterms of the specified type.

Examples include the building of a new term by filling in the holes in a term pattern, or analyzing an existing term by matching it against a term pattern.

The old ToolBus library used % signs followed by a single character to indicate a hole in a term pattern, e.g. `%s`. The single character following the % sign indicates the type of the hole. For instance the pattern `f(%s)` matches with terms like `f("abc")` and `f("")`, but not with terms like `f(1)` or `g("abc")`. A special term parser parses this kind of patterns and performes assignments (matching) or replacements (constructing) on the fly. There are several disadvantages to this approach:

- Everytime a match is performed the pattern has to be reparsed. Parsing is a time consuming operation, so this proved to be a major efficiency bottleneck.
- Having an implementation of both a regular term parser and a pattern term parser also has given rise to quite a number of bugs.

To solve these problems we re-use the special *placeholder* term type. A template term is a term of the form `<term>`. In the old ToolBus implementation, placeholders were used to implement the T-script type system. But this notion of placeholders has proved to be extremely useful as a replacement for the old % style term patterns.

In the new library implementations, we use placeholder terms to denote term patterns. The angled brackets denote a hole in a term, and the term between the angled brackets describes the expected type of the hole. The pattern `f(<str>)` for instance, is equivalent to the original pattern `f(%s)`.

This solves the problem of needing another parser just for pattern terms, because term patterns are just regular terms that can be parsed using the regular term parser. It also solves the efficiency problem, because all operations that need a term pattern, can now work with an actual term that represents the pattern instead of a string that has to be parsed. In practice this means that all term patterns are parsed once at program initialization time, and then reused for every matching or construction operation.

Other major design decisions, like memory management issues and error handling, depend very much on the implementation language, therefore we will now differentiate between the C library and the Java library.

3 C library

C is a very low level language. Therefore, the C term library implements a more functionality than the Java library. But this also creates a lot of opportunities for optimizations that cannot be expoited in a more high level language.

There are two issues we want to address explicitly, how to implement memory management in combination with term sharing, and how to represent the empty list.

3.1 Memory management

Terms are shared, so the same term can be in use deep within the term library and somewhere in the user application. This means that we cannot leave it to the user to decide when to deallocate the memory occupied by a term. C offers no support for automatic memory management, so we have to implement this ourselves. Obvious choices include *garbage collection* and *reference counting.* We want to use our library in a setting where multiple applications use the same term data space by means of shared memory, in order to avoid expensive message passing operations. We do not want to stop all applications using this term space when garbage collection is in progress, so we decided to use reference counting in combination with lazy deallocation[JL96] instead. Another solution is to use some kind of distributed garbage collection algorithm, but the advantages of such a (complex) solution are not obvious.

3.2 List representation

In the old ToolBus library, error handling is a serious problem. The usual strategy in C programs is to return `NULL` when an error is detected. In the old library, the empty list is represented by the same `NULL` value as well. This leeads to ackward C code in order to implement error handling. There are two obvious solutions to this problem:

- Choose another value to signal an error condition. This leads to code like this:

  ```
  aterm *t;

  t = read_term(stdin);
  if(t == ERROR)
    fprintf(stderr, "error while reading term.");
  if(t == NULL)
    fprintf(stdout, "the empty list.");
  ```

  This code is somewhat counter intuïtive for the average C programmer, but works perfectly.

- Choose another value representing the empty list. This leads to code like this:

  ```
  aterm *t;

  t = read_term(stdin);
  ```

```
if(t == NULL)
  fprintf(stderr, "error while reading term.");
if(t_is_empty(t))
  fprintf(stdout, "The empty list.");
```

This code has the added advantage that all access functions on terms also work on the empty list. This eliminates a lot of code involving the special case of the empty list. For the new library, we choose the second alternative.

4 Java library

Although Java offers a built in garbage collector, we have to take special actions in order to implement the term sharing. Curiously enough, representation of the empty list is also an issue in the Java implementation. Therefore the same issues will be addressed as in the C implementation.

4.1 Memory management

The most straightforward class diagram for the term library is shown in Figure 1.

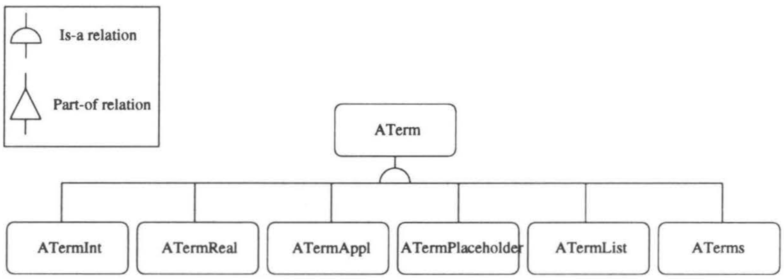


Figure 1: Simple ATerm class library

However, our design decision to use optimally shared terms implies that we have to record all terms in a central data structure that can be used to check if a term already exists before creating it. This means that the Java garbage collector cannot be used to remove unused terms, because there is always this central data structure that still refers to these terms.

This implies that we have to do our own memory management. As in the C case, we opted for reference counting. To implement this, we introduce a second layer of 'term references' that make sure the reference count of the actual term objects is kept up to date. The resulting class diagram is shown in Figure 2.

For every term type we introduce a complementary reference type. The user of our class library only works with these reference types. In the constructor of the reference type, the reference count of the underlying term is increased. In

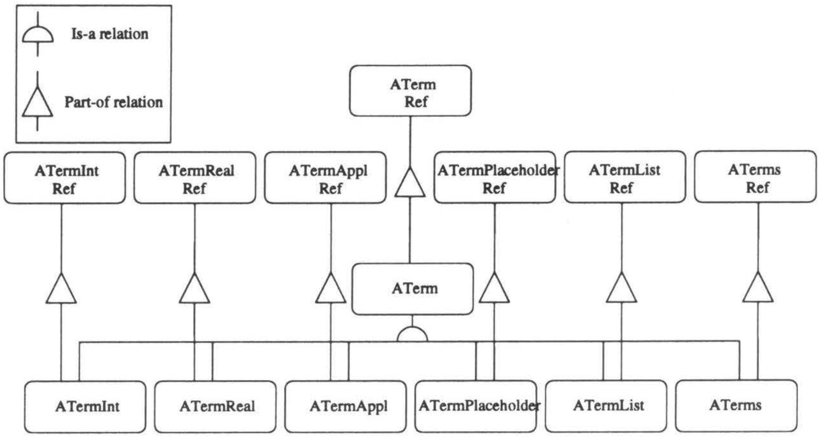


Figure 2: Introducing reference classes

the `finalize` method of the reference type, the reference count of the underlying term is decreased. When the reference count goes to zero, the term is removed from the central term repository. The Java garbage collector can then dispose of the term object when the object is no longer in use.

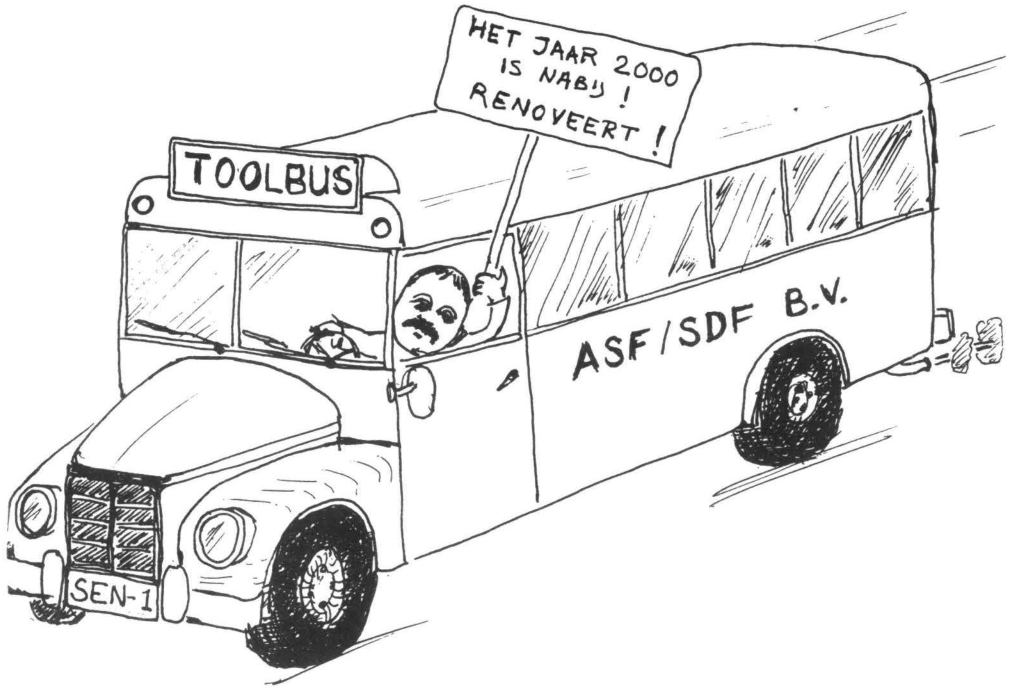
4.2 List representation

Error handling in Java is done through exception handling, eliminating the need for a special error element. However, in an object oriented language like Java it is a very bad idea to use a non-term object (`nil`) to represent a special kind of object (the empty list), because this would mean that every method handling terms must first check for the special empty list case before calling a method of a term object. So to represent the empty list in the Java case, we use a normal list object with both the first and next links initialized to nil.

References

[BK96] J.A. Bergstra and P. Klint. The toolbus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION)*, volume 1061 of *LNCS*, pages 75–88. Springer-Verlag, 1996.

[JL96] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, 1996.

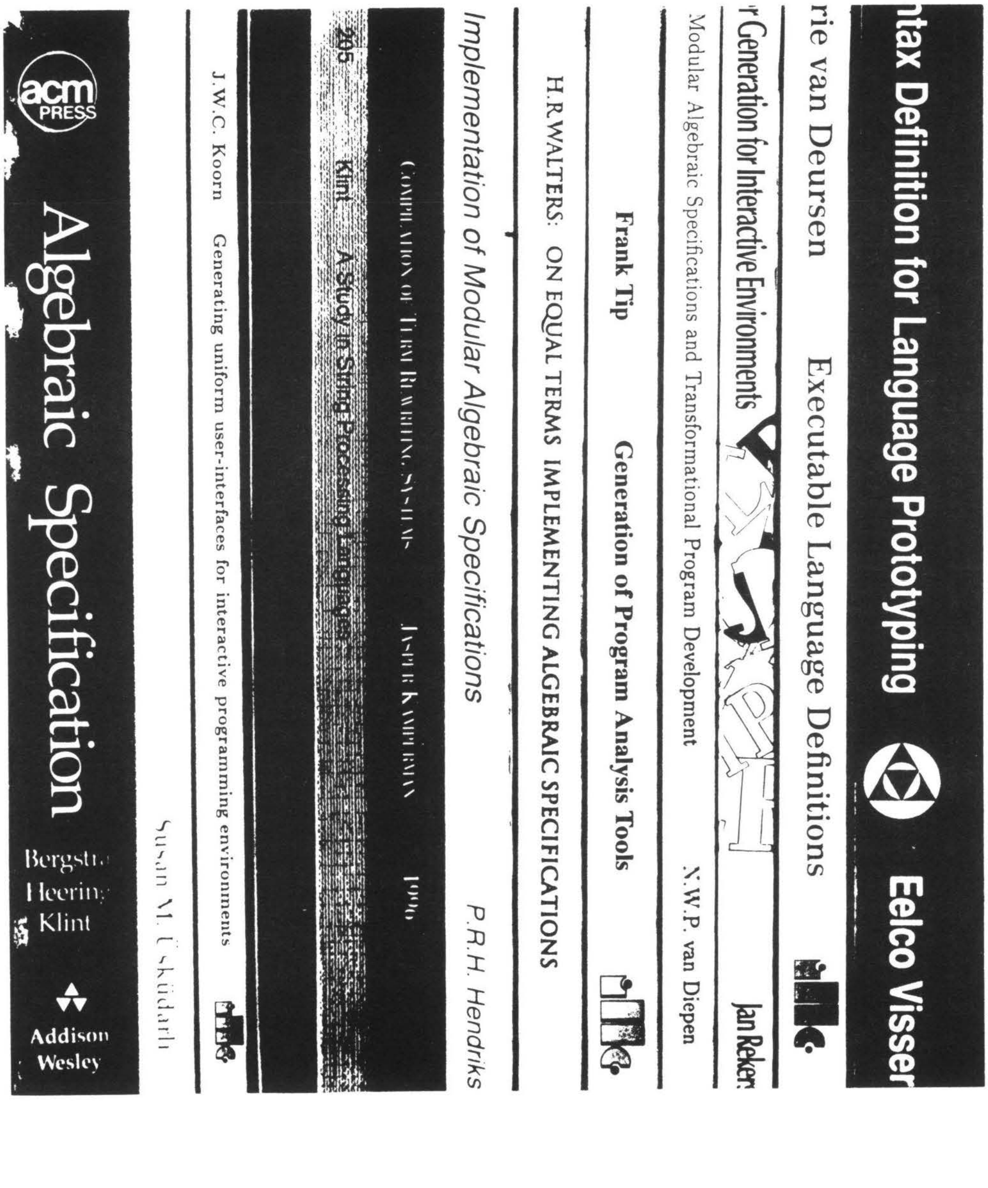


Paul Klint, een van CWI's hooggeleerde heren
Is al 25 jaar specialist in het programmeren
Hij wordt gefascineerd door het millennium probleem
En zegt: in het jaar 2000 crasht menig systeem
Tenzij u het door mij laat renoveren!

Gerard van Oortmerssen

Als een van de velen heb ik mogen genieten van de rijke inspiratie bron die Paul Klint heet

Jan Rekers



MEERSOORTIGE ALGEBRA MET MOGELIJK LEGE SOORTEN: VRIJE UITBREIDINGEN EN DE *HSP*-STELLING

P.H. Rodenburg
Gemeeentelijke Universiteit van Amsterdam

Inleiding

Meersoortige algebra werd geïntroduceerd door Higgins in 1963 [H]. Birkhoff en Lipson [BL] suggereerden in 1970 dat het onderwerp een bijzonder belang had voor de informatica. Zij dachten daarbij met name aan automaten, met de soorten van toestanden, invoerletters, en uitvoerletters. Het wijdere toepassingsbereik der datatypen werd voor het eerst opgemerkt door Kaphengst en Reichel [KaR] in 1971.

Vanaf het eerste begin gaat de meersoortige algebra gebukt onder de omstandigheid dat 'alles in principe net zo gaat als in de enkelsoortige algebra' — en dat dat toch nooit helemaal zeker is. Twee voorbeelden daarvan worden in het onderstaande besproken.

Ten eerste, uit de aard van de toepassing op datatypen volgt een vergrote belangstelling voor signatuurverandering. Als er meer soorten zijn, kunnen we van oorsprong verschillende soorten dezelfde naam geven. De oude soorten worden dan verenigd onder de nieuwe naam. Omdat homomorfismen soortsgewijs worden gedefinieerd verliezen we daarmee aan vrijheid: een element a van twee overlappende soorten A_{r_1} en A_{r_2} van een algebra $\mathfrak{A}$, met verschillende waarden onder de r_1- en r_2-componenten van een homomorfisme $\mathfrak{f}$, kan na een signatuurverandering μ die r_1

en r_2 identificeert maar één waarde hebben onder de $\mu(r_1)$-component van een met $\mathfrak{f}$ verwant homomorfisme — een rimpel in de uitbreiding van de enkelsoortige theorie die te denken geeft.

Ten tweede: de klassieke logica is zo georganiseerd dat het tautologisch waar is dat er iets bestaat; en als een klassieke logicus [S] uitlegt wat meersoortige logica zou moeten zijn, en hoe je die reduceert tot enkelsoortige, dan zijn er per axioma van elke soort exemplaren. Maar bij een nieuw begin willen we niet te zuinig doen, en het is wel eens handig om niet te hoeven eisen dat er van elke soort iets is. Met name heeft dan elke variëteit een initieel element — dus elke algebraïsche specificatie een initiële algebra-semantiek. Een consistente theorie vereist echter een wezenlijk dieper inzicht in de rol van variabelen in vergelijkingen dan de gemiddelde student meeneemt uit een Inleiding Logica (zie [GM1], [ELM], [GM2] en [MS]).

Om één en ander te illustreren behandel ik twee nauw samenhangende onderwerpen: vrije uitbreidingen en de ***HSP***-stelling. Vrije uitbreidingen vormen de semantiek van import (bij initiële algebra-semantiek) en parametrisering. De ***HSP***-stelling is een gevolg van het bestaan van vrije uitbreidingen dat de expressieve kracht van de equationele logica karakteriseert. Het gaat hier om wezenlijke trekken van de theorie der enkelsoortige algebra's, die bij generalisatie behouden dienen te blijven. Ze worden door zowel Higgins als Birkhoff en Lipson besproken. De hier beschreven constructie van vrije uitbreidingen is algemener dan de constructies van de pioniers.

Veel origineels zal in het onderstaande niet voorkomen. Een belangrijke inspiratiebron was het leerboek van Loeckx, Ehrich en Wolf [LEW].

Terminologie en notatie

We veronderstellen een klasse ¥ van primitieve *symbolen* waartussen geen verzamelingstheoretische relaties bestaan. Laat S een deelverzameling zijn van ¥. Een *functiedeclaratie over* S is een drietal $\mathbf{f} = \langle \mathrm{f}, \mathrm{s}, w \rangle$ bestaande uit, achtereenvolgens, een element f van ¥ (de *functienaam*), een element van S, en een niet-leeg rijtje (woord) w van elementen van S. We schrijven $\mathbf{f}\colon \mathrm{s} \leftarrow w$, of $\mathbf{f}\colon w \rightarrow \mathrm{s}$. Een *signatuur* is een verzameling $\Sigma = \mathrm{S} \cup \mathbf{F}$ waarin $\mathbf{F}$ een verzameling is van functiedeclaraties over S. We noemen S de verzameling der *soortnamen* van Σ; we noteren S als $\mathrm{S}\Sigma$ en $\mathbf{F}$ als $\mathbf{F}\Sigma$.

Laat Σ en Σ' signaturen zijn. Een *morfisme* van Σ naar Σ' is een functie $\mu\colon \Sigma' \leftarrow \Sigma$ die $\mathrm{S}\Sigma$ afbeeldt naar $\mathrm{S}\Sigma'$, en elke $\mathbf{f} = \langle \mathrm{f}, \mathrm{s}, \mathrm{s}_1 \ldots \mathrm{s}_k \rangle \in \mathbf{F}\Sigma$ naar een functiedeclaratie van de vorm $\langle \mathrm{g}, \mu(\mathrm{s}), \mu(\mathrm{s}_1) \ldots \mu(\mathrm{s}_k) \rangle$ in $\mathbf{F}\Sigma'$.

Zij Σ een signatuur. Een *algebra* van signatuur Σ (Σ-algebra) is een toekenning $\mathfrak{A}$ van een (desgewenst lege) verzameling $\mathfrak{A}(\mathrm{s})$ aan elke $\mathrm{s} \in \mathrm{S}\Sigma$, en een functie

$$\mathfrak{A}(\mathbf{f})\colon \mathfrak{A}(\mathrm{s}_0) \leftarrow \mathfrak{A}(\mathrm{s}_1) \times \ldots \times \mathfrak{A}(\mathrm{s}_k)$$

aan elke $\mathbf{f} = \langle \mathrm{f}, \mathrm{s}_0, \mathrm{s}_1 \ldots \mathrm{s}_k \rangle \in \mathbf{F}\Sigma$. De klasse van alle Σ-algebra's noteren we als Alg_Σ.

Laat $\mathfrak{A}$ en $\mathfrak{B}$ Σ-algebra's zijn. Een *homomorfisme* van $\mathfrak{A}$ naar $\mathfrak{B}$ is een familie $\mathfrak{h} = \langle \mathfrak{h}_\mathrm{s} \mid \mathrm{s} \in \mathrm{S}\Sigma \rangle$ van functies $\mathfrak{h}_\mathrm{s}$: $\mathfrak{A}(\mathrm{s}) \rightarrow \mathfrak{B}(\mathrm{s})$ die commuteren met de operaties van $\mathfrak{A}$ en $\mathfrak{B}$: als $\mathbf{f}\colon \mathrm{s}_1 \ldots \mathrm{s}_k \rightarrow \mathrm{s} \in \Sigma$, en $a_1 \in \mathfrak{A}(\mathrm{s}_1), \ldots, a_k \in \mathfrak{A}(\mathrm{s}_k)$, dan

$$\mathfrak{h}_\mathrm{s}(\mathfrak{A}(\mathbf{f})(a_1, \ldots, a_k)) = \mathfrak{B}(\mathbf{f})(\mathfrak{h}_{\mathrm{s}_1}(a_1), \ldots, \mathfrak{h}_{\mathrm{s}_k}(a_k)).$$

Een bijzonder homomorfisme is de *identiteit*

$$1_{\mathfrak{A}} := \langle 1_{\mathfrak{A}(s)} | \, s \in S\Sigma \rangle.$$

Compositie van homomorfismen is soortsgewijs: $\mathfrak{k} \circ \mathfrak{h} = \langle \mathfrak{k}_s \circ \mathfrak{h}_s | \, s \in S\Sigma \rangle$.

Laat $\mu \colon \Sigma' \leftarrow \Sigma$ een signatuurmorfisme zijn, en $\mathfrak{A} \in Alg_{\Sigma'}$. Het *μ-reduct* van $\mathfrak{A}$ is de Σ-algebra $\mathfrak{A}|\mu$ gedefinieerd door

$$(\mathfrak{A}|\mu)(\sigma) = \mathfrak{A}(\mu(\sigma))$$

voor alle $\sigma \in \Sigma$. Als $\Sigma \subseteq \Sigma'$, en μ is de kanonieke inbedding van Σ in Σ′, dan schrijven we ook $\mathfrak{A}|\Sigma$. In het bijzonder is het *universum* van de Σ′-algebra $\mathfrak{A}$ het reduct $A = \mathfrak{A}|S\Sigma'$ ($= \langle A_{s'} | \, s' \in S\Sigma' \rangle$; een SΣ′-algebra, of *Σ′-gesorteerde verzameling*). Als $\mathfrak{h} \colon \mathfrak{A} \rightarrow \mathfrak{B}$ een homomorfisme is, dan is $\mathfrak{h}|\mu$ het homomorfisme $\langle \mathfrak{h}_{\mu(s)} | \, s \in S\Sigma \rangle$ van $\mathfrak{A}|\mu$ naar $\mathfrak{B}|\mu$. Reductie langs μ is een *functor*: $1_{\mathfrak{A}}|\mu = 1_{\mathfrak{A}|\mu}$, en $(\mathfrak{k} \circ \mathfrak{h})|\mu = (\mathfrak{k}|\mu) \circ (\mathfrak{h}|\mu)$.

Laat $\mathfrak{A}$, $\mathfrak{B}$ Σ-algebra's zijn. Als voor elke $s \in S\Sigma$, $A_s \subseteq B_s$, en voor iedere functiedeclaratie $\mathbf{f} = \langle f, s_0, s_1 \ldots s_n \rangle$ in $\mathbf{F}\Sigma$,

$$\mathfrak{A}(\mathbf{f}) = \mathfrak{B}(\mathbf{f})|(\mathfrak{A}(S_1) \times \ldots \times \mathfrak{A}(s_n))$$

(de restrictie van $\mathfrak{B}(\mathbf{f})$ tot $\mathfrak{A}(S_1) \times \ldots \times \mathfrak{A}(s_n)$), dan heet $\mathfrak{A}$ een *subalgebra* van $\mathfrak{B}$, notatie $\mathfrak{A} \subseteq \mathfrak{B}$. Als $\mathfrak{A} \in Alg_\Sigma$ en $B \subseteq A$, dan heet B een *subuniversum* van $\mathfrak{A}$ als B gesloten is onder de operaties van $\mathfrak{A}$, i.e. voor iedere functiedeclaratie $\mathbf{f} = \langle f, s_0, s_1 \ldots s_n \rangle$ in $\mathbf{F}\Sigma$ en voor alle $a_1 \in B_{s_1}, \ldots, a_n \in B_{s_n}$, $\mathfrak{A}(\mathbf{f})(a_1, \ldots, a_n) \in B_{s_0}$. Als $\mathfrak{h} \colon \mathfrak{A} \rightarrow \mathfrak{B}$ een homomorfisme is, dan is in het bijzonder $\mathfrak{h}(A)$ ($= \langle \mathfrak{h}_s(A_s) | s \in S\Sigma \rangle$) een subuniversum van $\mathfrak{B}$, en $\mathfrak{h}^{-1}(B) = \langle \mathfrak{h}_s^{-1}(A_s) | s \in S\Sigma \rangle$ een subuniversum van $\mathfrak{A}$. Een subuniversum X van een algebra $\mathfrak{C}$ bepaalt een subalgebra $\mathfrak{X} \subseteq \mathfrak{C}$; zo hebben we $\mathfrak{h}(\mathfrak{A}) \subseteq \mathfrak{B}$

en $\mathfrak{h}^{-1}(\mathfrak{B}) \subseteq \mathfrak{A}$. Een Σ-algebra $\mathfrak{A}$ wordt *voortgebracht* door een Σ-gesorteerde verzameling $X \subseteq A$ als A het kleinste subuniversum is van $\mathfrak{A}$ dat X (soortsgewijs) omvat. Als $\mathfrak{A}$ wordt voortgebracht door X, en $\mathfrak{g}, \mathfrak{h}: \mathfrak{A} \to \mathfrak{B}$ zijn homomorfismen, dan $\mathfrak{g} = \mathfrak{h}$ als $\mathfrak{g}|X = \mathfrak{h}|X$. (De restrictie is soortsgewijs: $\mathfrak{f}|X = \langle \mathfrak{f}_s|X_s | \ s \in S\Sigma\rangle$.)

Een *congruentierelatie* op een Σ-algebra $\mathfrak{A}$ is een familie $\langle \theta_s | \ s \in S\Sigma\rangle$ van equivalentierelaties $\theta_s \subseteq A_s \times A_s$ met de eigenschap dat, als $\mathbf{f} = \langle f, s_0, s_1 \ldots s_k\rangle \in \mathbf{F}\Sigma$,

$$\text{als } \langle a_1, b_1\rangle \in \theta_{s_1}, \ldots, \langle a_k, b_k\rangle \in \theta_{s_k}, \text{ dan} \\ \langle \mathfrak{A}(\mathbf{f})(a_1, \ldots, a_k), \mathfrak{A}(\mathbf{f})(b_1, \ldots, b_k)\rangle \in \theta_{s_0}.$$

De verzameling van alle congruenties van een algebra $\mathfrak{A}$ noteren we als Con$\mathfrak{A}$. In plaats van $\langle a, b\rangle \in \theta_s$ schrijf ik meestal $a \equiv_s b\ (\theta)$. De θ_s-nevenklasse $\{b \in A_s | \ a \equiv_s b\ (\theta)\}$ noteer ik als a/θ_s. Een homomorfisme $\mathfrak{h}: \mathfrak{A} \to \mathfrak{B}$ induceert een congruentierelatie Ker $\mathfrak{h}$ gedefinieerd door

$$a \equiv_s b \ (\text{Ker}\ \mathfrak{h}) \text{ dan en slechts dan als } \mathfrak{h}_s(a) = \mathfrak{h}_s(b).$$

Het *quotient* van $\mathfrak{A}$ over de congruentie θ op $\mathfrak{A}$ is de Σ-algebra $\mathfrak{A}/\theta$ gedefinieerd door

$$(\mathfrak{A}/\theta)(s) = \{a/\theta_s | \ a \in A_s\}, \\ (\mathfrak{A}/\theta)(\mathbf{f})(a_1/\theta_s, \ldots, a_k/\theta_s) = \mathfrak{A}(\mathbf{f})(a_1, \ldots, a_k)/\theta_s .$$

Het kanonieke homomorfisme $\mathfrak{h}_\theta: \mathfrak{A} \to \mathfrak{A}/\theta$ is gedefinieerd door

$$(\mathfrak{h}_\theta)_s(a) = a/\theta_s ;$$

de componenten $(\mathfrak{h}_\theta)_s$ zijn surjectief. We noemen een homomorfisme surjectief als zijn componenten surjectief zijn; analoog voor 'injectief'. Een *isomorfisme* is een homomorfisme dat zowel injectief is als surjectief. Men schrijft $\mathfrak{A} \cong \mathfrak{B}$ ($\mathfrak{A}$ is isomorf met $\mathfrak{B}$) als er een isomorfisme van $\mathfrak{A}$ naar $\mathfrak{B}$

bestaat. Merk op dat een homomorfisme $\mathfrak{f}\colon \mathfrak{A} \to \mathfrak{B}$ dan en slechts dan een isomorfisme is als er een homomorfisme $\mathfrak{g}$: $\mathfrak{B} \to \mathfrak{A}$ bestaat zo dat $\mathfrak{g}\circ\mathfrak{f} = 1_{\mathfrak{A}}$ en $\mathfrak{f}\circ\mathfrak{g} = 1_{\mathfrak{B}}$. Als $\mathfrak{f}\colon \mathfrak{A} \to \mathfrak{B}$ surjectief is, schrijven we $\mathfrak{f}\colon \mathfrak{A} \twoheadrightarrow \mathfrak{B}$; $\mathfrak{B}$ heet dan een *homomorf beeld* van $\mathfrak{A}$. Een injectief homomorfisme heet ook een *inbedding*; notatie $\mathfrak{f}\colon \mathfrak{A} \hookrightarrow \mathfrak{B}$.

Een *verzameling variabelen* voor een gegeven signatuur Σ is een familie $X = \langle X_s | \, s \in S\Sigma \rangle$ van paarsgewijs disjuncte verzamelingen. De $\Sigma(X)$-*termen* zijn gedefinieerd door:
(i) als $x \in X_s$, dan is x een $\Sigma(X)$-term van soort s;
(ii) als $\mathbf{f} = \langle f, s_0, s_1 \ldots s_n \rangle \in \Sigma$, en t_i, voor $1 \leq i \leq n$, is een $\Sigma(X)$-term van soort s_i, dan is $\mathbf{f}t_1 \ldots t_n$ een $\Sigma(X)$-term van soort s_0.
Terwille van de leesbaarheid worden soms haakjes, komma's en spaties toegevoegd; een term als in (ii) wordt dan $\mathbf{f}(t_1, \ldots, t_n)$.

Merk op dat de aldus gedefinieerde termen hun eigen ontleding bevatten. Een *concretere* definitie zou afzien van de disjunctheid van de X_s en bij (ii) $ft_1 \ldots t_n$ invoeren in plaats van $\mathbf{f}t_1 \ldots t_n$. De overbelastingsproblemen waar dat toe zou leiden zijn interessant, maar hier (en misschien is ook dat een veeg teken) niet aan de orde.

De $\Sigma(X)$-termen vormen een Σ-algebra $\mathfrak{T} = \mathfrak{T}_{\Sigma,X}$ onder de definitie
(i) $\mathfrak{T}(s)$ is de verzameling der $\Sigma(X)$-termen van soort s;
(ii) als $\mathbf{f} = \langle f, s_0, s_1 \ldots s_n \rangle \in \Sigma$, en $t_i \in \mathfrak{T}(s_i)$, voor $1 \leq i \leq n$, dan

$$\mathfrak{T}(\mathbf{f})(t_1, \ldots, t_n) = \mathbf{f}t_1 \ldots t_n.$$

Een homomorfisme van een verzameling X van variabelen voor een gegeven signatuur Σ naar het universum van een Σ-algebra $\mathfrak{A}$ heet een *bedeling* aan X in $\mathfrak{A}$. Een bedeling α: X $\rightarrow$ A induceert een uniek homomorfisme $\overline{\alpha}:\mathfrak{T}_{\Sigma,X} \rightarrow \mathfrak{A}$. ($\mathfrak{T}_{\Sigma,X}$ heet daarom *volledig vrij voortgebracht door* X.) Een bedeling in een termalgebra heet een *substitutie*. Als σ een substitutie is, dan schrijven we i.h.a. t^{σ} in plaats van $\overline{\sigma}(t)$.

Het *direct product* $\prod\limits_{i\in I}\mathfrak{A}_i$ van een familie $\langle\mathfrak{A}_i | i\in I\rangle$ van Σ-algebra's is de Σ-algebra $\mathfrak{A}$ gedefinieerd door

$$\mathfrak{A}(\mathrm{s}) = \prod_{i\in I}\mathfrak{A}_i(\mathrm{s})$$

en voor $\mathbf{f} = \langle \mathrm{f}, \mathrm{s}_0, \mathrm{s}_1 \ldots \mathrm{s}_n\rangle$ in $\mathbf{F}\Sigma$ en $a_1 \in \mathfrak{A}(\mathrm{s}_1),\ldots,$ $a_n\in \mathfrak{A}(\mathrm{s}_n)$, voor alle $i \in I$,

$$\mathfrak{A}(\mathbf{f})(a_1,\ldots,a_n)(i) = \mathfrak{A}_i(\mathbf{f})(a_1(i),\ldots,a_n(i)).$$

De projecties $\mathfrak{p}_j : \langle a_i | i\in I\rangle \mapsto a_j$ $(j\in I)$ zijn homomorfismen van $\prod\limits_{i\in I}\mathfrak{A}_i$ naar $\mathfrak{A}_j$. Merk op dat $\mathfrak{p}_j$ surjectief is dan en slechts dan als voor alle s zo dat een $i \in I$ bestaat met $\mathfrak{A}_i(\mathrm{s}) = \emptyset$, ook $\mathfrak{A}_j(\mathrm{s}) = \emptyset$.

We noteren de diagonaal $\langle\{\langle x, x\rangle | x \in X_{\mathrm{s}}\} | \mathrm{s} \in \mathrm{S}\Sigma\rangle$ van een Σ-gesorteerde verzameling X als Δ_X.

Als $\mathfrak{f}_i : \mathfrak{B} \rightarrow \mathfrak{A}_i$ $(i\in I)$ homomorfismen zijn, dan is er precies één homomorfisme $\mathfrak{f}: \mathfrak{B} \rightarrow \prod\limits_{i\in I}\mathfrak{A}_i$ met de eigenschap dat $\mathfrak{p}_i \circ \mathfrak{f} = \mathfrak{f}_i$ voor alle $i\in I$. Als $\bigcap\limits_{i\in I}\mathrm{Ker}\mathfrak{f}_i = \Delta_B$, dan is $\mathfrak{f}$ injectief. We schrijven $\mathfrak{f} = (\mathfrak{f}_i | i\in I)$, of $(\mathfrak{f}_i)_{i\in I}$.

Vrije uitbreidingen

Laat μ: Σ → Σ′ een signatuurmorfisme zijn. Reductie langs μ maakt uit een Σ′-algebra een uniek bepaalde Σ-algebra, en van een Σ′-homomorfisme een uniek Σ-homomorfisme. De omgekeerde richting, *expansie* langs μ, is niet eenduidig.

We kunnen wel bij een gegeven Σ-algebra $\mathfrak{A}$ een paar $\langle \mathfrak{F}(\mathfrak{A}), \eta \rangle$ construeren van een Σ'-algebra $\mathfrak{F}(\mathfrak{A})$ en een homomorfisme $\eta: \mathfrak{A} \to \mathfrak{F}(\mathfrak{A})|\mu$ dat *universeel is van* $\mathfrak{A}$ *naar* $|\mu$: dat wil zeggen dat voor elke $\mathfrak{B} \in Alg_{\Sigma'}$ bij ieder Σ-homomorfisme $\mathfrak{g}: \mathfrak{A} \to \mathfrak{B}|\mu$ precies één Σ'-homomorfisme $\mathfrak{h}: \mathfrak{F}(\mathfrak{A}) \to \mathfrak{B}$ bestaat zo dat $(\mathfrak{h}|\mu) \circ \eta = \mathfrak{g}$.

$$
\begin{tikzcd}
\mathfrak{A} \arrow[r, "\eta"] \arrow[dr, "\mathfrak{g}"'] & \mathfrak{F}(\mathfrak{A})|\mu \arrow[d, dashed, "\mathfrak{h}|\mu"] \\
& \mathfrak{B}|\mu
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathfrak{F}(\mathfrak{A}) \arrow[d, dashed, "\mathfrak{h}"] \\
\mathfrak{B}
\end{tikzcd}
$$

De constructie berust op de homomorfiestelling.

1. *Homomorfiestelling*. Als $\mathfrak{f}: \mathfrak{B} \leftarrow \mathfrak{A}$ een Σ-homomorfisme is, en $\theta \subseteq \mathrm{Ker}\mathfrak{f}$ een congruentierelatie op $\mathfrak{A}$, dan bestaat er precies één homomorfisme $\mathfrak{g}: \mathfrak{B} \leftarrow \mathfrak{A}/\theta$ zo dat $\mathfrak{g} \circ \mathfrak{h}_\theta = \mathfrak{f}$. Als $\theta = \mathrm{Ker}\mathfrak{f}$ dan is $\mathfrak{g}$ injectief.

Bewijs. Laat $\mathfrak{f}: \mathfrak{B} \leftarrow \mathfrak{A}$ een homomorfisme zijn en $\theta \subseteq \mathrm{Ker}\mathfrak{f}$ een congruentierelatie op $\mathfrak{A}$. Definieer, voor elke s $\in \mathrm{S}\Sigma$: $\mathfrak{g}_\mathrm{s}(a/\theta_\mathrm{s}) = \mathfrak{f}_\mathrm{s}(a)$. Die definitie is eenduidig, want als $a/\theta_\mathrm{s} = b/\theta_\mathrm{s}$, dan $\langle a, b \rangle \in \theta_\mathrm{s} \subseteq (\mathrm{Ker}\mathfrak{f})_\mathrm{s}$, dus $\mathfrak{f}_\mathrm{s}(a) = \mathfrak{f}_\mathrm{s}(b)$. Verder is $\mathfrak{g}$ een homomorfisme omdat voor $\mathbf{f}$ = f: $\mathrm{s}_1 \ldots \mathrm{s}_n \to \mathrm{s} \in \mathbf{F}\Sigma$,

$$\mathfrak{g}_\mathrm{s}((\mathfrak{A}/\theta)(\mathbf{f})(a_1/\theta_{\mathrm{s}_1}, \ldots, a_n/\theta_{\mathrm{s}_n})) = \mathfrak{g}_\mathrm{s}((\mathfrak{A}(\mathbf{f})(a_1, \ldots, a_n))/\theta_\mathrm{s})$$

(per definitie van $\mathfrak{A}/\theta$)

$= \mathfrak{f}_\mathrm{s}(\mathfrak{A}(\mathbf{f})(a_1, \ldots, a_n))$ (per definitie van $\mathfrak{g}$)
$= \mathfrak{B}(\mathbf{f})(\mathfrak{f}_{\mathrm{s}_1}(a_1), \ldots, \mathfrak{f}_{\mathrm{s}_n}(a_n))$ ($\mathfrak{f}$ is een homomorfisme)
$= \mathfrak{B}(\mathbf{f})(\mathfrak{g}_{\mathrm{s}_1}(a_1/\theta_{\mathrm{s}_1}), \ldots, \mathfrak{g}_{\mathrm{s}_n}(a_n/\theta_{\mathrm{s}_n}))$ (per definitie van $\mathfrak{g}$).

Als $a \in A_s$, dan $(\mathfrak{g} \circ \mathfrak{h}_\theta)_s(a) = \mathfrak{g}_s((\mathfrak{h}_\theta)_s(a)) = \mathfrak{g}_s(a/\theta_s) = \mathfrak{f}_s(a)$, dus $\mathfrak{g} \circ \mathfrak{h}_\theta = \mathfrak{f}$. Als $\mathfrak{k} : \mathfrak{A}/\theta \to \mathfrak{B}$ met $\mathfrak{k} \circ \mathfrak{h}_\theta = \mathfrak{f}$, dan voor elk element a/θ_s van A/θ:

$$\mathfrak{k}_s(a/\theta_s) = (\mathfrak{k} \circ \mathfrak{h}_\theta)_s(a) = \mathfrak{f}_s(a) = \mathfrak{g}_s(a/\theta_s),$$

dus $\mathfrak{k} = \mathfrak{g}$.

Als $\theta = \text{Ker } \mathfrak{f}$, dan volgt uit $\mathfrak{g}_s(a/\theta_s) = \mathfrak{g}_s(a'/\theta_s)$, via $\mathfrak{f}_s(a) = \mathfrak{f}_s(a')$, dus $\langle a, a' \rangle \in (\text{Ker } \mathfrak{f})_s$, dat $\langle a, a' \rangle \in \theta_s$, dus $a/\theta_s = a'/\theta_s$; dat wil zeggen dat $\mathfrak{g}$ injectief is. ¶

Gevolg 1. Als $\mathfrak{f} : \mathfrak{A} \twoheadrightarrow \mathfrak{B}$, dan $\mathfrak{A}/\text{Ker}\mathfrak{f} \cong \mathfrak{f}(\mathfrak{A})$.

Merk op dat de nevenklassen onder Ker$\mathfrak{f}$ de volledige originelen $\mathfrak{f}_s^{-1}\{b\}$ zijn van de elementen van $\mathfrak{f}(\mathfrak{A})$, en dat het isomorfisme $\mathfrak{f}_s^{-1}\{b\}$ naar b afbeeldt.

Gevolg 2. Als $\mathfrak{f}: \mathfrak{A} \to \mathfrak{B}$, $\mathfrak{g}: \mathfrak{A} \twoheadrightarrow \mathfrak{C}$, en $\text{Ker}\mathfrak{g} \subseteq \text{Ker}\mathfrak{f}$, dan bestaat er precies één homomorfisme $\mathfrak{h}: \mathfrak{C} \to \mathfrak{B}$ met de eigenschap dat $\mathfrak{h} \circ \mathfrak{g} = \mathfrak{f}$.

Bewijs. Laat $\theta = \text{Ker}\mathfrak{g}$. Zij $\mathfrak{i} = \langle \mathfrak{i}_s : c \mapsto \mathfrak{g}_s^{-1}\{c\} \rangle_{s \in S\Sigma}$ het kanonieke isomorfisme van $\mathfrak{C}$ naar $\mathfrak{A}/\theta$, $\mathfrak{k}$ het unieke homomorfisme van $\mathfrak{A}/\theta$ naar $\mathfrak{B}$ dat voldoet aan $\mathfrak{k} \circ \mathfrak{h}_\theta = \mathfrak{f}$, en $\mathfrak{h} = \mathfrak{k} \circ \mathfrak{i}$.

$$\begin{tikzcd}
& \mathfrak{A} \arrow[dl, "\mathfrak{g}"'] \arrow[d, "\mathfrak{h}_\theta"] \arrow[dr, "\mathfrak{f}"] & \\
\mathfrak{C} \arrow[r, "\mathfrak{i}"'] & \mathfrak{A}/\theta \arrow[r, "\mathfrak{k}"'] & \mathfrak{B}
\end{tikzcd}$$

Als $a \in A_s$, dan $\mathfrak{i}_s \mathfrak{g}_s(a) = \mathfrak{g}_s^{-1}\{\mathfrak{g}_s(a)\}$; dus $\mathfrak{i}^{-1} \circ \mathfrak{h}_\theta = \mathfrak{g}$ en

$$\mathfrak{h} \circ \mathfrak{g} = \mathfrak{k} \circ \mathfrak{i} \circ \mathfrak{i}^{-1} \circ \mathfrak{h}_\theta = \mathfrak{k} \circ \mathfrak{h}_\theta = \mathfrak{f}.$$

Dat $\mathfrak{h}$ uniek is, volgt uit de surjectiviteit van $\mathfrak{g}$. ¶

Laat μ: $\Sigma \rightarrow \Sigma'$ een signatuurmorfisme zijn en X een verzameling variabelen voor Σ. Dan is $\mu{\cdot}X$ de Σ'-gesorteerde verzameling gedefinieerd door

$$(\mu{\cdot}X)_s = \bigcup_{\mu r = s} X_r\,.$$

(In het bijzonder $(\mu{\cdot}X)_s = \emptyset$ als $\mu^{-1}(s) = \emptyset$.) Het zij evident dat $\mu{\cdot}X$ een verzameling variabelen is voor Σ'.

We gebruiken dezelfde notatie voor vertalingen van Σ-termen:
Als t een Σ(X)-term is van soort r, dan is $\mu{\cdot}t$ de $\Sigma'(\mu{\cdot}X)$-term van soort μ(r) gedefinieerd door:
$\mu{\cdot}t = t$ als $t \in X_r$;
als $t = \mathbf{f}t_1\ldots t_n$, t_i van soort r_i $(1 \leq i \leq n)$, dan $\mu{\cdot}t =$

$$\mu(\mathbf{f})(\mu{\cdot}t_1,\ldots, \mu{\cdot}t_n).$$

We beginnen met een bijzonder geval: laat $\mathfrak{A}$ een termalgebra zijn, zeg $\mathfrak{A} = \mathfrak{T}_{\Sigma,X}$. We definiëren $\mathfrak{F}(\mathfrak{A})$ als $\mathfrak{T}_{\Sigma',\mu\cdot X}$. De voor de hand liggende bedeling aan X in $\mathfrak{F}(\mathfrak{A})|\mu$ is de inbedding ι gedefinieerd door

$$\iota_r(x) = x \in (\mu{\cdot}X)_{\mu(r)} \subseteq (\mathfrak{F}(\mathfrak{A})|\mu)(r).$$

Laat η het unieke Σ-homomorfisme $\bar{\iota}$ zijn dat ι uitbreidt.

Laat een Σ'-algebra $\mathfrak{B}$ gegeven zijn, met een homomorfisme $\mathfrak{g}$: $\mathfrak{A} \rightarrow \mathfrak{B}|\mu$. Definieer een bedeling β: $\mu{\cdot}X \rightarrow B$ door

$$\beta_{\mu(r)}(x) = \mathfrak{g}_r(x).$$

(N.B.: Als $x \in X_{r_1} \cap X_{r_2}$, dan vereist deze definitie dat $\mathfrak{g}_{r_1}(x) = \mathfrak{g}_{r_2}(x)$.)

Omdat $\mathfrak{F}(\mathfrak{A})$ volledig vrij wordt voortgebracht door $\mu{\cdot}X$, heeft β precies één uitbreiding $\mathfrak{h}$: $\mathfrak{F}(\mathfrak{A}) \rightarrow \mathfrak{B}$. Voor x $\in X_r$ geldt dat

$$(\mathfrak{h}|\mu)_r(\eta_r(x)) = \beta_{\mu(r)}(\iota_r(x)) = \mathfrak{g}_r(x).$$

Omdat $\mathfrak{A}$ volledig vrij wordt voortgebracht door X, volgt hieruit dat $(\mathfrak{h}|\mu)\circ\eta = \mathfrak{g}$.

Nu bewijzen we dat $\mathfrak{h}$ uniek is. Stel dat voor $\mathfrak{k}$: $\mathfrak{F}(\mathfrak{A}) \to \mathfrak{B}$ geldt dat $(\mathfrak{k}|\mu)\circ\eta = \mathfrak{g}$. Dan geldt voor $x \in X_r$:

$$\mathfrak{k}_{\mu(r)}(x) = (\mathfrak{k}|\mu)_r(\iota_r(x)) = ((\mathfrak{k}|\mu)\circ\eta)_r(x) = \mathfrak{g}_r(x) = \beta_{\mu(r)}(x);$$

dus $\mathfrak{k}|(\mu{\cdot}X) = \beta$, en daaruit volgt dat $\mathfrak{k} = \mathfrak{h}$. Samenvattend:

Lemma 1. Laat X een verzameling variabelen zijn voor Σ. Er bestaat een bedeling ι aan X in $\mathfrak{T}_{\Sigma',\mu\cdot X}|\mu$ zo dat voor elke $\mathfrak{B} \in Alg_{\Sigma'}$, voor elk homomorfisme $\mathfrak{g}: \mathfrak{A} \to \mathfrak{B}|\mu$, precies één homomorfisme $\mathfrak{h}$: $\mathfrak{T}_{\Sigma',\mu\cdot X} \to \mathfrak{B}$ bestaat met de eigenschap dat $(\mathfrak{h}|\mu)\circ\bar{\iota} = \mathfrak{g}$.

Lemma 2. Zij ι de bedeling aan X in $\mathfrak{T}_{\Sigma',\mu\cdot X}|\mu$ gedefinieerd door $\iota_r(x) = x$, voor alle $r \in S\Sigma$ en $x \in X_r$. Voor alle $r \in S\Sigma$ en $t \in (T_{\Sigma,X})_r$, $\bar{\iota}_r(t) = \mu{\cdot}t$.

Bewijs. Inductie naar t. Als $t \in X_r$, dan $\bar{\iota}_r(t) = t = \mu{\cdot}t$. Laat $t = \mathbf{f}t_1\ldots t_n$, met t_i van soort s_i $(1 \le i \le n)$ en $\mathbf{f} = \langle f, r, s_1\ldots s_n\rangle$. Dan

$\bar{\iota}_r(t) = (\mathfrak{T}_{\Sigma',\mu\cdot X}|\mu)(\mathbf{f})(\bar{\iota}_{s_1}(t_1),\ldots,\bar{\iota}_{s_n}(t_n))$ omdat $\bar{\iota}$ een homomorfisme is

$= (\mathfrak{T}_{\Sigma',\mu\cdot X})(\mu(\mathbf{f}))(\mu{\cdot}t_1,\ldots,\mu{\cdot}t_n)$ volgens inductiehypothese

$= \mu(\mathbf{f})(\mu{\cdot}t_1,\ldots,\mu{\cdot}t_n)$ per definitie van $(\mathfrak{T}_{\Sigma',\mu\cdot X})(\mu(\mathbf{f}))$

$= \mu{\cdot}t$. ¶

Laat A een willekeurige Σ-gesorteerde verzameling zijn. Als V een verzameling variabelen is voor Σ, soortsgewijs in 1-1-correspondentie met A (dus $V \cong A$, als $S\Sigma$-algebra's), dan is de termalgebra $\mathfrak{T}_{\Sigma,}V$ via de correspondentie $V \cong A$ op te vatten als een algebra van $\Sigma(A)$-termen. We gebruiken daarom de notatie $\mathfrak{T}_{\Sigma,A}$, voor een willekeurige termalgebra $\mathfrak{T}_{\Sigma,\mathrm{V}}$ met $\mathrm{V} \cong A$. We maken het onszelf daarbij gemakkelijk door te doen alsof $\mathrm{V} = A$.

Stelling 2. Laat $\mu\colon \Sigma \to \Sigma'$ een signatuurmorfisme zijn. Bij elke Σ-algebra $\mathfrak{A}$ bestaan een Σ'-algebra $\mathfrak{F}(\mathfrak{A})$ en een homomorfisme $\eta\colon \mathfrak{A} \to \mathfrak{F}(\mathfrak{A})|\mu$ zo dat $\langle \mathfrak{F}(\mathfrak{A}), \eta \rangle$ universeel is van $\mathfrak{A}$ naar $|\mu$.

Bewijs. Laat μ en $\mathfrak{A}$ gegeven zijn; laat ζ de kern zijn van het homomorfisme $\overline{1_A}\colon \mathfrak{T}_{\Sigma,A} \twoheadrightarrow \mathfrak{A}$. We definiëren θ als de kleinste congruentie van $\mathfrak{T}_{\Sigma',\mu\cdot A}$ zo dat voor elke $\mathrm{r} \in S\Sigma$

$$\{\langle \mu\cdot t, \mu\cdot u \rangle | \; t \equiv_{\mathrm{r}} u \; (\zeta)\} \subseteq \theta_{\mu(\mathrm{r})} ,$$

en $\mathfrak{F}(\mathfrak{A})$ als $\mathfrak{T}_{\Sigma',\mu\cdot A}/\theta$.

Stel dat $t \equiv_{\mathrm{r}} u(\zeta)$. Uit lemma 2 volgt dat $\bar{\iota}_{\mathrm{r}}(t) \equiv \bar{\iota}_{\mathrm{r}}(u)(\theta)$. Dus

$$\mathrm{Ker}\,\overline{1_A} \subseteq \mathrm{Ker}((\mathfrak{h}_\theta|\mu)\circ \bar{\iota}),$$

en wegens het tweede gevolg van de homomorfiestelling is er een uniek homomorfisme $\eta\colon \mathfrak{A} \to \mathfrak{F}(\mathfrak{A})|\mu$ met de eigenschap dat $\eta \circ \overline{1_A} = (\mathfrak{h}_\theta|\mu)\circ \bar{\iota}$.

$$\begin{tikzcd}
\mathfrak{T}_{\Sigma, A} \arrow[r, "\bar{\iota}"] \arrow[d, "\overline{1_A}"'] & \mathfrak{T}_{\Sigma', \mu\cdot A}|\mu \arrow[d, "\mathfrak{h}_\theta|\mu"] \\
\mathfrak{A} \arrow[r, dashed, "\eta"] & \mathfrak{F}(\mathfrak{A})|\mu
\end{tikzcd}$$

Laat een homomorfisme $\mathfrak{g}\colon \mathfrak{A} \to \mathfrak{B}|\mu$ gegeven zijn. Uit lemma 1 volgt het bestaan van een unieke $\mathfrak{k}\colon \mathfrak{T}_{\Sigma',\mu\cdot A} \to \mathfrak{B}$ met de eigenschap dat $(\mathfrak{k}|\mu)\circ \bar{\iota} = \mathfrak{g} \circ \overline{1_A}$. Volgens de homomorfiestelling factoriseert $\mathfrak{k}$ uniek over $\mathfrak{h}_\theta$, mits $\mathrm{Ker}\,\mathfrak{k} \supseteq \theta$.

Welnu: stel dat $t \equiv_{\mathrm{r}} u$ (ζ), en $\mu(\mathrm{r}) = \mathrm{s}$. Dan $(\overline{1_A})_{\mathrm{r}}(t) = (\overline{1_A})_{\mathrm{r}}(u)$, en a fortiori $\mathfrak{g}_{\mathrm{r}}((\overline{1_A})_{\mathrm{r}}(t)) = \mathfrak{g}_{\mathrm{r}}((\overline{1_A})_{\mathrm{r}}(u))$, dus

$$\mathfrak{k}_{\mathrm{s}}(\mu\cdot t) = (\mathfrak{k}|\mu)_{\mathrm{r}}(\bar{\iota}_{\mathrm{r}}(t)) = (\mathfrak{k}|\mu)_{\mathrm{r}}(\bar{\iota}_{\mathrm{r}}(u)) = \mathfrak{k}_{\mathrm{s}}(\mu\cdot u).$$

Zeg dat $\mathfrak{k} = \mathfrak{h}\circ\mathfrak{h}_\theta$.

$$\begin{tikzcd}
\mathfrak{T}_{\Sigma, A} \arrow[r, "\bar{\iota}"] \arrow[d, "\overline{1_A}"'] & \mathfrak{T}_{\Sigma', \mu\cdot A}|\mu \arrow[d, "\mathfrak{h}_\theta|\mu"'] \arrow[dd, dashed, bend left, "\mathfrak{k}|\mu"] \\
\mathfrak{A} \arrow[r, "\eta"] \arrow[dr, "\mathfrak{g}"'] & \mathbf{F}(\mathfrak{A})|\mu \arrow[d, dashed, "\mathfrak{h}|\mu"'] \\
 & \mathfrak{B}|\mu
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathfrak{T}_{\Sigma', \mu\cdot A} \arrow[d, "\mathfrak{h}_\theta"'] \arrow[dd, dashed, bend left, "\mathfrak{k}"] \\
\mathbf{F}(\mathfrak{A}) \arrow[d, dashed, "\mathfrak{h}"'] \\
\mathfrak{B}
\end{tikzcd}$$

Dan $(\mathfrak{h}|\mu)\circ\eta\circ\overline{1_A} = (\mathfrak{h}|\mu)\circ(\mathfrak{h}_\theta|\mu)\circ\bar{\iota} = (\mathfrak{k}|\mu)\circ\bar{\iota} = \mathfrak{g}\circ\overline{1_A}$, dus $(\mathfrak{h}|\mu)\circ\eta = \mathfrak{g}$ omdat $\overline{1_A}$ surjectief is. Het homomorfisme $\mathfrak{h}$ is uniek bepaald: als $(\mathfrak{h}'|\mu)\circ\eta = \mathfrak{g}$, dan $(\mathfrak{h}'|\mu)\circ\eta\circ\overline{1_A} = \mathfrak{g}\circ\overline{1_A}$, dus

$$((\mathfrak{h}'\circ\mathfrak{h}_\theta)|\mu)\circ\bar{\iota} = (\mathfrak{h}'|\mu)\circ(\mathfrak{h}_\theta|\mu)\circ\bar{\iota} = \mathfrak{g}\circ\overline{1_A},$$

dus $\mathfrak{h}'\circ\mathfrak{h}_\theta = k$ wegens uniciteit van $\mathfrak{k}$, en $\mathfrak{h}' = \mathfrak{h}$ wegens unieke factorisatie over $\mathfrak{h}_\theta$. ¶

Zij C een klasse van Σ'-algebra's; we noemen een paar $\langle \mathfrak{F}_C(\mathfrak{A}), \eta\rangle$ van een algebra $\mathfrak{F}_C(\mathfrak{A})$ in C en een homomorfisme $\eta\colon \mathfrak{A} \to \mathfrak{F}_C(\mathfrak{A})|\mu$ *universeel van* $\mathfrak{A}$ *naar* de beperking $(|\mu)|C$ van $|\mu$ tot C als voor elke $\mathfrak{C} \in C$ bij ieder Σ-homomorfisme $\mathfrak{g}\colon \mathfrak{A} \to \mathfrak{C}|\mu$ precies één Σ'-homomorfisme

$$\mathfrak{h}\colon \mathfrak{F}_C(\mathfrak{A}) \to \mathfrak{B}$$

bestaat zo dat $(\mathfrak{h}|\mu)\circ\eta = g$. Zonder de eis dat $\mathfrak{F}_C(\mathfrak{A}) \in C$ is universaliteit naar $(|\mu)|C$ een zwakkere eigenschap dan universaliteit naar $|\mu$, en zou het bestaan van universele paren naar $(|\mu)|C$ direct uit Stelling 2 volgen. In het vervolg zal onderzocht worden voor welke klassen C een universeel paar bestaat.

Variëteiten; de Stelling van Birkhoff

We noemen algebra's *vergelijkbaar* als ze dezelfde signatuur hebben.

Zij Σ een signatuur. Een (open) Σ-vergelijking is een paar $\langle t, u\rangle$ (genoteerd als $t \approx u$) van gelijksoortige Σ-termen; een *gesloten* Σ-vergelijking is een uitdrukking van de vorm

$$\forall \mathrm{X}.\ t \approx u,$$

waarin X alle variabelen bevat die voorkomen in t en u. Een bedeling α aan X in $\mathfrak{A}$ *vervult* de vergelijking $t \approx u$, notatie $\mathfrak{A} \models t \approx u\ [\alpha]$, als $\overline{\alpha}(t) = \overline{\alpha}(u)$. Een gesloten vergelijking $\forall \mathrm{X}.\varepsilon$ is waar in $\mathfrak{A}$, notatie $\mathfrak{A} \models \forall \mathrm{X}.\varepsilon$, als $\mathfrak{A} \models \varepsilon[\alpha]$ voor alle bedelingen α aan X in $\mathfrak{A}$. Als $\mathfrak{A}$ geen lege soorten heeft, dan zijn $\forall \mathrm{X}.\varepsilon$ en $\forall \mathrm{Y}.\varepsilon$ equivalent voor alle X en Y die alle variabelen bevatten die in ε voorkomen. Als echter $\mathfrak{A}(\mathrm{s}) = \emptyset$, en X bevat variabelen van soort s, dan $\mathfrak{A} \models \forall \mathrm{X}.\varepsilon$, ook als ε geen variabelen van soort X bevat.

Definitie. Zij $\mathcal{K}$ een klasse van vergelijkbare algebra's. Dan is

$\boldsymbol{H}(\mathcal{K})$ de klasse der homomorfe beelden van elementen van $\mathcal{K}$,

$\boldsymbol{I}(\mathcal{K})$ de klasse der isomorfe kopieën van elementen van $\mathcal{K}$,

$\boldsymbol{S}(\mathcal{K})$ de klasse der subalgebra's van elementen van $\mathcal{K}$,

$\boldsymbol{P}(\mathcal{K})$ de klasse der directe producten van families van elementen van $\mathcal{K}$,

$\boldsymbol{P}^+(\mathcal{K})$ de klasse der directe producten van niet-lege families van elementen van $\mathcal{K}$.

Verder definiëren we $\boldsymbol{E}(\mathcal{K})$ als de modelklasse van de gesloten equationele theorie van $\mathcal{K}$. Voor een verzameling Φ van gesloten vergelijkingen van signatuur Σ,

$$Mod_\Sigma(\Phi) = \{\mathfrak{A} \in Alg_\Sigma | \; \forall\varphi \in \Phi: \mathfrak{A} \models \varphi\}.$$

Birkhoff bewees dat voor elke klasse $\mathcal{K}$ van vergelijkbare eensoortige algebra's $\boldsymbol{E}(\mathcal{K}) = \boldsymbol{HSP}(\mathcal{K})$.[1] Het onderstaande voorbeeld illustreert deze stelling voor meersoortige algebra's met lege soorten.

Voorbeeld. Laat de signatuur Σ bestaan uit twee soortnamen r en s. Laat $\mathfrak{A}$ een algebra zijn waarin soort r twee elementen heeft en s geen, terwijl in $\mathfrak{B}$ juist s twee elementen heeft, en r geen. Laat u, v variabelen zijn van soort r, en x, y van soort s. De (gesloten) equationele theorie van $\{\mathfrak{A}, \mathfrak{B}\}$ wordt geaxiomatiseerd door

[1] Lege algebra's worden vaak buiten beschouwing gelaten, maar voor dit resultaat is dat niet nodig. De equationele theorie van de lege algebra $\boldsymbol{\emptyset}$ van een signatuur zonder constantenamen is gelijk aan die van de triviale (eenpunts) algebra $\mathbf{1}$; $\boldsymbol{\emptyset}$ is een subalgebra van elke algebra. Het maakt hier ook niet uit of je $\boldsymbol{E}(\mathcal{K})$ definieert als de modelklasse van de *open* equationele theorie of van de gesloten equationele theorie.

$$\forall uxy.\ x \approx y, \qquad \forall uvx.\ u \approx v\,.$$

Dus $\boldsymbol{E}\{\mathfrak{A}, \mathfrak{B}\}$ bevat de Σ-algebra's met lege soort s, die met lege soort r, en die waarin beide soorten precies één element hebben. Het laatste type is isomorf met $\prod\emptyset$; het tweede is $\boldsymbol{HSP}^{+}\{\mathfrak{B}\}$, het eerste $\boldsymbol{HSP}^{+}\{\mathfrak{A}\}$. En

$$\boldsymbol{HSP}\{\mathfrak{A}, \mathfrak{B}\} = \boldsymbol{I}\{\prod\emptyset\} \cup \boldsymbol{HSP}^{+}\{\mathfrak{A}\} \cup \boldsymbol{HSP}^{+}\{\mathfrak{B}\}.$$

Een open Σ-vergelijking ε is *geldig* in een Σ-algebra $\mathfrak{C}$ dan en slechts dan als $\mathfrak{C} \models \forall X.\,\alpha$, waarbij X precies de variabelen bevat die voorkomen in α. De *open* equationele theorie van een klasse $\mathcal{K}$ van Σ-algebra's is de verzameling van alle open Σ-vergelijkingen (in variabelen uit één of andere soortsgewijs oneindige verzameling) die geldig zijn in alle elementen van $\mathcal{K}$. Het is niet moeilijk in te zien dat een open $\{r, s\}$-vergelijking ε geldig is in de algebra's $\mathfrak{A}$ en $\mathfrak{B}$ dan en slechts dan als ε universeel geldig is (i.e. geldig in alle $\{r, s\}$-algebra's). De ***HSP***-stelling geldt dus *niet* voor klassen van meersoortige modellen — met mogelijk lege soorten — van open equationele theorieën.[2, 3]

Van de vier boven ingevoerde operaties op klassen van vergelijkbare algebra's is $\boldsymbol{I}$ de eenvoudigste. Een klasse $\mathcal{K}$ van vergelijkbare algebra's heet een *abstract datatype* als ze gesloten is onder isomorfie, d.w.z. als $\boldsymbol{I}(\mathcal{K}) \subseteq \mathcal{K}$. Men ziet gemakkelijk dat dan $\boldsymbol{I}(\mathcal{K}) = \mathcal{K}$; bijgevolg $\boldsymbol{II}(\mathcal{K}) = \boldsymbol{I}(\mathcal{K})$.

---

[2] Dezelfde redenering toont aan dat de ***HSP***-stelling niet opgaat voor de noties van zwakke en sterke vervulbaarheid van Manca en Salibra [MS]. *Au fond* is dat geen wonder, omdat de notie van gesloten vergelijking (afkomstig van Goguen en Meseguer) technisch gezien de enige juiste is (zie [GM0]).

[3] Higgins [H] bewijst de ***HSP***-stelling voor open equationele theorieen met behulp van het (onware) lemma dat projecties van directe producten surjectief zijn. Ehrig en Mahr [EM] gebruikten (in 1985) hetzelfde lemma.

De volgende propositie geeft een aantal van dit soort betrekkingen.

Propositie 1. Zij $\mathcal{K}$ een klasse van vergelijkbare algebra's. Dan
(i) $\boldsymbol{SH}(\mathcal{K}) \subseteq \boldsymbol{HS}(\mathcal{K})$,
(ii) $\boldsymbol{PH}(\mathcal{K}) \subseteq \boldsymbol{HP}(\mathcal{K})$,
(iii) $\boldsymbol{PS}(\mathcal{K}) \subseteq \boldsymbol{SP}(\mathcal{K})$,
(iv) $\boldsymbol{PP}(\mathcal{K}) \subseteq \boldsymbol{IP}(\mathcal{K})$,
(v) $\boldsymbol{IS}(\mathcal{K}) = \boldsymbol{SI}(\mathcal{K})$.

Bewijs.
(i) Stel $\mathfrak{A} \in \mathcal{K}$, $\mathfrak{f}$: $\mathfrak{A} \twoheadrightarrow \mathfrak{B}$, en $\mathfrak{C} \subseteq \mathfrak{B}$. Dan is $\mathfrak{f}^{-1}(C)$ het universum van een subalgebra $\mathfrak{f}^{-1}(\mathfrak{C})$ van $\mathfrak{A}$. Omdat $\mathfrak{f}$ surjectief is, moet gelden dat $C = \mathfrak{f}(\mathfrak{f}^{-1}(C))$. Dus

$$\mathfrak{A} \supseteq \mathfrak{f}^{-1}(\mathfrak{C}) \twoheadrightarrow \mathfrak{C},$$

i.e. $\mathfrak{C} \in \boldsymbol{HS}(\mathcal{K})$.
(ii) Zij $\langle \mathfrak{A}_i | i \in I \rangle$ een familie van elementen van $\mathcal{K}$, en $\langle \mathfrak{f}_i : \mathfrak{A}_i \twoheadrightarrow \mathfrak{B}_i | i \in I \rangle$ een familie van surjectieve homomorfismen. Dan is voor elke $i \in I$, $\mathfrak{f}_i \circ \mathfrak{p}_i$ een homomorfisme van $\prod_i \mathfrak{A}_i$ naar $\mathfrak{B}_i$. Laat $\mathfrak{f} = (\mathfrak{f}_i \circ \mathfrak{p}_i | \, i \in I)$, dus

$$\mathfrak{f}(a_i | \, i \in I) = \langle \mathfrak{f}_i(a_i) | \, i \in I \rangle.$$

We zijn klaar als we bewezen hebben dat $\mathfrak{f}$ surjectief is. Zij $b = \langle b_i | i \in I \rangle$ een element van $\prod_i B_{is}$. Voor elke i is er een $a_i \in A_{i,s}$ zo dat $\mathfrak{f}_{i,s}(a_i) = b_i$. Neem $a = \langle a_i | i \in I \rangle$; dan $\mathfrak{f}_s(a) = b$.
(iii) Zij $\langle \mathfrak{A}_i | i \in I \rangle$ een familie van elementen van $\mathcal{K}$, en voor elke $i \in I$, $\mathfrak{B}_i$ een subalgebra van $\mathfrak{A}_i$. Dan is $\prod_i \mathfrak{B}_i$ een subalgebra van $\prod_i \mathfrak{A}_i$. In het bijzonder is voor passende $b_1, \ldots, b_n \in \prod_i B_i$,

$$\begin{aligned}(\textstyle\prod_i \mathfrak{B}_i)(\mathbf{f})(b_1, \ldots, b_n) &= \langle \mathfrak{B}_i(\mathbf{f})(b_1(i), \ldots, b_n(i)) \rangle_{i \in I} \\ &= \langle \mathfrak{A}_i(\mathbf{f})(b_1(i), \ldots, b_n(i)) \rangle_{i \in I} \\ &= (\textstyle\prod_i \mathfrak{A}_i)(\mathbf{f})(b_1, \ldots, b_n),\end{aligned}$$

omdat $\mathfrak{B}_i \subseteq \mathfrak{A}_i$.

(iv) Laat voor elke $i \in I$, $\langle \mathfrak{A}_{ij} | j \in J_i \rangle$ een familie zijn van elementen van $\mathcal{K}$. Laat $H = \bigcup_{i \in I} J_i \times \{i\}$; en $\mathfrak{A}_{\langle j,i \rangle} = \mathfrak{A}_{ij}$. We willen bewijzen dat

$$\prod_{h \in H} \mathfrak{A}_h \cong \prod_{i \in I} \prod_{j \in J_i} \mathfrak{A}_{ij}.$$

We definiëren: $\mathfrak{A} = \prod_h \mathfrak{A}_h$, $\mathfrak{B}_i = \prod_{j \in J_i} \mathfrak{A}_{ij}$, en $\mathfrak{B} = \prod_{i \in I} \mathfrak{B}_i$.

Voor alle $i \in I$ en $j \in J_i$ is $\mathfrak{p}_{\langle j,i \rangle}$ een homomorfisme van $\mathfrak{A}$ naar $\mathfrak{A}_{ij}$; dus voor alle $i \in I$ is $\mathfrak{f}_i := (\mathfrak{p}_{\langle j,i \rangle} | j \in J_i)$ een homomorfisme van $\mathfrak{A}$ naar $\mathfrak{B}_i$. Dus $\mathfrak{f} := (\mathfrak{f}_i | i \in I)$ is een homomorfisme van $\mathfrak{A}$ naar $\mathfrak{B}$. Omgekeerd is $\mathfrak{p}_j \circ \mathfrak{p}_i$ een homomorfisme van $\mathfrak{B}$ naar $\mathfrak{A}_{ij}$, dus $\mathfrak{g} := (\mathfrak{p}_j \circ \mathfrak{p}_i | \langle j, i \rangle \in H)$ is een homomorfisme van $\mathfrak{B}$ naar $\mathfrak{A}$. Nu redeneren we als volgt. Er is precies één homomorfisme $\mathfrak{k}$ van $\mathfrak{A}$ naar $\mathfrak{A}$ met de eigenschap dat voor alle $h \in H$, $\mathfrak{p}_h \circ \mathfrak{k} = \mathfrak{p}_h$; omdat de identiteit $1_{\mathfrak{A}}$ die eigenschap heeft, moet $\mathfrak{k} = 1_{\mathfrak{A}}$. Maar $\mathfrak{p}_{\langle j,i \rangle} \circ \mathfrak{g}\mathfrak{f} = \mathfrak{p}_j \circ \mathfrak{p}_i \circ \mathfrak{f} = \mathfrak{p}_j \circ \mathfrak{f}_i = \mathfrak{p}_{\langle j,i \rangle}$, dus $\mathfrak{g}\mathfrak{f} = 1_{\mathfrak{A}}$. Analoog bewijst men dat $\mathfrak{f}\mathfrak{g} = 1_{\mathfrak{B}}$. Dus $\mathfrak{f}$ en $\mathfrak{g}$ zijn isomorfismen.

(v) ($\subseteq$) Zij $\mathfrak{B} \subseteq \mathfrak{A} \in \mathcal{K}$, en $\mathfrak{f}: \mathfrak{B} \to \mathfrak{C}$ een isomorfisme. Dan construeren we een algebra $\mathfrak{A}'$ uit $\mathfrak{A}$ en $\mathfrak{C}$ als volgt. Zij s een soortnaam in de signatuur. Neem een verzameling $X_s \cong A_s - B_s$ die disjunct is met C_s; laat

$$\mathfrak{g}_s : A_s - B_s \to X_s$$

een bijectie zijn. We definiëren $\mathfrak{A}'(s)$ als $C_s \cup X_s$. Door dit te doen voor alle soorten in de signatuur krijgen we een gesorteerde verzameling A', met een bijectie $\mathfrak{h}: A \to A'$ gedefinieerd door $\mathfrak{h}_s = \mathfrak{f}_s \cup \mathfrak{g}_s$. Nu definiëren we de operaties van $\mathfrak{A}'$: voor elke declaratie $\mathbf{f}$ en passende $a_1, \ldots, a_n$ in A,

$$\mathfrak{A}'(\mathbf{f})(\mathfrak{h}(a_1), \ldots, \mathfrak{h}(a_n)) = \mathfrak{h}(\mathfrak{A}(\mathbf{f})(a_1, \ldots, a_n)).$$

Dan $\mathfrak{h}\colon \mathfrak{A} \cong \mathfrak{A}'$, en $\mathfrak{C} \subseteq \mathfrak{A}'$.

($\supseteq$) Dit is een speciaal geval van (i): als het homomorfisme $\mathfrak{f}$ in het bewijs van (i) een isomorfisme is, dan is $\mathfrak{f}|\mathfrak{f}^{-1}(C)$ ook een isomorfisme. ¶

Een klasse $\mathcal{K}$ van algebra's heet *gesloten onder homomorfe beelden* als $\boldsymbol{H}(\mathcal{K}) \subseteq \mathcal{K}$; *gesloten onder subalgebra's* als $\boldsymbol{S}(\mathcal{K}) \subseteq \mathcal{K}$; en *gesloten onder directe producten* als $\boldsymbol{P}(\mathcal{K}) \subseteq \mathcal{K}$.

Definitie. Een *variëteit* is een klasse van vergelijkbare algebra's die gesloten is onder homomorfe beelden, subalgebra's en directe producten.

De klasse Alg_Σ van alle Σ-algebra's is een variëteit. Subvariëteiten van Alg_Σ (i.e. variëteiten $\mathcal{K} \subseteq Alg_\Sigma$) heten *variëteiten over* Σ, of Σ-*variëteiten*. Het is gemakkelijk te zien dat de doorsnede van een klasse van Σ-variëteiten opnieuw een Σ-variëteit is. Bij een gegeven klasse $\mathcal{K} \subseteq Alg_\Sigma$ bestaat daarom een *kleinste* Σ-variëteit die $\mathcal{K}$ omvat.

Stelling 3. Zij $\mathcal{K}$ een klasse van Σ-algebra's. De kleinste Σ-variëteit die $\mathcal{K}$ omvat is $\boldsymbol{HSP}(\mathcal{K})$.

Bewijs. De kleinste Σ-variëteit die $\mathcal{K}$ omvat, omvat zeker ook $\boldsymbol{HSP}(\mathcal{K})$. Het is dus voldoende te laten zien dat $\boldsymbol{HSP}(\mathcal{K})$ een variëteit is. Welnu: het is evident dat $\boldsymbol{H}(\boldsymbol{HSP}(\mathcal{K})) = \boldsymbol{HSP}(\mathcal{K})$. Volgens onderdeel (i) van de propositie,

$$\boldsymbol{S}(\boldsymbol{HSP}(\mathcal{K})) \subseteq \boldsymbol{HSSP}(\mathcal{K}) = \boldsymbol{HSP}(\mathcal{K}).$$

Volgens (ii)-(v) van de propositie,

$$\boldsymbol{P(HSP(\mathcal{K}))} \subseteq \boldsymbol{HPSP(\mathcal{K})} \subseteq \boldsymbol{HSPP(\mathcal{K})} \subseteq \boldsymbol{HSIP(\mathcal{K})}$$
$$= \boldsymbol{HISP(\mathcal{K})} = \boldsymbol{HSP(\mathcal{K})}. \qquad ¶$$

Lemma 3. Zij $\mu\colon \Sigma \to \Sigma'$ een signatuurmorfisme, en $\langle \mathfrak{A}_i \mid i \in I \rangle$ een familie van Σ'-algebra's. Dan

$$\Big(\prod_{i\in I} \mathfrak{A}_i\Big)|\mu = \prod_{i\in I} (\mathfrak{A}_i|\mu),$$

en als $\mathfrak{p}_j$ en $\mathfrak{r}_j$ de projecties zijn van, respectievelijk, $\prod \mathfrak{A}_i$ en $\prod(\mathfrak{A}_i|\mu)$ op hun j-coördinaat, dan $\mathfrak{r}_j = \mathfrak{p}_j|\mu$.

Stelling 4. Zij $\mu\colon \Sigma \to \Sigma'$ een signatuurmorfisme, $\mathfrak{A}$ een Σ-algebra, en $\mathcal{K}$ een klasse van Σ-algebra's waarvoor geldt dat $\boldsymbol{S}(\mathcal{K}) \subseteq \mathcal{K}$ en $\boldsymbol{P}(\mathcal{K}) \subseteq \mathcal{K}$. Dan bestaat er een universeel paar $\langle \mathfrak{f}, \gamma \rangle$ van $\mathfrak{A}$ naar $(|\mu)|\mathcal{K}$.

Birkhoff [Bi35] bewees deze stelling voor eensoortige algebra's en inbeddingen $S\Sigma \hookrightarrow \Sigma$. We zeggen dat $\mathfrak{f}$ *vrij* is *voor* $\mathcal{K}$ *over* γ (of over $\mathfrak{A}$, als γ de identieke inbedding is) en dat $\mathfrak{f}$ *vrij* wordt *voortgebracht* door γ (of $\mathfrak{A}$). We noteren $\mathfrak{f}$ als $\mathfrak{f}_{\mathcal{K}}(\gamma)$ of (als γ de identieke inbedding is) $\mathfrak{f}_{\mathcal{K}}(\mathfrak{A})$.

Bewijs. Zij G de klasse van alle homomorfismen van $\mathfrak{A}$ naar μ-reducten van elementen van $\mathcal{K}$. Beschouw $\mathfrak{g} \in G$, zeg $\mathfrak{g} : \mathfrak{A} \to \mathfrak{C}|\mu$ met $\mathfrak{C} \in \mathcal{K}$. Volgens Stelling 2 bestaat er een universeel paar $\langle \mathfrak{f}(\mathfrak{A}), \eta \rangle$ van $\mathfrak{A}$ naar $|\mu$. We hebben dus een homomorfisme $\overline{\mathfrak{g}} : \mathbf{F}(\mathfrak{A}) \to \mathfrak{C}$, uniek bepaald door zijn eigenschap dat $(\overline{\mathfrak{g}}|\mu) \circ \eta = \mathfrak{g}$. Omdat $\boldsymbol{S}(\mathcal{K}) \subseteq \mathcal{K}$, behoort $\overline{\mathfrak{g}}(\mathfrak{f}(\mathfrak{A}))$ tot $\mathcal{K}$. Uit de homomorfiestelling volgt dat $\overline{\mathfrak{g}}$ een isomorfisme

$$\mathfrak{f}(\mathfrak{A})/\mathrm{Ker}\,\overline{\mathfrak{g}} \cong \overline{\mathfrak{g}}(\mathfrak{f}(\mathfrak{A}))$$

induceert. Laat $C = \{\mathrm{Ker}\,\overline{\mathfrak{g}} \mid \mathfrak{g} \in G\} \subseteq \mathrm{Con}\,\mathfrak{f}(\mathfrak{A})$.

Kies voor elke $\theta \in C$ een afbeelding $\mathfrak{g}_\theta \in G$ zo dat $\theta = \mathrm{Ker}\ \overline{\mathfrak{g}_\theta}$; definieer $\mathfrak{B}_\theta$ als $\overline{\mathfrak{g}_\theta}(\mathfrak{F}(\mathfrak{A}))$. Dan $\mathfrak{B}_\theta \in \mathcal{K}$. Definieer:

$$\mathfrak{k} = \prod_{\theta \in C} \mathfrak{B}_\theta .$$

Dan $\mathfrak{k} \in \mathcal{K}$ omdat $\mathcal{K}$ gesloten is onder directe producten. Nu gebruiken we Lemma 3: laat

$$\gamma = (\mathfrak{g}_\theta)_{\theta \in C}\colon \mathfrak{A} \to \mathfrak{k}|\mu ,$$

zodat $(\mathfrak{p}_\theta|\mu)\circ\gamma = \mathfrak{g}_\theta$. Definieer $\overline{\gamma}\colon \mathfrak{F}(\mathfrak{A}) \to \mathfrak{k}$ door de conditie dat $(\overline{\gamma}|\mu)\circ\eta = \gamma$; laat $\mathfrak{F} = \overline{\gamma}(\mathfrak{F}(\mathfrak{A}))$. Dan $\mathfrak{F} \in \mathcal{K}$ omdat $\mathcal{K}$ gesloten is onder subalgebra's. We zullen laten zien dat $\langle \mathfrak{F}, \gamma \rangle$ universeel is van $\mathfrak{A}$ naar $(|\mu)|\mathcal{K}$. Merk op dat $((\mathfrak{p}_\theta|\mu)|F)\circ\gamma = (\mathfrak{p}_\theta|\mu)\circ\gamma$.

Zij $\mathfrak{D} \in \mathcal{K}$, en $\mathfrak{g}: \mathfrak{A} \to \mathfrak{D}|\mu$. We zoeken een homomorfisme $\hat{\mathfrak{g}}\colon \mathfrak{F} \to \mathfrak{D}$ zo dat $\mathfrak{g} = (\hat{\mathfrak{g}}|\mu)\circ\gamma$. Neem weer de uitbreiding $\overline{\mathfrak{g}}$ van $\mathfrak{g}$ over $\mathfrak{F}(\mathfrak{A})$, en laat $\theta = \mathrm{Ker}\,\overline{\mathfrak{g}}$. Dan $\overline{\mathfrak{g}}(\mathfrak{F}(\mathfrak{A})) \cong \mathfrak{F}(\mathfrak{A})/\theta \cong \mathfrak{B}_\theta$, met, voor $\mathrm{r} \in S\Sigma$ en $a \in A_\mathrm{r}$,

$$\mathfrak{g}_\mathrm{r}(a) \mapsto \eta_\mathrm{r}(a)/\theta_{\mu(r)} \mapsto (\mathfrak{g}_\theta)_\mathrm{r}(a) .$$

Zij $\phi\colon \mathfrak{B}_\Theta \to \overline{\mathfrak{g}}(\mathfrak{F}(\mathfrak{A}))$ een isomorfisme dat telkens $(\mathfrak{g}_\theta)_\mathrm{r}(a)$ afbeeldt naar $\mathfrak{g}_\mathrm{r}(a)$; dus $(\phi|\mu)\circ\mathfrak{g}_\theta = \mathfrak{g}$. Dan is $\phi\circ(\mathfrak{p}_\theta|F)$ de gevraagde $\hat{\mathfrak{g}}$:

$$((\phi\circ(\mathfrak{p}_\theta|F))|\mu)\circ\gamma = (\phi|\mu)\circ(\mathfrak{p}_\theta|\mu)\circ\gamma = (\phi|\mu)\circ\mathfrak{g}_\theta = \mathfrak{g}.$$

Tenslotte bewijzen we dat $\hat{\mathfrak{g}}$ door de gelijkheid

$$\mathfrak{g} = (\hat{\mathfrak{g}}|\mu)\circ\gamma$$

uniek wordt bepaald. Stel dat voor $\mathfrak{h}\colon \mathfrak{F} \to \mathfrak{D}$ geldt dat $\mathfrak{g} = (\mathfrak{h}|\mu)\circ\gamma$. Dan

$$\begin{aligned}((\phi^{-1}\circ\mathfrak{h}\circ\overline{\gamma})|\mu)\circ\eta &= (\phi^{-1}|\mu)\circ(\mathfrak{h}|\mu)\circ\gamma = (\phi^{-1}|\mu)\circ\mathfrak{g} = \mathfrak{g}_\theta \\ &= (\mathfrak{p}_\theta|\mu)\circ(\overline{\gamma}|\mu)\circ\eta.\end{aligned}$$

Wegens unieke factorisatie over η, $\phi^{-1}\circ\mathfrak{h}\circ\bar{\gamma} = \mathfrak{p}_\theta\circ\bar{\gamma}$. Maar $\bar{\gamma}$ is surjectief naar F, dus $\phi^{-1}\circ\mathfrak{h} = \mathfrak{p}_\theta|F$, en $\mathfrak{h} = \phi\circ(\mathfrak{p}_\theta|F)$ $= \hat{\mathfrak{g}}$. ¶

Lemma 4. Zij Σ een signatuur, X een verzameling variabelen voor Σ; laat $t, u \in T_{\Sigma,\mathrm{X}}$, $\mathfrak{F} \in \mathcal{K} \subseteq Alg_\Sigma$, en γ: $\mathrm{X} \to F$. Als $\mathfrak{F}$ vrij is voor $\mathcal{K}$ over γ, en $\mathfrak{F} \models t \approx u[\gamma]$, dan $\mathcal{K} \models \forall \mathrm{X}.t \approx u$.

Bewijs. Zij $\mathfrak{A} \in \mathcal{K}$, $\alpha : \mathrm{X} \to A$ een bedeling. We moeten bewijzen dat voor t, u van soort r,

$$(*) \qquad \bar{\alpha}_\mathrm{r}(t) = \bar{\alpha}_\mathrm{r}(u).$$

Er is een homomorfisme $\hat{\alpha}: \mathfrak{F} \to \mathfrak{A}$ zo dat $\hat{\alpha}\circ\gamma = \alpha$. Laat $\bar{\gamma}$ de uitbreiding zijn van γ over $\mathfrak{T}_{\Sigma,\mathrm{X}}$.

$$\begin{tikzcd}
\mathfrak{T} \arrow[r, "\bar{\gamma}"] \arrow[dr, "\bar{\alpha}"'] & \mathfrak{F} \arrow[d, "\hat{\alpha}"] \\
 & \mathfrak{A}
\end{tikzcd}$$

Er moet gelden dat $\bar{\alpha} = \hat{\alpha}\circ\bar{\gamma}$, wegens uniciteit van $\bar{\alpha}$, omdat, voor $\mathrm{s} \in \mathrm{S}\Sigma$, $\hat{\alpha}_\mathrm{s}(\bar{\gamma}_\mathrm{s}(\mathrm{x})) = \hat{\alpha}_\mathrm{s}(\gamma_\mathrm{s}(\mathrm{x})) = \alpha_\mathrm{s}(\mathrm{x})$. Maar $\bar{\gamma}_\mathrm{r}(t) = \bar{\gamma}_\mathrm{r}(u)$; dus

$$\bar{\alpha}_\mathrm{r}(t) = \hat{\alpha}_\mathrm{r}(\bar{\gamma}_\mathrm{r}(t)) = \hat{\alpha}_\mathrm{r}(\bar{\gamma}_\mathrm{r}(u)) = \bar{\alpha}_\mathrm{r}(u). \quad ¶$$

Stelling 5 (cf. [Bi35]). Zij Σ een signatuur. Voor elke variëteit $\mathcal{K}$ van Σ-algebra's is er een verzameling Φ van gesloten vergelijkingen van signatuur Σ zo dat $\mathcal{K} = Mod_\Sigma(\Phi)$.

Bewijs. Neem een verzameling X van variabelen voor Σ waarin elke soort oneindig is. Laat Φ de verzameling zijn van alle gesloten Σ-gelijkheden van de vorm

$$\forall \mathrm{Y}.\, t \approx u$$

met $\mathrm{Y} \subseteq \mathrm{X}$ die geldig zijn in $\mathcal{K}$. Dan is duidelijk dat $\mathcal{K} \subseteq Mod_{\Sigma}(\Phi)$.

Omgekeerd, laat $\mathfrak{B} \in Alg_{\Sigma}$ een model zijn van Φ; we moeten bewijzen dat $\mathfrak{B} \in \mathcal{K}$. Uit Stelling 4 volgt dat $\mathcal{K}$ een algebra $\mathfrak{F}$ bevat zo dat voor zekere $\gamma: B \to F$ het paar $\langle \mathfrak{F}, \gamma \rangle$ universeel is van B naar $(|\mathrm{S}\Sigma)|\mathcal{K}$. Laat $\mathfrak{T} = \mathfrak{T}_{\Sigma,B}$, en laat $\overline{1_B}: \mathfrak{T} \to \mathfrak{B}$ en $\bar{\gamma}: \mathfrak{T} \to \mathfrak{F}$ de homomorfismen zijn die respectievelijk 1_B en γ uitbreiden. Stel dat voor $t, u \in T_{\mathrm{s}}$, $\bar{\gamma}_{\mathrm{s}}(t) = \bar{\gamma}_{\mathrm{s}}(u)$. Dan $\mathfrak{F} \models t \approx u[\gamma]$; dus $\mathcal{K} \models \forall B.\, t \approx u$, volgens lemma 4.

Definieer nu $\mathrm{Y} \subseteq \mathrm{X}$ en een substitutie $\sigma: B \to \mathrm{Y}$ als volgt. Zij $\mathrm{r} \in \mathrm{S}\Sigma$. Als $B_{\mathrm{r}} = \emptyset$, dan ook $\mathrm{Y}_{\mathrm{r}} = \emptyset$ (en $\sigma_{\mathrm{r}} = \emptyset$). Als $B_{\mathrm{r}} \neq \emptyset$, en in t en u komen geen elementen voor van B_{r}, dan heeft Y_{r} precies één element, en σ_{r} beeldt alle elementen van B_{r} af op dat ene element. Tenslotte, als in t en u samen de (verschillende) elementen $b_1, \ldots, b_k$ van B_{r} voorkomen, dan heeft Y_{r} k elementen $\mathrm{y}_1, \ldots, \mathrm{y}_k$, en

$$\sigma_{\mathrm{r}}(b_i) = \mathrm{y}_i \; (1 \leq i \leq k),$$
$$\text{en } \sigma_{\mathrm{r}}(b) = \mathrm{y}_k \text{ voor } b \in B_{\mathrm{r}} - \{b_1, \ldots, b_k\}.$$

Uit $\forall B.\, t \approx u$ volgt $\forall \mathrm{Y}.\, t^{\sigma} \approx u^{\sigma}$, dus $\mathcal{K} \models \forall \mathrm{Y}.\, t^{\sigma} \approx u^{\sigma}$, en $\forall \mathrm{Y}.\, t^{\sigma} \approx u^{\sigma} \in \Phi$. Dus $\mathfrak{B} \models \forall \mathrm{Y}.\, t^{\sigma} \approx u^{\sigma}$. Nu is gemakkelijk te zien dat er een injectie $\tau: \mathrm{Y} \hookrightarrow B$ bestaat met de eigenschap dat $t^{\sigma\tau} = t$ en $u^{\sigma\tau} = u$; en bijgevolg

$$\mathfrak{B} \models \forall B.\, t \approx u.$$

Dan moet in het bijzonder $\mathfrak{B} \models t \approx u[1_B]$, i.e. $(\overline{1_B})_s(t) = (\overline{1_B})_s(u)$. We concluderen dat $\mathrm{Ker}\,\bar{\gamma} \subseteq \mathrm{Ker}\,\overline{1_B}$.

Nu volgt uit de homomorfiestelling dat $\overline{1_B}$ factoriseert over $\bar{\gamma}$, en dus, omdat $\overline{1_B}$ surjectief is, $\mathfrak{B} \in \boldsymbol{H}(\mathcal{K}) \subseteq \mathcal{K}$.¶

LITERATUUR

[Bi35] Garrett Birkhoff: On the structure of abstract algebras. Proc. of the Camb. Phil. Soc. XXXI (1935), 433-54. Herdrukt in [Bi87].

[Bi87] Selected Papers on Algebra and Topology by Garrett Birkhoff. Red. G.-C. Rota & J.S. Oliveira. Birkhäuser, 1987.

[BL] Garrett Birkhoff & John D. Lipson: Heterogeneous algebras. Journal of Combinatorial Theory 8 (1970), 115-133. Herdrukt in [Bi87].

[ELM] Hartmut Ehrig, Jacques Loeckx & Bernd Mahr, Remarks on the equational calculus for many-sorted algebras with possibly empty carrier sets. Bull. of the EATCS 30 (1986) 61-6.

[EM] H. Ehrig & B. Mahr, Fundamentals of Algebraic Specification, 1 (1985) and 2 (1990), Springer-Verlag, Berlijn etc.

[GM0] Joseph A. Goguen & José Meseguer: Universal realization, persistent interconnection and implementation of abstract modules. In *Automata, Languages and Programming*, verhandelingen van het 9e colloquium, Aarhus 1982, red. M. Nielsen & E.M. Schmidt, 265-81.

[GM1] J.A. Goguen, J. Meseguer, Completeness of many-sorted equational logic, Houston Journal of Mathematics 11 (1985), 307-334.

[GM2] Joseph A. Goguen & José Meseguer: Remarks on Remarks on many-sorted equational logic. Bull. of the EATCS 30 (1986) 66-73.

[H] P.J. Higgins: Algebras with a scheme of operators. Mathematische Nachrichten 27 (1963), 115-32.

[KaR] H. Kaphengst & H. Reichel: Algebraische Algorithmentheorie. VEB Robotron, Dresden WIB, 1971.

[LEW] Jacques Loeckx, Hans-Dieter Ehrich & Markus Wolf: Specification of abstract data types. Chichester/Stuttgart, 1996.

[MS] Vincenzo Manca & Antonino Salibra: Soundness and completeness of the Birkhoff equational calculus for many-sorted algebras with possibly empty carrier sets. Theoretical Computer Science 94:101-24 (1992).

[S] Joseph Shoenfield: Mathematical Logic. Reading, Mass., 1967.

Regular Expressions and Coinduction

J.J.M.M. Rutten
CWI

Abstract

In 1995, Paul Klint was one of the Promotores of B.W. Watson, who defended his thesis entitled 'Taxonomies and toolkits of regular language algorithms'. In the present note, we show how to prove equality and inclusion of (the languages of) regular expressions using *coinduction*, a proof method from the world of coalgebras based on the notions of bisimulation and simulation, respectively. The use of coinduction is conceptually rather different from the traditional methods, which are based on the minimization of automata.

1 Deterministic automata

Let A be a (possibly infinite) set of input symbols. A *(deterministic) automaton* is a triple $S = \langle S, o, t\rangle$ consisting of a set S of *states*, an *output function* $o : S \to 2$, and a *transition function* $t : S \to S^A$. Here $2 = \{0,1\}$ and S^A is the set of all function from A to S. The output function o indicates whether a state s is *terminating* ($o(s) = 1$) or not ($o(s) = 0$). The transition function t assigns to s a function $t(s) : A \to S$, which specifies the next state $t(s)(a)$ that is reached when accepting an input symbol a. We shall sometimes write $s\downarrow$ for $o(s) = 1$, $s\uparrow$ for $o(s) = 0$, and $s \xrightarrow{a} s'$ for $t(s)(a) = s'$.

Contrary to the standard definition, both the state space S of an automaton and the set A of input symbols may be infinite. If S and A are finite then we speak of a *finite automaton*. Another difference is that our automata do not have an initial state.

A *bisimulation* between two automata $S = \langle S, o, t\rangle$ and $S' = \langle S', o', t'\rangle$ is any relation $R \subseteq S \times S'$ with, for all s in S, s' in S', and a in A:

$$\text{if } s\,R\,s' \text{ then } \begin{cases} o(s) = o'(s') & \text{and} \\ t(s)(a)\,R\,t'(s')(a). \end{cases}$$

A bisimulation between S and itself is called a bisimulation *on* S. Unions and (relational) compositions of bisimulations are bisimulations again. We write $s \sim s'$ whenever there exists a bisimulation R with $s\,R\,s'$. This relation $\sim$ is the union of all bisimulations and, therewith, the greatest bisimulation. The greatest bisimulation on one and the same automaton, again denoted by $\sim$, is called the *bisimilarity* relation. It is an equivalence relation.

A *homomorphism* between S and S' is any function $f : S \to S'$ with, for all s in S, $o(s) = o'(f(s))$ and, for all a in A, $f(t(s)(a)) = t'(f(s))(a)$.

An automaton $S' = \langle S', o', t' \rangle$ is a *subautomaton* of $S = \langle S, o, t \rangle$ if $S' \subseteq S$ and the inclusion $i : S' \to S$ is a homomorphism. Given $\langle S, o, t \rangle$ and S', the functions o' and t' in that case are uniquely determined. For a state s in S, $\langle s \rangle$ denotes the subautomaton *generated* by s: it is the smallest subautomaton of S containing s, and can be obtained by including all states from S that are reachable via a finite number of transitions from s.

Homomorphisms map subautomata to subautomata: for a homomorphism $f : S \to T$ and subautomaton $S' \subseteq S$, $f(S)$ is a subautomaton of T. For s in S, moreover, $f(\langle s \rangle) = \langle f(s) \rangle$.

The notions of automaton, homomorphism and bisimulation are closely related: a function $f : S \to S'$ is a homomorphism if and only if $\{\langle s, f(s) \rangle \mid s \in S\}$ is a bisimulation. And bisimulations are themselves automata: if R is a bisimulation between S and S', then $o_R : R \to 2$ and $t_R : R \to R^A$, given for $\langle s, s' \rangle$ in R and a in A by $o_R(\langle s, s' \rangle) = o(s) = o'(s')$ and $t_R(\langle s, s' \rangle)(a) = \langle t(s)(a), t'(s')(a) \rangle$, define an automaton $\langle R, o_R, t_R \rangle$.

For an example, let $A = \{a, b\}$ and consider the automata $S = \{s_1, s_2, s_3\}$ and $T = \{t_1, t_2\}$, with transitions and termination as specified by the following tables:

$S \setminus A$	a	b	
s_1	s_2	s_3	$\uparrow$
s_2	s_2	s_3	$\downarrow$
s_3	s_2	s_3	$\downarrow$

$T \setminus A$	a	b	
t_1	t_2	t_3	$\uparrow$
t_2	t_2	t_3	$\downarrow$

where, for instance, the second row of the first table denotes $s_2 \xrightarrow{a} s_2$, $s_2 \xrightarrow{b} s_3$, and $s_2\downarrow$. Then $\{\langle s_1, s_1 \rangle, \langle s_2, s_2 \rangle, \langle s_3, s_3 \rangle\}$ and $\{\langle s_2, s_3 \rangle, \langle s_2, s_2 \rangle, \langle s_3, s_3 \rangle\}$ are bisimulations on S; $\{s_2, s_3\} = \langle s_2 \rangle = \langle s_3 \rangle$ is a subautomaton of S; and $f : S \to T$ mapping s_1 to t_2, and s_2 and s_3 to t_2 is a homomorphism.

2 Languages

Let A^* be the set of all finite words over A. Prefixing a word w in A^* with an input symbol a in A is denoted by aw. Concatenation of words w and w' is denoted by ww'. Let ϵ denote the empty word. A *language* is any subset of A^*. The language *accepted* by a state s of an automaton $S = \langle S, o, t \rangle$ is $l_S(s) = \{a_1 \cdots a_n \mid s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n\downarrow\}$, where $s_1 = t(s)(a_1)$ and $s_{i+1} = t(s_i)(a_i)$, for $1 < i < n$.

Let $\mathcal{L} = \{L \mid L \subseteq A^*\}$ be the set of all languages. For a word w in A^*, the *w-derivative* of a language L is $L_w = \{v \in A^* \mid wv \in L\}$. A special case is the *a-derivative* $L_a = \{v \in A^* \mid av \in L\}$, for a in A, which can be used to turn the set $\mathcal{L}$ into an automaton $\langle \mathcal{L}, o_{\mathcal{L}}, t_{\mathcal{L}} \rangle$, defined, for $L \in \mathcal{L}$ and $a \in A$, by

$$o_{\mathcal{L}}(L) = \begin{cases} 1 & \epsilon \in L \\ 0 & \epsilon \notin L \end{cases} \quad \text{and} \quad t_{\mathcal{L}}(L)(a) = L_a.$$

That is,

$$L\downarrow \text{ iff } \epsilon \in L \quad \text{and} \quad L \xrightarrow{a} L' \text{ iff } L' = L_a.$$

This automaton has the pleasing property that the language accepted by a state L in $\mathcal{L}$ is precisely L itself. This can be proved formally, but is here only illustrated by the following example. For $L = \{a, ab, ac\}$, there are the following transitions:

$$\{a, ab, ac\} \xrightarrow{a} \{\epsilon, b, c\}\downarrow \xrightarrow{b,c} \{\epsilon\}\downarrow,$$

where we have omitted transitions leading to the empty set. It follows that $l_{\mathcal{L}}(L) = L$.

If the *behaviour* of a state is the language it accepts, then states in $\mathcal{L}$ could be said to 'do as they are'. For them, in other words, 'being is doing'.

3 Coinduction

The automaton $\mathcal{L} = \langle \mathcal{L}, o_{\mathcal{L}}, t_{\mathcal{L}} \rangle$ satisfies the following *coinduction proof principle*: for all languages K and L,

$$\text{if } K \sim L \text{ then } K = L.$$

(The converse trivially holds.) The proof principle says that in order to prove the equality of languages K and L, it is sufficient to establish the existence of a bisimulation relation on $\mathcal{L}$ that includes the pair $\langle K, L \rangle$. The principle follows from the fact that for all words w in A^* of length n and for all languages K and L with $K \sim L$: if $w \in K$ then $w \in L$, which we show next by *induction* on n. First note that a bisimulation on $\mathcal{L}$ is any relation R such that for all K and L with $K\,R\,L$, $K\downarrow$ iff $L\downarrow$, and for any a in A, $K_a\,R\,L_a$. Now consider K and L with $K \sim L$. Because $\sim$ is a bisimulation, $\epsilon \in K$ implies $\epsilon \in L$. Next consider a word $w = aw'$, of length $n+1$, in K. Because $K \sim L$ also $K_a \sim L_a$. Because $w' \in K_a$ and the length of w' is n, it follows from the inductive hypothesis that $w' \in L_a$. Thus $w \in L$. This shows that $K \sim L$ implies $K \subseteq L$. Since $K \sim L$ implies $L \sim K$, also $L \subseteq K$.

4 Regular expressions

Let the set $\mathcal{R}$ of *regular expressions* be given by the following syntax:

$$E ::= 0 \mid 1 \mid a \in A \mid E + F \mid EF \mid E^*$$

Let the funcion $\lambda : \mathcal{R} \to \mathcal{L}$, which assigns to an expression E the language $\lambda(E)$ it represents, be defined by induction on the structure of E:

$$\begin{aligned}
\lambda(0) &= \emptyset \\
\lambda(1) &= \{\epsilon\} \\
\lambda(a) &= \{a\} \\
\lambda(E+F) &= \lambda(E) + \lambda(F) \\
\lambda(EF) &= \lambda(E)\lambda(F) \\
\lambda(E^*) &= \lambda(E)^*,
\end{aligned}$$

where on the right hand side of these equations the following so-called *regular operators* are used: for languages K and L,

$$\begin{aligned} K+L &= K \cup L \\ KL &= \{vw \mid v \in K \text{ and } w \in L\} \\ K^* &= \bigcup_{n \geq 0} K^n, \end{aligned}$$

with $K^0 = \{\epsilon\}$ and $K^{n+1} = KK^n$. Languages $L = \lambda(E)$ are called *regular languages*.

The following rules for calculating the a-derivative L_a of a language L are easily verified:

$$\begin{aligned} \emptyset_a &= \emptyset \\ \{\epsilon\}_a &= \emptyset \\ \{b\}_a &= \begin{cases} 1 & \text{if } b = a \\ 0 & \text{if } b \neq a \end{cases} \\ (K+L)_a &= K_a + L_a \\ (KL)_a &= \begin{cases} K_a L & \text{if } K\uparrow \\ K_a L + L_a & \text{if } K\downarrow \end{cases} \\ (K^*)_a &= K_a K^* \end{aligned}$$

There are also the following rules for termination: $0\uparrow$, $1\downarrow$, $a\uparrow$, $K+L\downarrow$ iff $K\downarrow$ or $L\downarrow$, $KL\downarrow$ iff $K\downarrow$ and $L\downarrow$, $K^*\downarrow$. All these rules will be of great help when proving the equality of languages by means of coinduction, as we shall see below.

5 Proofs by coinduction

The use of coinduction is illustrated by first proving some of the familiar laws for the regular operators, and next some equalities of concrete expressions. Note that the completeness of these laws in not the issue here. They merely serve as examples, and some of them will be used as lemma's in subsequent proofs.

The strength of the coinduction proof principle is that it works for any valid equality, and that it always works in the same way: first define a relation consisting of the pair(s) of languages that you want to prove equal; then look at all possible transitions and continue to add pairs of resulting languages if they were not present yet. The original equality holds if and only if this process yields a bisimulation. If the languages with which one starts are regular, then the construction terminates in finitely many steps, essentially because of Kleene's theorem.

Some laws

All the familiar laws for regular expressions can be proved by coinduction. Some of them are easily proved directly on the basis of the definitions of the regular operators, others are less straightforward. Below some of the following will be proved by coinduction:

$$\begin{array}{rcll}
K+0 & = & K & (1)\\
K+K & = & K & (2)\\
K+L & = & L+K & (3)\\
(K+L)+M & = & K+(L+M) & (4)\\
1K & = & K & (5)\\
K1 & = & K & (6)\\
K0 & = & 0 & (7)\\
0K & = & 0 & (8)\\
(KL)M & = & K(LM) & (9)\\
1+LL^* & = & L^* & (10)\\
K(L+M) & = & KL+KM & (11)\\
(L+M)K & = & LK+MK & (12)\\
L\uparrow \wedge (K=LK+M) & \Rightarrow & K=L^*M & (13)\\
(K+L)^* & = & K^*(LK^*)^* & (14)\\
(K+L)^* & = & (K^*L)^*K^* & (15)
\end{array}$$

As a consequence of (4) and (9), brackets can often be omitted.

Although all of (1)–(9) are immediate from the definitions, we prove as an example equation (1) by coinduction. We show that

$$\{\langle K+0,\, K\rangle \mid K \in \mathcal{L}\}$$

is a bisimulation. Then (1) follows by coinduction. First note that $(K+0)\downarrow$ if and only if $K\downarrow$. And for any a in A,

$$\begin{array}{ll}
(K+0)_a & \\
= & K_a + 0_a\\
= & K_a + 0\\
R & K_a.
\end{array}$$

Laws (2)-(9) can be proved similarly. Equality (10) follows by coinduction from the fact that

$$\{\langle 1+LL^*,\, L^*\rangle \mid L \in \mathcal{L}\} \cup \{\langle L, L\rangle \mid L \in \mathcal{L}\}$$

is a bisimulation. For (11), one could try to prove that the relation $\{\langle K(L+M),\, KL+KM\rangle \mid K,L,M \in \mathcal{L}\}$ is a bisimulation. It turns out to be convenient to consider the (by (1)) larger set

$$R = \{\langle K(L+M)+N,\, KL+KM+N\rangle \mid K,L,M,N \in \mathcal{L}\}$$

instead. (Cf. the strengthening of the inductive hypothesis in an inductive argument.) We show that R is a bisimulation. Consider a in A and a pair $\langle K(L+M)+N,\, KL+KM+N\rangle$ in R. First note that $K(L+M)+N$

terminates if and only if $KL+KM+N$ does. Suppose that $K\downarrow$ (the case that $K\uparrow$ is similar and a little easier). Then

$$\begin{array}{cll} & (K(L+M)+N)_a & \\ = & K_a(L+M)+L_a+M_a+N_a & \\ R & K_aL+K_aM+L_a+M_a+N_a & \\ = & K_aL+L_a+K_aM+M_a+N_a & \text{[by (3) and (4)]} \\ = & (KL)_a+(KM)_a+N_a & \\ = & (KL+KM+N)_a, & \end{array}$$

which concludes the proof that R is a bisimulation. Now (11) follows by coinduction. Similarly for (12). For (13), let K, L, and M be expressions with $L\uparrow$ and $K = LK + M$. Then $K = L^*M$ follows by coinduction from the fact that $\{\langle UK+V,\, UL^*M+V\rangle \mid U,V \in \mathcal{L}\}$ is a bisimulation on $\mathcal{L}$. Equations (14) and (15) follow from the fact that $\{\langle M(K+L)^*,\, MK^*(LK^*)^*\rangle \mid K,L,M \in \mathcal{L}\}$ and $\{\langle M(K+L)^*,\, M(K^*L)^*K^*\rangle \mid K,L,M \in \mathcal{L}\}$ are bisimulations.

Some regular languages

Below the language $\lambda(E)$ of a regular expression E will be simply denoted by E itself. Similarly, E_a denotes $\lambda(E)_a$. Let $A = \{a,b\}$. We want to show

$$[(b^*a)^*ab^*]^* \quad = \quad 1+a(a+b)^*+(a+b)^*aa(a+b)^*. \qquad (16)$$

Let $E_1 = [(b^*a)^*ab^*]^*$ and $F_1 = 1+a(a+b)^*+(a+b)^*aa(a+b)^*$. Using the calculation rules for a-derivatives of Section 4, the following tables are easily computed:

	a	b	
E_1	E_2	E_4	$\downarrow$
E_2	E_2	E_3	$\downarrow$
E_3	E_2	E_3	$\downarrow$
E_4	E_5	E_4	$\uparrow$
E_5	E_2	E_4	$\uparrow$

	a	b	
F_1	F_2	F_4	$\downarrow$
F_2	F_2	F_3	$\downarrow$
F_3	F_2	F_3	$\downarrow$
F_4	F_5	F_4	$\uparrow$
F_5	F_2	F_4	$\uparrow$

where

$$\begin{array}{rcl} E_2 & = & [(b^*a)^*ab^*+b^*]E_1, \\ E_3 & = & [(b^*a)(b^*a)^*ab^*+b^*]E_1, \\ E_4 & = & [(b^*a)(b^*a)^*ab^*]E_1, \\ E_5 & = & [(b^*a)^*ab^*]E_1, \\ F_2 & = & (a+b)^*+(a+b)^*aa(a+b)^*+a(a+b)^*, \\ F_3 & = & (a+b)^*+(a+b)^*aa(a+b)^*, \\ F_4 & = & (a+b)^*aa(a+b)^*, \\ F_5 & = & (a+b)^*aa(a+b)^*+a(a+b)^*. \end{array}$$

As a consequence, $T = \{\langle E_i, F_i\rangle \mid 1 \leq i \leq 5\}$ is a bisimulation. Hence $E_i = F_i$, by coinduction, for $1 \leq i \leq 5$. This proves (16).

It follows from the tables above that $\{\langle E_2, E_3\rangle, \langle E_2, E_2\rangle, \langle E_3, E_3\rangle\}$ is a bisimulation as well. Thus $E_2 = E_3$, by coinduction, and similarly $F_2 = F_3$. There is, therefore, some redundancy in the representation of the bisimulation T, which turns out to consist of only 4 different pairs. The interesting point of this observation is that this knowledge was not needed for the conclusion above that T is a bisimulation.

Because $((a+b)^*)_a = (a+b)^*$ and $((a+b)^*)_b = (a+b)^*$ imply that $\{\langle F_2, (a+b)^*\rangle, \langle F_3, (a+b)^*\rangle\}$ is a bisimulation, we also have, as another example, the following equalities:

$$E_2 = E_3 = F_2 = F_3 = (a+b)^*.$$

Inequalities

The coinduction proof method is clearly also of help in proving that two languages are different. In order to prove $E_1 \neq E_2$ in the example above, it is sufficient to show that there is no bisimulation relation containing $\langle E_1, E_2\rangle$. Now the assumption that $\langle E_1, E_2\rangle$ is in some bisimulation leads to a contradiction, since $(E_1)_b = E_4$ and $(E_2)_b = E_3$, but $(E_4)\uparrow$ and $(E_3)\downarrow$.

6 Simulation

The notion of bisimulation is a special case of the more general notion of *simulation*, which will be introduced below. Simulation is used in the formulation of yet another coinduction principle on $\mathcal{L}$ which generalizes that of Section 3, and which will be used to prove language inclusions.

A *simulation* between two automata $S = \langle S, o, t\rangle$ and $S' = \langle S', o', t'\rangle$ is any relation $R \subseteq S \times S'$ with, for all s in S, s' in S', and a in A:

$$\text{if } s\,R\,s' \text{ then } \begin{cases} o(s) \leq o'(s') & \text{and} \\ t(s)(a)\; R\; t'(s')(a). \end{cases}$$

In other words, if sRs' then $s\downarrow$ implies $s'\downarrow$. A simulation between S and itself is called a simulation *on* S. Unions and (relational) compositions of simulations are simulations again. We write $s \leq s'$ whenever there exists a simulation R with $s\,R\,s'$. This relation $\leq$ is the union of all simulations and, therewith, the greatest simulation. The greatest simulation on one and the same automaton S, denoted by $\leq$ (or $\leq_S$, if the name of the automaton is relevant), is called the *similarity* relation. It is a preorder: $s \leq s$ and if $s \leq t$ and $t \leq u$ then $s \leq u$.

Clearly every bisimulation is a simulation. The converse does not hold but $s \leq t$ and $t \leq s$ imply $s \sim t$: if $s\,R\,t$ and $t\,T\,s$ for two simulations R and T then $R \cap T^{-1}$ is a bisimulation with $s(R \cap T^{-1})t$. It follows that $\sim = \leq \cap \leq^{-1}$.

The automaton $\mathcal{L} = \langle \mathcal{L}, o_{\mathcal{L}}, t_{\mathcal{L}}\rangle$ satisfies the following proof principle, which is again called coinduction: for all languages K and L,

$$\text{if } K \leq L \text{ then } K \subseteq L.$$

(The converse trivially holds.) The proof principle says that in order to prove the inclusion of a language K in a language L, it is sufficient to establish the existence of a simulation relation R on $\mathcal{L}$ with $K\,R\,L$. Inspecting the proof of the previous coinduction principle in Section 3, we see that it contains a proof of the statement above.

The regular operations on languages can be easily shown to be monotonic with respect to $\subseteq$. For instance, if $K \subseteq K'$ and $L \subseteq L'$ then $KL \subseteq K'L'$. Also $K \subseteq L$ implies $K_a \subseteq L_a$.

The above coinduction principle is often best applied in combination with the following weakening of the notion of simulation. A simulation *up-to-similarity* on automata $S = \langle S, o, t\rangle$ and $S' = \langle S', o', t'\rangle$ is any relation $R \subseteq S \times S'$ with, for all s in S, s' in S', and a in A:

$$\text{if } sRs' \text{ then } \begin{cases} o(s) \leq o'(s') & \text{and} \\ t(s)(a)\; R_{\leq}\; t'(s')(a), & \end{cases}$$

where $R_{\leq} = \leq_S \circ R \circ \leq_{S'}$ ($\circ$ denotes composition of relations). Interestingly, if sRt for a simulation up-to-similarity R then $s \leq t$, since in that case $R_{\leq}$ is a simulation and $R \subseteq R_{\leq}$. Thus in order to prove $K \subseteq L$ it suffices to point to a simulation up-to-similarity R with $K\,R\,L$.

We treat a few examples. The following inclusions and equational implications can all be proved by coinduction:

$$KL \subseteq L \;\Rightarrow\; K^*L \subseteq L \tag{17}$$

$$LK + M \subseteq K \;\Rightarrow\; L^*M \subseteq K \tag{18}$$

$$KL \subseteq LM \;\Rightarrow\; K^*L \subseteq LM^* \tag{19}$$

For (17) consider K and L with $KL \subseteq L$. Then

$$S = \{\langle MK^*L + N,\ ML + N\rangle \mid M, N \in \mathcal{L}\}$$

is a simulation up-to-similarity: if $(MK^*L + N)\downarrow$ then $(ML + N)\downarrow$. And for a in A,

$$\begin{array}{ll} (MK^*L + N)_a & \\ = & M_aK^*L + K_aK^*L + L_a + N_a \quad [\text{supposing that } M\downarrow] \\ = & (M_a + K_a)K^*L + L_a + N_a \\ R & (M_a + K_a)L + L_a + N_a \\ = & M_aL + K_aL + L_a + N_a \\ \subseteq & M_aL + L_a + N_a \quad [KL \subseteq L \text{ implies } (KL)_a \subseteq L_a \text{ whence } K_aL + L_a \subseteq L_a] \\ = & (ML + N)_a. \end{array}$$

Thus (17) follows by coinduction. Law (18), which refines equation (13) in Section 5, and law (19) are proved similarly.

As another example, we prove the inclusion of the following regular languages:

$$[(b^*a)^*ab^*]^* \subseteq [(b^*a)^*ab^* + b^*][(b^*a)^*ab^*]^*,$$

which we recognize as E_1 and E_2 from Section 5. The inclusion follows by coinduction from the fact that we have a simulation

$$\{\langle E_1, E_2\rangle,\ \langle E_2, E_2\rangle,\ \langle E_3, E_3\rangle,\ \langle E_4, E_3\rangle,\ \langle E_5, E_2\rangle\}.$$

7 Notes and discussion

The notions and observations of Sections 1 through 3 all are instances of far more general ones, belonging to a theory called *(universal) coalgebra* [RT93, Rut96, JR97]. It is formulated in the language of categories and functors, and its main ingredients are *coalgebra*, *homomorphism*, and *bisimulation*. In the categorical language of coalgebras, our automata $\langle S, o, t\rangle$ correspond to coalgebras $\langle o, t\rangle : S \to F(S)$ of the functor F defined on sets S by $F(S) = 2 \times S^A$, and on functions correspondingly. The coalgebraic definition of bisimulation is a categorical generalization, due to Aczel and Mendler [AM89], of Park's [Par81] and Milner's [Mil80] notion of bisimulation for concurrent branching processes. This general categorical definition applies to many different examples, including nondeterministic (possibly probabilistic) transition systems, object-based systems, infinite data structures, various other types of automata, and dynamical systems. The coinduction principle of Section 3 for $\mathcal{L}$, together with the corresponding 'being is doing' characterization, applies more generally to any *final* coalgebra. Coinduction as a proof principle for greatest fixed points of monotone operators is already around for some time. For (final coalgebras of) the powerset functor, it has been introduced in [Acz88]. In [RT93], the principle is stated in its generality for arbitrary functors. Its application to languages and regular expressions in Section 5 is to the best of our knowledge new. The calculation rules for a-derivatives (Section 4) of regular combinations of languages are well-known, have been reinvented several times, and are originally due to Brzozowski [Brz64] (see also [Con71]). A well-known way of proving equality of regular expressions is to use a complete axiom system (of which the laws in Section 5 form a subset), such as given by Salomaa in [Sal66], and apply purely algebraic reasoning. The reader is invited to consult [Gin68, pp.68-69], from which the example $E_1 = F_1$ in Section 5 was taken, and convince himself of the greater complexity of that approach. The most common and practical way of proving equality of two expressions is firstly, to construct for each expression an automaton that accepts the language it represents, and secondly, to minimize both automata. The two expressions are then equal iff the two resulting automata are isomorphic. For both the construction and the minimization step many different and efficient algorithms exist (see [Wat95] for an extensive overview and comparison). This classical approach is related to the coinduction proof method by the observation, in Section 1, that bisimulations are automata themselves. Thus also a proof by coinduction consists of the construction of an automaton. Our way of constructing this 'bisimulation automaton' is essentially based on Brzozowski's algorithm, using a-derivatives, but note that only one automaton is constructed for both expressions at the same time. Another difference is that this automaton need not be minimized in order to conclude that the two expressions are equal (this was illustrated by the bisimulation T used for the proof of $E_1 = F_1$ at the end of Section 5). The question whether this can lead to (more) efficient algorithms is yet to be addressed. Simulation relations have been studied in several forms and ways. We believe the present definition in Section 6, as well as the coinduction principle based on it, to be new. The definition of simulation ***up-to-similarity*** is a straightforward variation of Milner's notion of bisimulation ***up-to-bisimilarity*** [Mil80]. Some of the laws of Section 6 have been taken from [Koz94], where a complete axiom system for equality of regular expressions is presented in terms of equational implications. Kozen's approach is emphatically algebraic, whereas ours is essentially coalgebraic.

References

[Acz88] P. Aczel. *Non-well-founded sets.* Number 14 in CSLI Lecture Notes. Center for the Study of Languages and Information, Stanford, 1988.

[AM89] P. Aczel and N. Mendler. A final coalgebra theorem. In D.H. Pitt, D.E. Ryeheard, P. Dybjer, A. M. Pitts, and A. Poigne, editors, *Proceedings category theory and computer science*, Lecture Notes in Computer Science, pages 357–365, 1989.

[Brz64] J.A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[Con71] J.H. Conway. *Regular algebra and finite machines.* Chapman and Hall, 1971.

[Gin68] A. Ginzburg. *Algebraic theory of automata.* ACM Monograph series. Academic Press, 1968.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of EATCS*, 62:222–259, 1997.

[Koz94] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110:366–390, 1994.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1980.

[Par81] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings 5th GI conference*, volume 104 of *Lecture Notes in Computer Science*, pages 15–32. Springer-Verlag, 1981.

[RT93] J.J.M.M. Rutten and D. Turi. On the foundations of final semantics: non-standard sets, metric spaces, partial orders. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop on Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 477–530, Beekbergen, June 1993. Springer-Verlag.

[Rut96] J.J.M.M. Rutten. Universal coalgebra: a theory of systems. Report CS-R9652, CWI, 1996.

[Sal66] A. Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM*, 13(1):158–169, 1966.

[Wat95] B.W. Watson. *Taxonomies and toolkits of regular language algorithms.* PhD thesis, Eindhoven University of Technology, 1995.

Evolution of COBOL software in Dutch IT Industry

Alex Sellink*, Chris Verhoef†

University of Amsterdam, Programming Research Group
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands

alex@wins.uva.nl, x@wins.uva.nl

Newbie language designer error #17:
Overgeneralizing from experience with natural languages.
English makes a terrible programming language.

Abstract

We are interested in the evolution of COBOL software in Dutch IT industry. We observe that the original design decisions of COBOL caused a style of programming that is nowadays undesirable. We give an example of a year 2000 solution that causes problems in the year 2000 due to the original design of COBOL. We briefly mention that solutions are available in the scientific literature.

Categories and Subject Description: D.2.6 [**Software Engineering**]: Programming Environments—Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring;
Additional Key Words and Phrases: Reengineering, System renovation, Interface reengineering, Control flow normalization, Language migration, COBOL.

1 Introduction

Today's Internet originates from a project initiated by the American Department of Defence. Years ago, this innovative Department was also involved in the initial phase of the development of COBOL. Various computer suppliers and users created in the late fifties the *Conference on Data System Languages*, abbreviated CODASYL. The objectives of CODASYL were to create a non-proprietary language that could serve as a universal communication means that would solve the problems of device dependent languages with all their different dialects. The CODASYL conference is one of the early — if not the earliest

*Alex Sellink was sponsored in part by bank ABN Amro, software house DPFinance, and the Dutch Ministry of Economical Affairs via the Senter Project #ITU95017 "SOS Resolver".

†Chris Verhoef was supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organization for Scientific Research (NWO), project *Interactive tools for program understanding*, 612-33-002.

— examples of an attempt to prevent incompatibility problems in software by means of standardization. The goal of CODASYL was to develop a programming language with the following properties:

- based on English Language (so that it could be easily read);
- geared towards administrative programming;
- maintainable;
- usable on every hardware platform.

After a year the first proposal for a programming language referred to as COBOL-60 emerged. Although this language had many of the above properties incorporated there was still a problem with device independency. After the emerge of compiler technology, these problems were solved. During the sixties several development extensions were proposed to COBOL and in 1968 the *American National Standards Institute* (ANSI) published a COBOL standard. In 1970 this standard was approved by the *International Organization for Standardization* (ISO).

Since then, the ongoing development of COBOL lead to new COBOL standards in 1974 and 1985. Recently, an object oriented version of COBOL is developed. Whether or not al these standards meet the requirements as formulated by CODASYL is a question that we try to answer in the next section.

2 The Design Objectives in Retrospect

A COBOL program consists of four divisions. We can compare them to chapters in natural language. Each division consists of sections, the sections contain paragraphs, the paragraphs in turn contain sentences, and the sentences contain statements. These concepts are prominent consequences of the aim to resemble a natural language. Since COBOL sentences — as the following code example shows — indeed show a close correspondence to natural language we can conclude that the designers of COBOL succeeded in fulfilling the first design property.

```
IF X EQUALS 3 OR 4
   ADD 1 TO X
   MULTIPLY Y BY 2
   MOVE SPACES TO S.
```

The massive usage of COBOL in the administrative sector proves that the second design property (geared towards administrative programming) is also fulfilled.

As for the maintainability of COBOL we can conclude that this objective is not accomplished. Numerous publications on this subject and the myriad of so-called legacy systems are our witness. Since COBOL is available on virtually every hardware platform the latter objective is reached as well.

The initial idea that COBOL would be a solution to dialects is not reached. Form there are many different dialects in use. Each with their own special features, extensions etc.

Let us mention some dialects:

- ANSI-74 COBOL
- ANSI-85 COBOL
- IBM OS/VS COBOL
- IBM COBOL/II
- IBM COBOL SAA
- Unix COBOL X/Open
- MicroFocus COBOL
- Microsoft COBOL
- Ryan McFarland RM/COBOL
- Ryan McFarland RM/COBOL-85
- DOSVS COBOL
- UNIVAC COBOL
- Fujistu COBOL

3 Reading versus Parsing

In the previous section we argued that CODASYL succeeded in fulfilling two of the four design requirements, thus answering the question we formulated in the end of the introduction. Another, even more interesting, question that naturally arises is whether or not the original design requirements are still considered valuable after 30 years of experience in software development. For three of the four design requirements we can give a positive answer to this question. Only the first requirement (based on English language) is not considered a desirable design requirement nowadays. Of course, a syntax that is easy to read is preferred above one that is not. However, the assumption that this property can be met by means of having a close resemblance to natural language is not considered correct anymore. The point is that there is an essential difference between conventional text (as can be found in books, papers, etc.) and source text that serves as input for a computer program. The logical structure of the latter one is far more complex. We argue that this should have consequences for the syntax one chooses: in general we can distinguish two different activities both having plain text as input. These two activities are *reading* and *parsing*. When the syntax for some language is developed one should have in mind whether text in this new language should be easy to read, easy to parse, or both. In the early days of programming the answer to this question roughly sounded as follows:

Humans read text and computers parse text. Since life should be made easy for humans — and not necessarily for computers — reading is more important than parsing.

Nowadays we think that the situation is not that simple. To explain this let us first take a look at a COBOL sentence that is slightly more complicated than the previous one, and review its readability. The following example is taken from [Volmac82].

```
IF condition-1
    statement-1-1
    IF condition-2
        IF condition-3
            statement-3-1
           ELSE statement-3-2
       ELSE statement-2-2.
```

In our opinion the above COBOL sentence is hard to read. The Volmac Groep has the same opinion. We quote from [Volmac82]:

> Because their logic is difficult to follow, nested IF statements should wherever possible be avoided in a COBOL program.

Thus, we can savely conclude that sentences with a complex logical structure are difficult to read — even if the syntax is close to natural language. A second important observation is that whenever a human is forced to read a sentence with a logical structure that is too complex to comprehend he or she will try to parse that sentence. Thus breaking the information into smaller pieces that are easier to understand. Combining these two observations we find that there might be at least one argument why computer programs (that usually have a complex logical structure) should not only be easy to read but also be easy to parse. The alternative is to try to avoid complex logical structures (like nested IF-statements) in computer programs. In [Volmac82] this alternative is advocated. We quote:

> Often a series of simple IF statements can be used in place of the nested IF statement.

We display the alternative they propose for the nested IF sentences given on page 4. The label R1014 is a fresh label of a newly created empty paragraph:

```
    IF condition-1
            statement-1-1
      ELSE GO TO R1041.
    IF condition-2
            statement-2-2
            GO TO R1041.
    IF condition-3
            statement-3-1
      ELSE statement-3-2.
R1014.
```

Obviously, the price that is paid now is the introduction of two unstructured jump instructions. Apparently, this was not considered as undesirable as it is considered today. Presently, most companies oblige their programmers to program without using unstructured jump instructions, in order to lower the costs of software maintenance. If we don't want to use the alternative for the nested `IF` sentence on page 4, as proposed by [Volmac82], we have to accept that source text for computers is logically too complex to comprehend and must be parsed. Thus, we must see to it that parsing becomes as easy as possible. Obviously, the complexity of parsing is immediately related to the complexity of the grammar in question. Thus, syntax for programming languages must have grammar rules that are as simple as possible. This can be reached by adding all kinds of markers to reveal the logical structure. Scope terminator symbols are examples of such markers. Obviously, in a language that should have a close resemblance to natural language (old COBOL dialects) such markers are not present. The dramatic consequences for the complexity of the grammar is well illustrated by the following quotation[1] from [ANSI 85, loc. cit. p. IV-40] explaining the scoping rules of statements:

> When statements are nested within other statements which allow optional conditional phrases, any optional conditional phrase encountered is considered to be the next phrase of the nearest preceding unterminated statement with which that phrase is permitted to be associated according to the general format and the syntax rules for that statement, but with which no such phrase has already been associated.

About 20 years ago, one started to realise that such complex semantical properties should be eliminated from the grammar, at the cost of a syntax that is less close to natural language. In 1985, when the new COBOL standard was published [ANSI 85], one of the important changes was that the initial design decision to have a "natural language syntax" as much as possible was departed. Amongst the 21 major changes incorporated in the COBOL specifications within the CODASYL Journal of Development 1978 was the inclusion of 19 explicit scope terminators. We quote from [ANSI 85, loc. cit. p. XVIII-7]:

> The inclusion of additional facilities to support structured programming, including implicit and explicit terminators to delimit the scope of statements and the `CONTINUE` statement.

Due to downwards compatibility issues, the old paradigm could not be completely departed. Implicitly closed `IF`-constructions are still valid syntax in COBOL 85. Thus, both the code example given on page 4 and the following code fragment are correct COBOL 85-syntax:

```
IF condition-1
```

[1]This quotation also serves as a excellent example illustrating that even pure english natural language must be parsed in order to detect its meaning, if the sentence in question has a logical complexity that is sufficiently high.

```
        statement-1-1
        IF condition-2
            IF condition-3
                statement-3-1
            ELSE
                statement-3-2
            END-IF
        ELSE
            statement-2-2
        END-IF
    END-IF.
```

The presence of explicit scope terminators `END-IF` not only improves the readability of this sentence, but it also — as these things are related — simplifies the structure of the underlying grammar. Apart from the "readability issue", simple grammars have another important advantage. This latter advantage is illuminated in the next section.

4 Code Restructuring

Although new COBOL dialects are as much as possible upwards compatible[2] with respect to older dialects, the owners of COBOL software usually want to adapt their code to new standards as soon as these new standards have become sufficiently stable. In other words: the adaptations carried out by the compatibility tools are not satisfactory in some sense. This is due to the fact that compatibility tools solve the incompatibility problems with the least possible effort. Typically, a tool for upgrading COBOL 74 to COBOL 85 does not add scope terminators or reformulate patterns where an `EVALUATE`-statement (a case construction, not present in COBOL 74) could be used. This is hardly surprising. In order to introduce these new concepts in old source code one has to trace patterns that give rise to usage of new features. Location of such patterns is far more complicated than the kind of patterns that are currently traced by the compatibility tools, and require deep knowledge of the underlying grammar. The currently existing compatibility tools do not have this deep grammar knowledge built in. Obviously, the development of tools that perform non-trivial transformations on source code is easier on languages that have a simple grammar. Thus, another important advantage of simple grammar rules is the positive effect that it has on the complexity of the supporting tool environment. The impact of this should not be underestimated. There is a strong need for tools that perform the kind of non-trivial transformations we mentioned. From reliable sources we understood that a large Dutch financial company deployed a programmer (the poor chap) for one year to add explicit scope terminators into a large COBOL application by hand. In [BSV97b, BSV97a]

[2]Moreover, tools that (almost) automatically convert your code to the new standard are supplied with the new release of the compiler. An example of a collection of such tools is CCCA facility of IBM.

it is explained how this task could have been performed in a few hours rather than in a year.

We give an example of a transformation which is directly related to the evolution of the COBOL language. Consider the following code fragment taken from [LL89, loc. cit. p. 129]:

```
IF BALANCE > ZERO
    IF BALANCE > 100
       COMPUTE FINANCE-CHARTS = BALANCE * .015
       ADD FINANCE-CHARTS TO BALANCE
    ELSE
       MOVE 1.50 TO FINANCE-CHARTS
       ADD FINANCE-CHARTS TO BALANCE
ELSE
    MOVE ZERO TO FINANCE-CHARTS.
```

Note that ADD FINANCE-CHARTS TO BALANCE occurs in both the THEN and ELSE branch. The reason for this is that a conditional statement is forced to be the last construction of a sentence due to the implicit scope termination using a separator period. Thus the part of the code that actually is performed unconditionally is duplicated and added at the end of both branches. Note that it is not a solution to put ADD FINANCE-CHARTS TO BALANCE as first statement in the next sentence since then it will be outside the scope of the outermost IF statement.

Using the COBOL 85 standard we can transform this into its natural form:

```
IF BALANCE > ZERO
    IF BALANCE > 100
       COMPUTE FINANCE-CHARTS = BALANCE * .015
    ELSE
       MOVE 1.50 TO FINANCE-CHARTS
    END-IF
    ADD FINANCE-CHARTS TO BALANCE
ELSE
    MOVE ZERO TO FINANCE-CHARTS
END-IF.
```

It is possible to automatically detect the old patterns and to replace them for the more natural pattern that avoids code duplication. See [BSV97b, BSV97a] for details.

Not all transformations on source code are related to the evolution of compilers. Also a design decision (2 digits for year variables), changing business requirements, or even political factors (the Euro) may give rise to transformation on source code. We end this article with an example of a so-called Year 2000 conversion. This example is particularly interesting to us because it is not just an example of a transformation on COBOL code, but also an illustration of "implicit scope termination grief". Due to the possibility of implicit scope termination, the incorrect replacement pattern does not lead to compiler errors and thus not to syntactically incorrect code. The example is communicated to us by Dr. Joseph Kisting. He is manager of the Competence Center Kalenderjahr 2000 of Debis Systemhaus. Debis is owned by Daimler-Benz InterServices and Cap Gemini Sogeti. We kindly acknowledge permission of Debis Systemhaus to publish the code fragment in question:

```
    IF LIEFER-JJ < 50
        MOVE 20 TO LIEFER-JH
    ELSE
        MOVE 19 TO LIEFER-JH

    MOVE BETRAG TO BUCHUNGSBETRAG
    MOVE  MENGE TO BUCHUNGSMENGE
    WRITE BUCHUNGSDATEI
        FROM BUCHUNGSSATZ.
B99.
    ...
```

The first four lines were produced by a Year 2000 solution tool. The last four lines are original code. A standard solution used by Y2K tools[3] is used here: for values for `LIEFER-JJ` smaller than `50` it puts in the variable `LIEFER-JH` the value 20 and for other years the value it assigns is 19 so that the complete calculation will be a Year 2000 compliant. This is in order, and the code *is* Year 2000 compliant. The code compiles and gives the good output before the Year 2000. However, as soon as `LIEFER-JJ` is smaller than `50` the original code is no longer evaluated since the separator period on the last line delimits the scope of the `IF`. The problem here is that the conditional statement that was produced by the Y2K tool did not include a scope terminator. Due to the separator period at the last line the original code became conditional. Of course, this is a bad Y2K tool. Still, the implicit scope terminator rules do not reveal the mistake, for instance with a compile error, since the separator period that first just ended a sentence now also serves as an implicit scope terminator. Here is a replacement pattern that would not cause a mistake in the above example:

```
IF LIEFER-JJ < 50
    MOVE 20 TO LIEFER-JH
ELSE
    MOVE 19 TO LIEFER-JH.
```

Next we will show that using a separator period in the replacement pattern will not solve the problem in all cases. To illustrate this we will show that using the separator period as a scope terminator does not give correct results in all cases. Consider an original conditional code fragment below. The task of the Y2K tool is to change the 3rd line into a conditional one in order to take care of the year 2000. Let us suppose that the original code fragment of Debis Systemhaus was:

[3]Y2K tool stands for year 2 kilo tool, i.e., year 2000 tool.

```
     IF BETRAG > 0
    MOVE BETRAG TO BUCHUNGSBETRAG
    MOVE  MENGE TO BUCHUNGSMENGE
    WRITE BUCHUNGSDATEI
        FROM BUCHUNGSSATZ.
B99.
    ...
```

We add the replacement pattern that we proposed above in the code fragment:

```
     IF BETRAG > 0
IF LIEFER-JJ < 50
MOVE 20 TO LIEFER-JH
ELSE
MOVE 19 TO LIEFER-JH.
    MOVE BETRAG TO BUCHUNGSBETRAG
    MOVE  MENGE TO BUCHUNGSMENGE
    WRITE BUCHUNGSDATEI
        FROM BUCHUNGSSATZ.
B99.
    ...
```

which places the original code outside the scope of the outermost `IF`. In fact, this implies that a solution to solve the Y2K problem using `IF` constructs relies on the use of explicit scope terminators. For, if the replacement pattern would have been

```
IF LIEFER-JJ < 50
    MOVE 20 TO LIEFER-JH
ELSE
    MOVE 19 TO LIEFER-JH
END-IF
```

the Y2K tool would have given a correct output in all cases. So we can conclude that an mportant part of Y2K solutions in COBOL is to take care of adding explicit scope terminators in order to prevent mistakes of automated conversions and in order to detect erroneous Y2K conversions. Dealing with this problem is nontrivial as pointed out earlier in this paper. But it can be done.

5 Conclusions

Concluding we can say that the initial design of COBOL that it should resemble a natural language as close as possible has lead to significant problems in Dutch COBOL software. The use of sophisticated restructuring technology is needed in order to improve the maintainability of such code. The Y2K problem puts restructuring of COBOL software in an actual context and accentuates the necessity to structure the COBOL systems.

References

[ANSI 85] American National Standards Institute, Inc. *Programming Language - COBOL*, ANSI X3.23–1985 edition, 1985.

[BSV97a] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *proceedings of the fourth working conference on reverse engineering*, pages 144–153, 1997.

[BSV97b] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purposes. In M.P.A. Sellink, editor, *Proceedings of the 2nd international workshop on the theory and practice of algebraic specifications*, 1997. To appear.

[LL89] J. Longhurst and A. Longhurst. *COBOL*. Prentice-Hall, 1989.

[Volmac82] Volmac Groep. *Volmac Standard COBOL*, second edition, 1982.

ToolBus and Tree Substitution

-- onzin door Yerhpmuh Trebor Sretlaw --

Met de Latin Klup, goedemiddag!
Met Paul Klint. Kan ik misschien Luap Tnilk spreken?
Met de Latin Klup. Kan ik u misschien helpen?
Met Paul Klint. Kunt u mij verbinden met Luap Tnilk?
Ik kan u niet verstaan.
LUAP TNILK

`tuut tuut tuut`

Met de Latin Klup, Goedemiddag!
Met Paul Klint. Kunt u mij verbinden met Luap Tnilk?
Ik kan u goed verstaan, meneer Klint. Heeft u een momentje?
"~er ~pilk toest~l 12, ~ pilk ~stel 12"

-

Meneer Klint, ik verbind u door.
Met Lunat Pilk.
Goedemiddag. U spreekt met Paul Klint. Ik was feitelijk op zoek naar Luap Tnilk.
Neen. Haha. Neen. Dan zit u fout. U moet geloof ik doorkiesnummer 317 hebben, maar ik kan u niet doorverbinden. Goedemiddag.

`tuut tuut tuut`

Hallo, met LuPi-Klant, met Knut
Uh. Goeiedag. Is dit het toestel van dhr. Tnilk?
Wie?
Tnilk. Dhr. Luap Tnilk.
Nee. Ken ik niet. Goeiemiddag!

`tuut tuut tuut`

Met de Latin Klup, goedemiddag!
Met Paul Klint. Ik ben op zoek naar Dhr. Luap Tnilk , maar ik werd zojuist met Dhr. Pilk verbonden.
Met Lunat, Ja.

-

Maar ik wilde graag Dhr. Tnilk spreken.
Maar het is dinsdag.
Hoe bedoeld u?
Nou, het is dinsdag en dan is Dhr. Tnilk natuurlijk niet hier bereikbaar.
Weet u ook waar hij wel bereikbaar is?
U bedoelt op dinsdag, want normaal is hij hier wel bereikbaar.
Jawel, maar ik bedoel op dinsdag.
Neen.
Heeft u echt geen ander ...

Wacht! Tnilk. Daar stond iets... Uhm. Heeft u het al bij zijn 2e kantoor geprobeerd?

Neen. Ik geloof niet ...

Wacht!. Hier heb ik het. Hihi. U moet zijn bij, hihi, Pliant Luk. En dan moet u vragen naar Dhr. Tnilk.

Ah. Dank u wel

`tuut tuut tuut`

Pliant Luk Enterprises. How can I help you?

This is Paul Klint speaking. I would like to speak to Mister Tnilk please.

Sorry. I didn't catch that name.

Tnilk. Mister Luap Tnilk.

Sorry. You're sure he's with us?

Well. I was just told that...

Do you know his extension?

No. Well... 317?

No. I dont't think so. That would be Nat Pilluk, and I don't think he has a guest today. Could you hold on for a moment?

-

Pilluk.

Hello, my name is Klint. I was hoping to speak to mister Tnilk.

Wat zegt u?

Oh. Ah. Mijn naam is Klint. Ik probeer dhr. Tnilk te bereiken.

Ja. Blijft dat zo. Ik ben hier de receptioniste niet. Ik verbind u wel door.

`tuut tuut tuut`

Pliant Luk Enterprises. How can I help you?

This is Paul Klint speaking. I am trying to locate mr. Luap Tnilk. You have just connected me to your mr. Pilluk, but apparently mr. Tnilk isn't there.

Let me see. Tnilk, you said. What an odd name... No, He's not on my lists. Could you tell me what this is about.

Well. Uhm. I am a professor of Computer Science, and one of my projects is called 'Toolbus'

I see.

A few weeks ago I gave a demonstration of the Toolbus, and in my audience was a gentleman who introduced himself as Luap Tnilk.

Yes.

This gentleman told me he was in fact also a professor of Computer Science, on the subject of term rewriting, or tree substitution, as he called it.

I see.

He told me that his pet project, a tree substitution engine, was somehow complementary to my Toolbus.

Yes.

He asked me to call him at the Latin Klup, and they referred me to you.

Ah, the Latin Klup. You mean our Tibettan Professor Tnilk . Could you hold on for a moment.

-

Tem forp Luap Tnilk.
Good afternoon mr. Tnilk. My name is Paul Klint. We met the other day when ...
Soo sorrry not so spek 'nglish. Ples to spek dutch understoond.
U verstaat Nederlands, meneer Tnilk. Wat prettig, want ikzelf versta wat tibettaans.
Jilb tad et neroh.
Wij hebben elkaar laatst ontmoet toen ik u de Toolbus presenteerde...
Ha! Paul Klint fo ToolBus
Inderdaad. U vertelde mij toen in het kort over uw revolutionaire tree-replacement engine, of, zoals u het noemde, uw loot substitutie model.
Aj, tool eitutitsbus.
U suggereerde toen dat onze systemen complementair waren, maar achteraf kon ik uw overwegingen niet goed meer recapituleren.
Ratnemelpmoc?
Inderdaad.
Ha... Srevni.
Ja.
-
Of bedoelt u soms...
-
U bedoelt toch niet...
Lewaj.
De naam van uw loot substitutie machine is dus.
SubLoot
En u vermoedt niet dat beide tools functioneel complementair zijn.
Tein thce. Eiw nak te neggez?
Goed. Nu ja. In ieder geval prettig weer met u gesproken te hebben.
Gad Paul Klint nav ed ToolBus
Tot ziens, Luap Tnilk van SubLoot.

$\bar{\vee}$

WAT DEED PAUL DE DAG NADAT KENNEDY VERMOORD WAS?

Als je ergens lang werkt, gaan je collega's bijna voelen als familie. Maar Paul voelde altijd nog net iets vertrouwder dan de andere MC-ers, want wij hadden op de middelbare school vier jaar lang bij elkaar in de klas gezeten. Het was een berucht lastige klas, waarvan wij de twee stilste en onopvallendste leerlingen waren. Toen ik in 1972 in de Ruyschstraat kwam wonen, zag ik Paul wel eens aan het eind van de dag op de tramhalte van lijn 3 bij de Camperstraat staan. Ik kon me niet voorstellen dat hij bij de Amstelbrouwerij zou werken. Ik had dan ook een onvervalste Aha Erlebnis toen hij op mijn allereerste werkdag bij het MC de bibliotheek binnenstapte.

Als een CWI bandje weer eens een drummer zoekt, stromen de berichten binnen dat Paul kan drummen. Maar ik geloof dat ik de enige CWI-er ben die hem ooit aan de slag heeft gezien, en wel in 1964 in onze eigen versie van Zo is het toevallig ook nog eens een keer, geregisseerd door Loetje, het knapste jongetje van de klas. Loetje kwam een paar jaar geleden een boek lenen bij het CWI. Ik maakte een praatje met hem en het verbaasde hem geenszins dat ik in een bibliotheek beland was, want dat was precies hoe hij mij altijd ingeschat had. Wel vroeg hij nieuwsgierig of die hoogleraar in de informatica Paul was. "Dat had ik toch niet gedacht destijds," zei hij. "Hoezo niet?" vroeg ik. "Volgens mij doet hij precies wat hij leuk vindt, die heeft echt zijn draai gevonden." "Ik bedoel," mompelde Loetje. Hij zei het niet ronduit, maar het kwam erop neer dat hij dertig jaar geleden niet erg onder de indruk was van Paul's rapportcijfers. Dat kan wel kloppen. Ik herinner me dat we Sallustius lazen. Vraag me niet wie Sallustius was of waar hij over schreef. Ik herinner me alleen zijn naam en dat ik hem met de grootst mogelijke tegenzin vertaalde. Voor zover je dat vertalen kunt noemen, want een medeleerling had uitgerekend dat de rij waarin zowel Paul als ik zat gemiddeld een 3 had voor Latijn.

Paul en ik hebben elkaar eigenlijk pas leren kennen toen we ons van het Vossius bevrijd hadden: ik van het standsverschil waarvan ik me altijd bewust was en hij van meneer Tielrooy door wie hij nog jaren in zijn dromen achterna is gezeten. Meneer Tielrooy gaf Nederlands, maar er was niets fijnzinnigs aan hem. Hij was precies een adelaar, niet alleen in zijn uiterlijk maar ook in de manier waarop hij op zijn prooi dook. Het was een forse man met een imposante ronde schedel onder wat schaarse blonde krullen. Een fikse haviksneus domineerde het gezicht en vanonder de zware wenkbrauwen hielden twee rollende blauwe ogen de klas in de gaten. Het leek mij een vrij simpele, niet onvriendelijke man. Alleen werd hij af en toe overvallen door angstwekkende driftbuien. En Paul zat nou eenmaal in die berucht lastige klas. Dan liep dat zware ronde hoofd rood aan, de neus trok nog dieper over de toch al verbeten mond, de ogen loerden van links naar rechts en meneer Tielrooy rende als een getergde stier in het gangpad tussen de banken heen en weer en sloeg dreunend met zijn vlakke hand op het tafelblad. De lastige klas was doodstil en durfde zich zelfs niet te verroeren, want elke beweging kon fataal zijn en dan zou meneer Tielrooy, ondanks zijn blanke pit, een moord begaan.

Voor mij is de goede man onlosmakelijk verbonden met een echte moord. Wat deed je toen Kennedy ...? Het was een vrijdagavond: Paul Vitányi zat, toen al, te drinken in een studentencafé, John Tucker mocht langer opblijven en speelde met zijn Lego, en ik moet bij die vraag altijd aan meneer Tielrooy denken, want ik hing boven mijn Nederlandse boek. We hadden nog school op zaterdag en ik bereidde een Nederlands proefwerk voor. Toen de leraar de volgende ochtend aanstalten maakte om de blaadjes rond te delen, barstte de klas verontwaardigd los: "Ah meneer! Nee meneer! Kennedy vermoord, meneer, helemaal niks meer kunnen doen!" "Onzin!" brieste meneer Tielrooy en deelde de proefwerkblaadjes uit.

Zo lagen de gezagsverhoudingen toen nog. Een half jaar later begaf Paul zich naar een veilinghal in Blokker voor het concert van de Beatles. Hij mocht dan een stille, onopvallende leerling zijn, hij was erbij toen de tijden, ook in Nederland, gingen veranderen.

Joke Sterringa

From Context-free Grammars with Priorities to Character Class Grammars

Eelco Visser

Priority and associativity declarations are used to disambiguate ambiguous fragments of context-free grammars. Usually this concerns expression grammars. It is possible to describe the same language by means of an unambiguous context-free grammar only, but using auxiliary non-terminals and extra chain productions resulting in a grammar that generates different and larger trees and extra parse steps.

In this paper we introduce a grammar transformation that translates a context-free grammar with priorities to a character class grammar that does only generate trees without priority conflicts. The transformed grammar has the property that each production corresponds to a production in the original grammar and that no extra productions are used. The parse trees over the transformed grammar are therefore isomorphic to parse trees over the original grammar.

1 Introduction

1.1 Priority and Associativity

Priority and associativity are notions that exist ever since formal languages have been used in mathematics. To read the mathematical expression $x \oplus y \ominus z$ it is necessary to know the priority relation between the operators $\oplus$ and $\ominus$. For instance, if $\oplus$ has higher priority than $\ominus$ the expression is read as $(x \oplus y) \ominus z$. In order to use mathematical expressions in a programming language the notions of priority and associativity have to be formalized in the description of the syntax of the language. Several formalizations have been developed.

Floyd (1962) introduced *operator precedence grammars* that allow the declaration of precedence and associativity among the operators of a context-free grammar over a single non-terminal in which no two non-terminals are adjacent. A restricted version of operator precedence can still be encountered in most reference manuals of programming languages that give a list of operators ordered by precedence. However, operator precedence is a very restricted method. It is not defined for grammars with ϵ-productions and cannot cope with 'invisible' operators such as the application operator in functional languages.

Another way to express priority and associativity is to encode the information in the context-free grammar of the language. This produces a considerable overhead (1) in the number of non-terminals, since each priority level is encoded by means of a non-terminal and (2) in the number of productions, since extra chain productions are needed. The effect is more parse steps (reductions with chain rules) and parse trees that are different from the desired abstract syntax. Moreover, this encoding 'hardwires' priority information in the productions making the grammar harder to understand and to extend.

To overcome these drawbacks a number of more declarative methods have been proposed to separate the priority rules from the grammar rules and directly use the notions of priority and associativity. An ambiguous grammar of expressions is combined with a declaration of priority and associativity between the productions of the grammar. This results in better abstract syntax because no auxiliary non-terminals and chain rules have to be used. Earley (1975) and Aho *et al.* (1975) independently introduced priority declarations as a number of binary relations (priority, and left-, right-, and non-associativity) on productions of context-free grammars. These relations were restricted to range over unary and binary operators.

Aho *et al.* (1975) interpret such declarations by solving conflicts in an LR(1) parse table. Certain patterns of shift/reduce conflicts produced by ambiguous binary expressions can be solved by considering priority declarations. For instance, a shift/reduce conflict between the items $E \bullet\ + E \to E$ and $E + E \bullet\ \to E$ is solved in favor of reduce if $+$ is declared as left-associative. This implementation method is used in the YACC parser generator of Johnson (1975).

1.2 Disambiguation Filters

Heering *et al.* (1989) introduce a more general definition of priorities in the syntax definition formalism SDF. Priority declarations can range over arbitrary productions in the grammar. The YACC approach cannot be used for this more general definition of priorities, because it cannot deal with mixfix operators or binary expressions with an implicit operator (e.g., application in functional languages). Furthermore, SDF parsers are required to cope with arbitrary context-free grammars because this eases grammar development. These problems are solved by the following approach to parsing: A string is parsed using the context-free grammar part of a syntax definition using Generalized-LR parsing (Tomita, 1985; Rekers, 1992). The result is a parse forest representing all possible parses for the string. This forest is pruned using a filter that interprets the priority declarations. In fact two consecutive filters are used, one that selects trees without priority conflicts and a second one that selects the smallest tree in a multi-set ordering on parse trees based on priorities. In this paper we will concentrate on the first phase. The method is formalized by Klint (1988) who also gives a large number of examples.

Aasa (1992) and Thorup (1994) explore variants of priorities interpreted as filters. Aasa (1992) gives an alternative interpretation of priorities that is more complete than the definition of SDF. It is defined as a filter on sets of parse trees and in a variation of Earley's parsing algorithm. Thorup (1994) introduces disambiguation by means of tree rewrite rules. This can be used for instance to translate a right-associative tree to a left-associative one, thereby declaring that the construct should be interpreted as left-associative. A special case is a rewrite rule that rewrites a tree to $*$, meaning any other tree. The effect is that the tree in the rule is excluded as a subtree. This method is interpreted by an algorithm that tries to solve conflicts in an LR(k) parse table based on the rewrite rules. An important consideration is that the method should be complete, i.e., the resulting parser should be able to parse all strings generated by the original grammar.

Klint and Visser (1994) generalize the notion of a filter on sets of parse trees providing a framework of *disambiguation filters* for formalization and comparison of disambiguation methods independent of parsing algorithms. Implementation of disambiguation is achieved by composing a Generalized-LR parser for the context-free part of the grammar with a filter expressing the disambiguation method. The filter prunes the parse forest produced by the GLR parser, deleting all trees that are not selected by the disambiguation method. This framework gives a very general account of disambiguation of context-free grammars. In Klint and Visser (1994) and Visser (1997d) several applications are discussed.

As an implementation paradigm disambiguation filters are not adequate for all disambiguation methods. If the number of possible trees generated by the grammar is very large, filtering after parsing leads to a bad performance of the resulting parsers. By partially evaluating the composition of a GLR parser and a filter this performance might be improved. In Visser (1997a) a case study in such a partial evaluation is discussed: a parser generation time interpretation of priorities is derived from the composition of the SLR(1) algorithm and a filter based on priority conflicts. Parsers produced by this partial evaluation do not construct parse trees with priority conflicts, thus saving computation time and memory. The method is used in the implementation of a parser generator for SDF2 (Visser, 1997c, 1997d), which is a redesign of SDF.

1.3 Grammar Transformation

In this paper we introduce a grammar transformation to interpret priorities. The advantage of this method is that it simplifies the parser generator by shifting the computation with priority rules from the parser generator to a grammar transformer. The main idea is to use character classes, compact representations of sets of numbers, to represent the priority levels in the grammar. By using such sets, no chain productions are needed and the exact structure of the original grammar can be preserved.

The paper is structured as follows. In §2 context-free grammars with priorities are introduced. A grammar generates a family of sets of parse trees. Priorities give a restriction on this set of parse trees by excluding trees matching certain patterns. In §3 character class grammars are introduced. Grammar symbols are restricted to sets of characters represented by numbers. By means of such character classes a non-terminal can be represented by the set of production numbers that define the non-terminal. It is shown how context-free grammars can be expressed in terms of character class grammars and vice versa

In §4 priorities are expressed as an operation on a character class grammar. The result is a character class grammar that represents the original context-free grammar with priorities.

2 Context-free Grammars with Priorities

In this section we introduce context-free grammars, the interpretation of context-free grammars as tree generators, and the framework of priority and associativity declarations for the disambiguation of a certain class of ambiguous context-free grammars.

2.1 Context-Free Grammars

Context-free grammars were introduced by Chomsky (1956) to describe the *phrase structure* of natural language sentences. In most later work this aspect of context-free grammars is often ignored and the emphasis is on the string rewriting interpretation of context-free productions. Here we will use the phrase structure view to define context-free grammars because we are not just interested in the language defined by a grammar, but particularly in the structure assigned to sentences, since that will be the input for compilers and interpreters.

A commonly used notation for context-free grammars is BNF introduced by Naur *et al.* (1960) for the definition of Algol60. We use the notation of the syntax definition formalism SDF (Heering *et al.*, 1989). The most significant difference is the $\alpha \to A$ notation for productions, instead of the traditional $A \to \alpha$ or $A ::= \alpha$, to emphasize the use of productions as mixfix function declarations.

Formally, we have the following definition of context-free grammars.

Definition 2.1 (CFG) A *context-free grammar* $\mathcal{G}$ is a triple $\langle V_N, V_T, \mathcal{P}\rangle$, with V_N a finite set of nonterminal symbols, V_T a finite set of terminal symbols, V the set of symbols of $\mathcal{G}$ is $V_N \cup V_T$, and $\mathrm{P}(\mathcal{G}) = \mathcal{P} \subseteq V^* \times V_N$ a finite set of productions. We write $\alpha \to A$ for a production $p = \langle \alpha, A\rangle \in \mathcal{P}$.

Observe that we do not distinguish a start symbol from which sentences are derived. Each nonterminal in V_N generates a set of phrase structures or parse trees as is defined in the following definition.

Definition 2.2 (Parse Trees) A context-free grammar $\mathcal{G}$ generates a family of sets of *parse trees* $\mathcal{T}(\mathcal{G}) = (\mathcal{T}(\mathcal{G})(X) \mid X \in V)$ that contains the minimal sets $\mathcal{T}(\mathcal{G})(X)$ such that

$$\frac{X \in V_T}{X \in \mathcal{T}(\mathcal{G})(X)}$$

$$\frac{A_1 \ldots A_n \to A \in \mathrm{P}(\mathcal{G}),\ t_1 \in \mathcal{T}(\mathcal{G})(A_1),\ \ldots\ ,\ t_n \in \mathcal{T}(\mathcal{G})(A_n)}{[t_1 \ldots t_n \to A] \in \mathcal{T}(\mathcal{G})(A)}$$

We will write t_α for a list $t_1 \dots t_n$ of trees where α is the list of symbols $X_1 \dots X_n$ and $t_i \in \mathcal{T}(\mathcal{G})(X_i)$ for $1 \leq i \leq n$. Correspondingly we will denote the set of all lists of trees of type α as $\mathcal{T}(\mathcal{G})(\alpha)$. Using this notation $[t_1 \dots t_n \to A]$ can be written as $[t_\alpha \to A]$ and the concatenation of two lists of trees t_α and t_β is written as $t_\alpha t_\beta$ and yields a list of trees of type $\alpha\beta$.

The *yield* of a tree is the concatenation of its leaves. The language $L(\mathcal{G})$ defined by a grammar $\mathcal{G}$ is the family of sets of strings $L(\mathcal{G})(A) = \text{yield}(\mathcal{T}(\mathcal{G})(A))$.

Definition 2.3 (Parsing) A *parser* is a function Π that maps each string $w \in V_T^*$ to a set of parse trees. A parser Π *accepts* a string w if $|\Pi(w)| > 0$. A parser Π is *deterministic* if $|\Pi(w)| \leq 1$ for all strings w. A parser for a context-free grammar $\mathcal{G}$ that accepts exactly the sentences in $L(\mathcal{G})$ is defined by

$$\Pi(\mathcal{G})(w) = \{t \in \mathcal{T}(\mathcal{G})(A) \mid A \in V_N, \text{yield}(t) = w\}$$

Example 2.4 As an example consider the following ambiguous grammar of expressions:

```
[\+]        -> "+"
[\*]        -> "*"
[a-z]       -> E
E "+" E     -> E
E "*" E     -> E
```

In this examples character classes are used to denote a number of alternative terminal characters. According to this grammar the string $a + a * a$ has two parses:

$$\begin{aligned}\Pi(\mathcal{G})(a + b * c) = \{&[[[a \to E] + [b \to E] \to E] * [c \to E] \to E] \\ &[[a \to E] + [[b \to E] * [c \to E] \to E] \to E]\}\end{aligned}$$

2.2 Priorities

Earley (1975) defines a priority declaration by means of the binary relations L, R and N that declare left-, right- and non-associativity, respectively, between productions and the relation > that declares priority between productions. In SDF (Heering *et al.*, 1989) a formalism with the same underlying structure but with a less Spartan and more concise syntax is used. In SDF one writes `left` for L, `right` for R and `non-assoc` for N. We will use both notations, the formal definition is the following:

Definition 2.5 (Priority Declaration) A *priority declaration* $\text{Pr}(\mathcal{G})$ for a context-free grammar $\mathcal{G}$ is a tuple $\langle \text{L}, \text{R}, \text{N}, > \rangle$, where $\oplus \subseteq \mathcal{P} \times \mathcal{P}$ for $\oplus \in \{\text{L}, \text{R}, \text{N}, >\}$, such that L, R and N are symmetric and > is irreflexive and transitive. □

A priority declaration is interpreted as the declaration of a set of parse tree patterns that cannot be used as subtrees of parse trees.

Definition 2.6 (Priority Conflict) The set conflicts($\mathcal{G}$) generated by the priority declaration of a grammar $\mathcal{G}$ is the smallest set of parse tree patterns of the form $[\alpha[\beta \to B]\gamma \to A]$ defined by the following rules.

$$\frac{\alpha B\gamma \to A > \beta \to B \in \mathrm{Pr}(\mathcal{G})}{[\alpha[\beta \to B]\gamma \to A] \in \mathrm{conflicts}(\mathcal{G})}$$

$$\frac{\gamma \neq \epsilon,\ \beta \to B\ (\texttt{right} \cup \texttt{non-assoc})\ B\gamma \to A \in \mathrm{Pr}(\mathcal{G})}{[[\beta \to B]\gamma \to A] \in \mathrm{conflicts}(\mathcal{G})}$$

$$\frac{\alpha \neq \epsilon,\ \beta \to B\ (\texttt{left} \cup \texttt{non-assoc})\ \alpha B \to A \in \mathrm{Pr}(\mathcal{G})}{[\alpha[\beta \to B] \to A] \in \mathrm{conflicts}(\mathcal{G})}$$

This set defines the patterns of trees with a priority conflict. □

Using the definition of priority conflict we can define a filter on sets of parse trees that selects parse trees without a conflict.

Definition 2.7 (Priority Conflict Filter) A tree t has a *root priority conflict* if its root matches one of the tree patterns in conflicts($\mathcal{G}$). A tree t has a *priority conflict*, if t has a subtree s that has a root priority conflict. The filter $\mathcal{F}^{\mathrm{Pr}}$ is now defined by $\mathcal{F}^{\mathrm{Pr}}(\Phi) = \{t \in \Phi \mid t \text{ has no priority conflict}\}$. The pair $\langle \mathcal{G}, \mathrm{Pr} \rangle$ defines the disambiguated grammar $\mathcal{G}/\mathcal{F}^{\mathrm{Pr}}$. □

Example 2.8 Consider the following grammar with priority declaration

```
syntax
  [a-z]    -> E
  E "*" E -> E {left}
  E "+" E -> E {left}
priorities
  E "*" E -> E >
  E "+" E -> E
```

Here the attribute `left` of a production p abbreviates the declaration $p\,\mathrm{L}\,p$. The tree

$$[[[a \to E] + [a \to E] \to E] * [a \to E] \to E]$$

has a priority conflict over this grammar—it violates the first priority condition since multiplication has higher priority than addition. The tree

$$[[a \to E] + [[a \to E] * [a \to E] \to E] \to E]$$

does not have a conflict. These trees correspond to the (disambiguated) strings $(a + a) * a$ and $a + (a * a)$, respectively. The implication operator in logic is an example of a right associative operator: $a \to a \to a$ should be read as $a \to (a \to a)$. Non-associativity can be used to exclude unbracketed nested use of the equality operator in expressions using the production `E "=" E -> E`. □

3 Character Class Grammars

In the next section we will define a grammar transformation that embeds the definition of priorities in the productions of the grammar. For this purpose we introduce the notion of character class grammars in this section. In character class grammars the grammar symbols are character classes. Character classes are compact representations of sets of characters, i.e., numbers. Character classes originate from the definition of lexical syntax (for example, in LEX (Lesk and Schmidt, 1986)), where they are used to summarize a large number of chain productions. For instance, `[a-z]` denotes the set of all lowercase letters.

Character class grammars are used to reduce the complexity in the number of productions and non-terminals. Character classes denote an arbitrary collection of productions. Any character class grammar can be translated to a context-free grammar as we will show below, but this can lead to an exponential amount of non-terminals and for each non-terminal a number of chain productions. We also explore the reverse translation from context-free grammars to character class grammars. This translation will be the basis for the interpretation of priorities.

3.1 Character Classes

A character class is a compact representation of a set of characters.

Definition 3.1 (Character Class) A character class is a list $[cr_1 \ldots cr_n]$ of numbers and pairs of numbers that represents the set of those numbers that are either contained in the list or that are contained in one of the intervals. Operations on character classes are union ($\vee$), intersection ($\wedge$), difference ($/$) and complement with respect to the interval `\0` $-$ `\TOP`.

Proposition 3.2 (Normal Form) *For each finite set of characters there is a unique, most compact character class representing it.*

Character classes can be specified using symbolic characters. These are translated to numbers using some character encoding. In this paper we use the ASCII encoding for characters in the ASCII range. An example character class is `[a-zA-Z]` denoting the set of all lower and uppercase characters. It's normal form is `[\65-\90\97-\122]`. Usually the characters in a character class are restricted to a fixed range, for instance, the 127 ASCII characters, the 256 characters that can be represented by a byte or the 16 bit characters of UniCode. However, there is no fundamental reason for this restriction. We can just as easily work with character classes in which there is no upperbound for characters or character ranges.

Visser (1997b) gives a specification of character classes in ASF+SDF including the normalization to the most compact normal form using rewrite rules. In the rest of this paper we will consider character classes modulo equivalence, i.e., assume that character classes are in normal form.

3.2 Character Class Grammars

Character classes are usually used in grammars to abbreviate a set of terminal characters. For example, the productions

```
[a-z]    -> Id
Id [a-z] -> Id
```

define the syntax of identifiers as a sequence of one or more lowercase letters. We generalize the use of character classes in grammars by allowing characters as non-terminals. In fact in character class grammars only character classes are used as grammar symbols.

Definition 3.3 (Character Class Grammar) A *character class grammar* is a finite set of productions $\alpha \to A$ such that $\alpha \in CC^*$ and $A = [c]$ is a singleton character class.

A character is a non-terminal in a CCG $\mathcal{G}$ if it is defined by at least one of the productions. Otherwise it is a terminal character.

Definition 3.4 (Parse Trees for CCGs) Given a character class grammar $\mathcal{G}$, the family of sets of parse trees $\mathcal{T}(\mathcal{G}) = (\mathcal{T}(\mathcal{G})(cc) \mid cc \in CC)$ contains the minimal sets $\mathcal{T}(\mathcal{G})(cc)$ such that

$$\frac{c \in cc,\ c \text{ is a terminal in } \mathcal{G}}{c \in \mathcal{T}(\mathcal{G})(cc)} \tag{Ch}$$

$$\frac{cc_1 \ldots cc_n \to cc_0 \in \mathrm{P}(\mathcal{G}),\ t_1 \in \mathcal{T}(\mathcal{G})(cc_1),\ \ldots\ ,\ t_n \in \mathcal{T}(\mathcal{G})(cc_n)}{[t_1 \ldots t_n \to cc] \in \mathcal{T}(\mathcal{G})(cc)} \tag{App}$$

$$\frac{t \in \mathcal{T}(\mathcal{G})(cc),\ cc \subseteq cc'}{t \in \mathcal{T}(\mathcal{G})(cc')} \tag{Sub}$$

3.3 From CCG to CFG

Any character class grammar can be expressed by means of a context-free grammar. This translation shows us why character class grammars are useful.

Algorithm 3.5 (CCG to CFG) Given a character-class grammar $\mathcal{G}$ construct the context-free grammar cfg($\mathcal{G}$)according to the following algorithm:

(0) Define the terminal alphabet V_T of cfg($\mathcal{G}$) as the terminal characters of $\mathcal{G}$.

(1) Assign to each character class cc used in some of the productions a non-terminal nt(cc).

(2) For each production $cc_1 \ldots cc_n \to cc_0$ in the grammar $\mathcal{G}$ define the production nt(cc_1)... nt(cc_n) $\to$ nt(cc_0).

(3) For each non-terminal nt(cc) define the chain productions nt($[n]$) $\to$ nt(cc) for each $n \in cc$. □

Theorem 3.6 *The trees genererated by a character class grammar $\mathcal{G}$ and its corresponding context-free grammar* cfg($\mathcal{G}$) *are isomorphic, that is,* $\mathcal{T}(\mathcal{G}) \cong \mathcal{T}(\mathrm{cfg}(\mathcal{G}))$.

Proof. Extend the translation cfg to trees, i.e., define the function cfg : $\mathcal{T}(\mathcal{G}) \to \mathcal{T}(\mathrm{cfg}(\mathcal{G}))$ as follows:

$$\frac{c \in V_T}{\mathrm{cfg}(c) = c}$$

$$\frac{cc_1 \ldots cc_n \to cc_0 \in \mathrm{P}(\mathcal{G})}{\mathrm{cfg}([t_1 \ldots t_n \to cc_0]) = [[\mathrm{cfg}(t_1) \to \mathrm{nt}([cc_1])] \ldots [\mathrm{cfg}(t_n) \to \mathrm{nt}([cc_n])] \to \mathrm{nt}(cc_0)]}$$

This translation is clearly a bijection. □

In this translation we see that character classes are more concise than context-free grammars because of the chain rules implied by the character classes.

3.4 From CFG to CCG

Conversely, and more interestingly for our intended application, character class grammars can be used to describe context-free grammars. The following algorithm constructs a character class grammar for a given context-free grammar such that it generates trees with the same structure. This time the structure is more faithfully copied.

Algorithm 3.7 (CFG to CCG) Given a context-free grammar $\mathcal{G}$, construct the character-class grammar ccg($\mathcal{G}$) according to the following algorithm:

(0) Assign a character num(X) to each terminal symbol X in $\mathcal{G}$.

(1) Assign a unique number num($\alpha \to A$) to each production $\alpha \to A$ in the grammar such that the smallest production number is larger than the largest character number.

(2) Assign to each non-terminal A in the grammar a character class nums(A) containing the numbers of the productions for that non-terminal, i.e., such that if $\alpha \to A$ is a production then num($\beta \to A$) $\in$ nums(A).

(3) For each production $\alpha \to A$: (3a) Replace the result A by the character class [num($\alpha \to A$)] containing the number of the production. (3b) Replace each non-terminal A_i in α by the character class nums(A_i). □

Theorem 3.8 *The trees genererated by a context-free grammar $\mathcal{G}$ and its corresponding character class grammar* ccg($\mathcal{G}$) *are isomorphic, that is,* $\mathcal{T}(\mathcal{G}) \cong \mathcal{T}(\mathrm{ccg}(\mathcal{G}))$.

Proof. Extend the translation ccg to trees, i.e., define the function ccg : $\mathcal{T}(\mathcal{G}) \rightarrow \mathcal{T}(\text{ccg}(\mathcal{G}))$ as follows:

$$\frac{X \in V_T}{\text{ccg}(X) = \text{num}(X)}$$

$$\frac{A_1 \ldots A_n \rightarrow A_0 \in \text{P}(\mathcal{G})}{\text{ccg}([t_1 \ldots t_n \rightarrow A_0]) = [\text{ccg}(t_1) \ldots \text{ccg}(t_n) \rightarrow \text{num}(A_0)]}$$

It is clear that $\text{ccg}(t) \in \mathcal{T}(\text{ccg}(\mathcal{G}))$ and that this translation is a bijection. □

Example 3.9 Take the following grammar of expressions

```
[\+]      -> "+"
[\*]      -> "*"
[a-z]     -> E
E "*" E   -> E
E "+" E   -> E
```

Applying the algorithm to this grammar leads to the following steps. (0) The terminals of this grammar are already character classes. Note that the normal form of `[\+]` is `[\43]`. (1) Assign numbers to the productions

```
num([\43]         -> "+")  = \258
num([\42]         -> "*")  = \259
num([\97-\122]  -> E)    = \260
num(E "*" E       -> E)    = \261
num(E "+" E       -> E)    = \262
```

(2) Assign character classes to the non-terminals.

```
nums(E)    = [\260-\262]
nums("*")  = [\259]
nums("+")  = [\258]
```

(3) Replace the non-terminals.

```
[\43]                                   -> [\258]
[\42]                                   -> [\259]
[\97-\122]                              -> [\260]
[\260-\262] [\259] [\260-\262] -> [\261]
[\260-\262] [\258] [\260-\262] -> [\262]
```

Observe that this is not the most compact encoding possible for this grammar. A more compact encoding of the example grammar is:

```
[\97-\122]                               -> [\260]
[\260-\261] [\42-\43] [\260-\261] -> [\261]
```

But the point of the encoding is not the compact representation for pure context-free grammars. In the next section we will use the encoding above as basis for the grammar transformation to embed priorities.

4 Priorities as Grammar Transformation

Given a context-free grammar with priorities we can derive a character class grammar (without priorities) that exactly describes the parse trees of the original grammar. The idea is to remove from a character class at some position in a left-hand side the numbers of productions that would cause a priority conflict at that position. For instance, to express that an addition should not occur as the child of a multiplication the production number for the addition can be removed from the class `[\260-\262]` resulting in the adapted production

```
[\260-\261] [\259] [\260-\261] -> [\261]
```

for multiplication. The functions $\mathbf{L}$, $\mathbf{M}$ and $\mathbf{R}$ in the algorith refer to left positions, middle positions and right positions in the left-hand side of a production.

Algorithm 4.1 (Priority CFG to CCG) Given a context-free grammar $\mathcal{G}$ with priorities $\mathrm{Pr}(\mathcal{G})$, construct the character-class grammar $\mathrm{pccg}(\mathcal{G})$ according to the following algorithm:

(1) Construct the CCG for the context-free part of the grammar according to Algorithm 3.7.

(2) Translate the priorities to three functions $\mathbf{L}$, $\mathbf{M}$ and $\mathbf{R}$ mapping productions numbers to sets of numbers as follows:

$$\frac{p_1\ (> \cup \texttt{right} \cup \texttt{non-assoc})\ p_2}{\mathrm{num}(p_2) \in \mathbf{L}(\mathrm{num}(p_1))} \quad \text{(L)}$$

$$\frac{p_1 > p_2}{\mathrm{num}(p_2) \in \mathbf{M}(\mathrm{num}(p_1))} \quad \text{(M)}$$

$$\frac{p_1\ (> \cup \texttt{left} \cup \texttt{non-assoc})\ p_2}{\mathrm{num}(p_2) \in \mathbf{R}(\mathrm{num}(p_1))} \quad \text{(R)}$$

(3) Filter the character classes $cc_1 \ldots cc_n$ in the left-hand side of each production $cc_1 \ldots cc_n \to [p]$ as follows:

   (a) If $n = 0$ (ϵ-production) or $n = 1$ (empty production), then do nothing.

   (b) If $n > 1$: take $cc'_1 := cc_1 / \mathbf{L}(p)$, take for $1 < i < n$: $cc'_i := cc_i / \mathbf{M}(p)$, and take $cc'_n := cc_n / \mathbf{R}(p)$. Replace the production by $cc'_1 \ldots cc'_n \to [p]$. □

Theorem 4.2 *The trees genererated by a character class grammar* $\mathrm{pccg}(\mathcal{G})$ *do not contain priority conflicts.*

Proof. For each of the clauses of Definition 2.6 we have to check that the excluded parse tree patterns are not generated by $\mathrm{pccg}(\mathcal{G})$.

(1) $[\alpha[\beta \to B]\gamma \to A] \in \text{conflicts}(\mathcal{G})$ due to the priority $\alpha B\gamma \to A > \beta \to B$. Consider the translation of the production: First the non-terminals are replaced by character classes

$$\text{ccg}(\alpha B\gamma \to A) \mapsto \text{nums}(\alpha)\,\text{nums}(B)\,\text{nums}(\gamma) \to \text{num}(\alpha B\gamma \to A)$$

Then these character classes are filtered. In particular, because of the priority rule, $\text{num}(\beta \to B) \notin cc'_B$. Hence, $\text{ccg}([t_\alpha[t_\beta \to B]t_\gamma \to A]) \notin \mathcal{T}(\text{pccg}(\mathcal{G}))(A)$ for any t_α, t_β and t_γ.

(2,3) The other cases are similar. □

Example 4.3 (1) Take the character class grammar obtained in Example 3.9. (2) We derive the following encoding of the priority rules:

```
L(\261) = [\262]             L(\262) = []
M(\261) = []                 M(\262) = []
R(\261) = [\261-\262]        R(\262) = [\262]
```

(3) Filter the non-terminal classes using this encoding

```
[\43]                             -> [\258]
[\42]                             -> [\259]
[\97-\122]                        -> [\260]
[\260-\261] [\259] [\260]         -> [\261]
[\260-\262] [\258] [\260-\261] -> [\262]
```

Observe that the transformed grammar has the exact same structure as the original context-free grammar, i.e., each production in the transformed grammar corresponds to a production in the original grammar and the left-hand sides of productions also have the same structure. The only difference is a more fine-grained specification of usage of productions at specific positions in left-hand sides. After parsing the production numbers can be used to construct parse trees over the original grammar.

Compare this to the usual encoding using extra non-terminals and extra chain productions. To help the comparison the production numbers have been added.

```
[\43]     -> "+"        [\258]
[\42]     -> "*"        [\259]
[a-z]     -> F          [\260]
T "*" F -> T            [\261]
F         -> T
E "+" T -> E            [\262]
T         -> E
```

In this grammar chains $F \to T \to E$ are built to include 'simple' expressions into sums. The length of such chains grows with the number of priority levels.

Now consider again our example string `a+b*c`. According to the transformed grammar above, this string has only one parse tree, which is the following:

```
[[\97 -> \260] [\43 -> \258]
 [[\98 -> \260] [\42 -> \259] [\99 -> \260] -> \261]
 -> \262]
```

This corresponds to the tree

$$[[a \to E] + [[b \to E] * [c \to E] \to E] \to E]$$

that is declared by the priority rules.

5 Discussion

We have defined a transformation on context-free grammars that compiles priority and associativity declarations into the productions of the grammar by using character classes to concisely encode sets of productions.

Even though transformed grammars do not generate trees with priority conflict this does not mean that they are unambiguous or do not cause conflicts in parse tables. Conflicts and ambiguities can have been overlooked or caused by constructs that can not be dealt with by means of priorities. See (Visser, 1997d) for further discussions and solutions.

A first prototype of the transformation algorithm has been implemented as part of the implementation of a parser generator for the syntax definition formalism SDF2. The usage of character classes in context-free grammars came natural in this setting because SDF2 integrates lexical and context-free syntax of languages by combining them into a single context-free grammar. Parsers for such grammars do not need separate lexical analyzers and are thus called *scannerless* parsers (Visser, 1997d).

Although the parser generator spends no time on the lookup of information in the priority table, more time is spent on character class computations. An efficient implementation of character classes is therefore essential for a successful implementation.

Further optimizations can be achieved by further transforming the derived character class grammars. An obvious candidate is chain rule elimination. If $[n] \to [m]$ is a chain production, replace everywhere m by n, effectively removing a production from the grammar.

References

Aasa, A. (1992). *User Defined Syntax*. Ph.D. thesis, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden.

Aho, A. V., Johnson, S. C., and Ullman, J. D. (1975). Deterministic parsing of ambiguous grammars. *Communications of the ACM*, **18**(8), 441–452.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, **2**, 113–124.

Earley, J. (1975). Ambiguity and precedence in syntax description. *Acta Informatica*, **4**(1), 183–192.

Floyd, R. W. (1962). Syntactic analysis and operator precedence. *Communications of the ACM*, **5**(10), 316–333.

Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989). The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, **24**(11), 43–75.

Johnson, S. C. (1975). YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, Murray Hill, N.J.

Klint, P. (1988). Definitie van prioriteiten in SDF. Unpublished technical note (in dutch).

Klint, P. and Visser, E. (1994). Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

Lesk, M. E. and Schmidt, E. (1986). *LEX — A lexical analyzer generator*. Bell Laboratories. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

Naur, P. *et al.* (1960). Report on the algorithmic language ALGOL 60. *Communications of the ACM*, **3**(5), 299–314.

Rekers, J. (1992). *Parser Generation for Interactive Environments*. Ph.D. thesis, University of Amsterdam. ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z.

Thorup, M. (1994). Controlled grammatic ambiguity. *ACM Transactions on Programming Languages and Systems*, **16**(3), 1024–1050.

Tomita, M. (1985). *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers.

Visser, E. (1997a). A case study in optimizing parsing schemata by disambiguation filters. In *International Workshop on Parsing Technology (IWPT'97)*, pages 210–224, Boston, USA. Massachusetts Institute of Technology.

Visser, E. (1997b). Character classes. Technical Report P9708, Programming Research Group, University of Amsterdam.

Visser, E. (1997c). A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam.

Visser, E. (1997d). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam.

Ontwerp Tobias Baanders © 1997