Nested collections and polytypism

by

Eerke Boiten and Paul Hoogendijk

ISSN 0926-4515

All rights reserved
editors: prof.dr. R.C. Backhouse
        prof.dr. J.C.M. Baeten

Reports are available at:
http://www.win.tue.nl/win/cs

Computing Science Reports 96/17
Eindhoven, November 1996
Nested collections and polytypism

Eerke Boiten* and Paul Hoogendijk†

November 1996

Abstract

A point-free calculus of so-called “collection types” is presented, similar to the monadic calculus of Tannen, Buneman and Wong. We observe that our calculus is parametrised by a monad thus making the calculus “polytypic”. A novel contribution of the paper is to discuss situations in which a single application involves more than one collection type. In particular, we outline the contribution to database research that may be obtained by exploiting current developments in polytypic programming.

1 Introduction and overview

“Collection types” such as trees, lists, and bags have been studied extensively in computing science. In particular, in the research area of formal program development, the observation (attributed by L.Meertens [19] to H.Boom) that these types form a hierarchy has proved fruitful. The most important aspect of this so-called Boom hierarchy is that a calculus of higher order functions (like map, reduce and filter) can be defined on all its types. This calculus is commonly known as the Bird-Meertens Formalism (BMF) [19, 5], and it is widely used for the development and description of functional and parallel [21] programs. A generalisation of this calculus was found in the category theoretic [18] and relational [14] approaches to abstract data types, and it was observed that the Boom hierarchy types form an instance of another popular category theoretic concept, the monad. This provided a new syntax and calculus for comprehensions on these types.

In the area of databases, the interest in these types arises from the quest for query languages for databases containing structured data. The traditional “flat” relational database model [8] only describes sets of tuples, whose attributes are assumed to be of atomic type

---

*Now at Computing Laboratory, University of Kent, Canterbury Kent CT2 7NF, UK, email: E.A.Boiten@ukc.ac.uk; research was carried out at Eindhoven University of Technology.
†Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, email:paulh@win.tue.nl.
Various "nested" relational calculi\(^1\) (i.e. calculi providing for set-valued attributes) have been proposed [20, 2, 9]. The most general of these is the "monadic" calculus described by Tannen, Buneman and Wong [6, 22]. (The observation that collection types "are" monads with additional properties ("ringads") can be attributed to Wadler [24] and Trinder [23].) They prove that their calculus (using the set monad) is equivalent to "the" nested relational calculus [9].

The calculus we present here is inspired by (and can be instantiated to) that "monadic calculus". Thus, this paper does not claim to directly advance research in database query languages. Rather, our intent is to present the "state of the art" in that area in such a way that it connects more easily with recent developments in formal program development. This explains the differences between our calculus and the monadic calculus: our calculus is non-extensional ("point-free") to facilitate equational reasoning; the monad (functor) involved is an explicit parameter; and instead of an underlying signature of basic functions we assume a category of partial functions with a few extra properties. The importance of these differences has to do with the emerging interest in the area of so-called polytypic programs, the current focus of research in the BMF [3, 4, 11, 15, 12]. Such programs are parametrised by type constructors (as opposed to polymorphic programs which are parametrised by types). Beginning with the work of Malcolm [18] it has been observed that several programming concepts and building blocks can be profitably formulated in polytypic terms thus enhancing their (re)usability.

The presentation of the elements of our calculus is such that it amounts to a constructive proof that it is (in our setting) the smallest orthogonal calculus that can describe the flat relational calculus – or, in other words, our calculus (instantiated with the set monad) is the extension of the flat relational calculus with sets as "first class citizens". The presentation is in two layers: the first layer identifies "tuples" and the basic operations on them in a category of partial functions with a special product. The second layer "lifts" these operations to operations on sets of tuples. Partiality of the basic operations requires some special attention in this lifting procedure.

In the final section of this paper we describe issues for further research, and their relation to research issues in formal program development.

2 Special products: the tuple operations

In relational database theory, a tuple is either a function from a set of labels to a set of "values", or a member of a product type. We choose the latter approach, confident that the strict categorical typing will provide the labels.

We work in a category of partial functions, and write the typing of arrows in such a way that it looks natural for compositions: if \( f : A \leftarrow B \) and \( g : B \leftarrow C \), then \( f \circ g : A \leftarrow C \). Sets are represented as identity arrows, as usual. In particular, for each arrow \( f : A \leftarrow B \)

---

\(^1\)Any reference to "relational calculus" in this paper (except for this footnote...) should be taken to refer to relational database calculus (and the corresponding algebra) rather than Tarski's calculus of binary relations, even though that plays a crucial rôle in polytypic programming.
we assume the existence of an arrow \( f \triangleright B \leftarrow B \) ("domain") which equals the identity on \( f' \)'s domain and is undefined elsewhere. We assume a particular product, whose unique mediating arrow (written as \( J \cdot g \)) is characterised by the following

**Axiom 1 (Product)** \( f \cdot g = h \iff f \circ g = \ll A, B \cdot h \land g \circ f = \gg A, B \cdot h \)

assuming that \( f : A \leftarrow C, g : B \leftarrow C \), with \( \ll A, B \) and \( \gg A, B \) the projections on \( A \times B \). We make product a bifunctor by setting \( f \times g = (f \circ \ll) \triangle (g \circ \gg) \). We take \( ! \) to be a terminal object in the category, with \( ! \) the unique total arrow of type \( 1 \leftarrow A \).

In database theory, tuples are elements of \( n \)-ary products for arbitrary \( n \). To define unique \( n \)-ary products, we make the binary product associative with unit element \( ! \). This is axiomatised by equating the relevant isomorphisms to the appropriate identities:

**Axiom 2** \( ass_{A,B,C} = \text{def} \ (id_A \times \ll B,C) \circ (\gg B,C \circ \gg A,B \times C) = id_A \times B \times C \)

**Axiom 3** \( ! \cdot id_A = id_A = id_A \cdot ! \)

Arrows can only be equal if their types are; thus \( \times \) is associative on objects as well, which justifies writing \( A \times B \times C \) without brackets above.

From the desire to address *any* field of an \( n \)-ary product directly (for example the \( B \) field in \( A \times B \times C \)) with a single projection it follows that product should be commutative as well. However, if there are multiple fields of one and the same type it seems likely that we would wish to distinguish those. So we introduce the following

**Axiom 4 (Semi-commutativity)** Let \( swap_{A,B} = \gg A,B \cdot \ll A,B \); the isomorphism between \( A \times B \) and \( B \times A \). If \( A \) and \( B \) are *relatively prime*, then \( swap_{A,B} = id_{A \times B} \).

Two types are relatively prime when their greatest common divisor is \( ! \). In order to be able to define division, \( / \), and greatest common divisors, we complete our axiomatisation of product by postulating:

**Axiom 5** All objects have a unique prime factorisation.

Now we can address any field directly provided it occurs once: the projection on \( B \) in \( A \) is \( \ll B, A/B \). Since sets are represented by partial identity arrows, it makes sense also to represent predicates for selection ("filtering") in the same way. In particular, we assume the existence of \( equal_A : A \times A \leftarrow A \times A \) which is defined exactly on those pairs \( (a, b) \) of type \( A \times A \) such that \( a = b \). As an abbreviation for the inverse of the duplicating function \( id_A \circ id_A \) we use \( diag_A = \ll A, A \cdot equal_A \). These give us the building blocks for projection and selection on the level of sets in the next section. A final basic operation is (natural) \( join \) of two tuples, which is only defined if they have matching values for fields of the same label (type), and in that case contains the combination of all their fields. Using division, we define it as

\[
join_{A,B} = id_A / C \times id_B / C \times diag_C
\]

where \( C \) is the greatest common divisor of \( A \) and \( B \), and \( A/C \) and \( B/C \) are both required to be relatively prime with \( C \). The result type of \( join_{A,B} \) is the least common multiple of \( A \) and \( B \). For \( A \) and \( B \) relatively prime, the join equals the Cartesian product, as one might expect.
3 Second layer: lifting to sets

Intuitively, we would like to define the operators at the set level as fairly simple set comprehensions. For example, projecting a set $S$ over the type $A \times B$ on $A$ is (using $\cdot$ for function application):

$$\Pi_{A,B}.S = \{\langle A,B.x \mid x \in S \rangle\}$$

The "map" or "apply-to-all" operation on sets will do exactly this, i.e. $\Pi_{A,B} = \text{map}_{\cdot}\langle A,B \rangle$ where we index the map operation with the functor involved, here it is $P$, for powerset. (Also we have $\Pi'_{A,B} = \text{map}_{\cdot}\Rightarrow_{A,B}$ which may differ from $\Pi_{B,A}$ when $A$ and $B$ are not relatively prime.)

However using $\text{map}$ for comprehensions is not good enough in general. For total operations, like projections $\langle A,B \rangle$ there is no problem. However, some of the other functions we would like to lift to the level of sets are partial (for example, predicates encoded as partial identity functions) and that needs to be taken into account. The lifted definition of selection as a comprehension is (assuming $\text{select}_Q$ is the partial identity representing the predicate $Q$)

$$\sigma_Q.S = \{x \mid x \in S \land \text{DEF.}(\text{select}_Q.x)\}$$

where $\text{DEF}$ is a meta-predicate accounting for the partiality of $\text{select}_Q$. The function $\text{map}$ would produce the wrong result if we used it to lift such functions to the level of sets: it would deliver functions that are undefined whenever any of the elements in the set does not satisfy the predicate.

Using some more basic functions on sets we can resolve this problem. The results of the partial function will be "packed" using the singleton set former ($\text{unitp} : P A \leftarrow A$), we return the empty set ($\text{zerop} : P A \leftarrow B$) for values on which the function is undefined, and flatten ($\text{flattenp} : P A \leftarrow PP A$) the resulting set of sets:

$$\text{liftp}.f = \text{flattenp} \circ \text{mapp}.\text{Totp}.f$$

where

$$(\text{Totp}.f).x = \begin{cases} \text{unitp}.f.x & \text{if } \text{DEF.}(f.x) \\ \text{zerop}.x & \text{otherwise} \end{cases}$$

It is easy to prove that this makes $\text{liftp}$ (the arrow part of) a functor that coincides with $\text{map}$ on objects and total functions. $\text{liftp}$ being a functor is a kind of healthiness criterion: it means we can use equational reasoning on the level of function compositions for expressions involving $\text{liftp}$, in particular distribution of functors over composition.

Using $\text{liftp}$, we can define selection by $\sigma_Q = \text{liftp}.\text{select}_Q$. For join, however, we need to assume one more basic function. The function $\text{liftp}.\text{join}_{A,B}$ has type $PC \leftarrow P(A \times B)$ for some type $C$, but the natural join $\mathfrak{M}_{A,B}$ has type $PC \leftarrow PA \times PB$. We need a transformation from $PA \times PB$ to $P(A \times B)$ now, viz. the "crossproduct". This will be defined using the

\[2\text{Actually, } \text{Totp} \text{ is also a functor, viz. from the base category into the corresponding Kleisli category. } \text{Totp} \text{ can for any } F \text{ be expressed without applications and case distinctions if the base category is a boolean division allegory [13].}\]
function that pairs all elements of a set with one particular value, also known as the strength of the powerset functor. Defined as a comprehension, it is

$$\text{strp}(S, x) = \{(y, x) \mid y \in S\}$$

In general the strength of a functor \( F \) is a natural transformation from \((FA) \times B\) to \(F(A \times B)\), which has interesting links with the concept of membership \([4]\). A related function, definable in terms of the strength, is \(\text{stlp}(x, S) = \{(x, y) \mid y \in S\}:\)

$$\text{stlp}_{A,B} = \text{mapp} \circ \text{swap}_{B,A} \circ \text{strp}_{B,A} \circ \text{swap}_{A,B}$$

The crossproduct can then be computed by two applications of these functions: one of each, one nested. (At the level of sets, it does not matter which one is chosen in which position, since there is no order between the elements.\(^3\))

$$\text{crossp} = \text{flatten} \circ \text{mapp} \circ \text{stlp} \circ \text{strp}$$

With this we can define join at the set level:

$$\mathcal{A}_{A,B} = \text{liftp} \circ \text{join}_{A,B} \circ \text{crossp}_{A,B}$$

At this point all standard operators of relational algebra that operate elementwise have been lifted to sets. Finally, one wishes to have the “normal” set operations available as operations on databases. Union is not definable with the current set of primitive operations so we add a primitive \( \text{unionp} \) with one of its characteristic properties: that it forms a monoid with \( \text{zerop} \). Intersection can be defined using cross-product:

$$\cap_A = \text{liftp} \circ \text{diag}_A \circ \text{crossp}_{A,A}$$

and set difference is also expressible using equality (in fact by encoding boolean \( \text{false} \) as the empty set\(^4\)). Let \( \text{notmem} \) be the partial identity function which is only defined on pairs of an element and a set such that the element is not in the set:

$$\text{notmem} = (\text{diag}_A \circ (\text{zerop}_A \land (\text{liftp} \circ \text{diag}_A \circ \text{stlp}_{A,A})))$$

we then have that

$$-A = \text{liftp} \circ (\ll_A) \circ \text{notmem} \circ \text{strp}_{A,A}$$

\(^3\)Note that here follows another Kleisli composition.

\(^4\)Even though we allowed equality test for all types, it must be taken into account that it is an expensive operation on sets. Comparing to the empty set is still “cheap”.

5
4 The complete calculus (a single monad)

Let us summarise the calculus as we have defined it thus far.

The first layer consists of a category of partial functions with a special product (associative with unit \(1\), semi-commutative, unique prime factorisation), which includes arrows \(\text{equal}_A\) for all objects \(A\), and \(f\rightharpoonup\) for all arrows \(f\).

The second (monadic) layer consists of

- a functor \(P\) from the category of the first layer into a category of total functions, whose arrow part is denoted by \(\text{lift}_P;\)
- operations \(\text{unit}_P : PA \rightharpoonup A\) and \(\text{flatten}_P : PA \rightharpoonup PPA\).

Because these operations on sets satisfy the relevant properties for all \(f : B \rightharpoonup A:\)

\[
\begin{align*}
\text{flatten}_P,A \circ \text{mapp.}\text{flatten}_P,A &= \text{flatten}_P,A \circ \text{flatten}_P,PA \\
\text{flatten}_P,A \circ \text{mapp.}\text{unit}_P,A &= \text{flatten}_P,A \circ \text{unit}_P,PA = \text{id}_PA \\
\text{unit}_P,B \circ f &= \text{mapp.}f \circ \text{unit}_P,A \\
\text{flatten}_P,B \circ \text{mapp.(mapp.f)} &= \text{mapp.f} \circ \text{flatten}_P,A
\end{align*}
\]

the structure up to this point is a monad.

Additionally, we have

- an operation \(\text{str}_P,A,B : P(A \times B) \rightharpoonup (PA) \times B\)

Because this operation on sets satisfies the relevant properties:

\[
\begin{align*}
\text{str}_P \circ (\text{mapp.f}) \times g &= \text{mapp.(f \times g)} \circ \text{str}_P \\
\text{str}_P,A,B &= \text{id}_A \\
\text{str}_P,A,B \times C &= \text{str}_P,A \times B,C \circ (\text{str}_P,A,B \times \text{id}_C) \\
\text{str}_P,A,B \circ (\text{unit}_P,A \times \text{id}_B) &= \text{unit}_P,A \times B \\
\text{str}_P,A,B \circ (\text{flatten}_P,A \times \text{id}_B) &= \text{flatten}_P,A \times B \circ \text{mapp.}\text{str}_P,A,B \circ \text{str}_P,PA,B
\end{align*}
\]

the structure up to this point is a strong monad.

Finally, we also have

- operations \(\text{zero}_P : PA \rightharpoonup B\) and \(\text{union}_P : PA \rightharpoonup PA \times PA\)

satisfying the following properties:

\[
\begin{align*}
\text{union}_P \circ (\text{union}_P \times \text{id}) &= \text{union}_P \circ (\text{id} \times \text{union}_P)
\end{align*}
\]
\[ \text{zerop} \circ ! = \text{zerop} \]
\[ \text{unionp} \circ (\text{zerop} \triangle \text{id}) = \text{id} \]
\[ \text{unionp} \circ (\text{id} \triangle \text{zerop}) = \text{id} \]
\[ \text{mapp}.f \circ \text{zerop} = \text{zerop} \]
\[ \text{flattenp} \circ \text{mapp}.\text{zerop} = \text{zerop} \]

\[ \text{flattenp} \circ \text{mapp}.f \circ \text{unionp} = \text{unionp} \circ ((\text{flattenp} \circ \text{mapp}.f) \times (\text{flattenp} \circ \text{mapp}.f)) \]

making the entire structure a **strong ringad** [24].

This equals the calculus \( \mathcal{M}_{\lambda}(=,\text{cond}) \) of Tannen et al [6, 7], whose expressive power equals that of the nested relational algebra with equality test [20], see [6, 7] for a proof. An important property of this calculus is that the complexity of a query is exponential in the size of its input. The nesting and unnesting operations, specified using comprehensions, are:

\[
\begin{align*}
\text{nest}. S &= \{(x, T) \mid T = \{y \mid (x, y) \in S\} \land T \neq \emptyset\} \\
\text{unnest}. T &= \{(x, y) \mid \exists (S : (x, S) \in T : y \in S)\}
\end{align*}
\]

In our calculus they are the beauty:

\[ \text{unnestp}_{A, B} = \text{flattenp}_{A \times B} \circ \text{mapp}.\text{stlp}_{A, B} \]

and the beast (translated from [22]):

\[ \text{nest}_{A, B} = \text{mapp}.((\alpha_{A, P(A \times B)} \circ f) \circ \text{str}_{A, P(A \times B)} \circ (\Pi_{A, B} \triangle \text{id}_{P(A \times B)}) \circ \text{stlp}_{A, A \times B}) \quad \text{where} \]

\[ f = \Pi'_{A \times A, B} \circ \text{lift}_{A}.((\text{equal}_{A} \times \text{id}_{B}) \circ \text{stlp}_{A, A \times B}) \]

Note that this nested algebra came about naturally from the wish to express the standard operations of relational algebra as extensions of operations on tuples. That we are now able to express the entire nested relational algebra tells us that this algebra is in a sense the smallest orthogonal extension of the flat relational algebra. An informal corollary of this is the conclusion that “first normal form” is a rather artificial restriction (in this setup).

Moreover, we have not needed any other properties of the operations used, besides those listed above. Thus, a monadic calculus of this form can be defined for any strong ringad. Important examples of these are lists and bags with their obvious (certainly for Boom hierarchy adepts) operations. For this reason, Trinder [23] and Tannen et al [6, 22] have argued that strong ringads describe the essence of so-called bulk- or collection types in databases.

As an aside, not all operations defined above generalise from sets to other collection types all that well. Consider for example nesting/unnesting on lists, where \( L \) is the list functor. For sets we had that \( \text{unnestp} \circ \text{nestp} = \text{id} \). For lists \( \text{nest}_{L} \) is an injective function, but \( \text{unnest}_{L} \) is not its left inverse:
\[
\text{unnest}_L \cdot \text{nest}_L \cdot [(a,5), (b,1), (a,7)]
\]
\[
= \{ \text{definition } \text{nest}_L \}\}
\]
\[
\text{unnest}_L \cdot [(a,5,7), (b,1), (a,5,7)]
\]
\[
= \{ \text{definition } \text{unnest}_L \}\}
\]
\[
[(a,5), (a,7), (b,1), (a,5), (a,7)]
\]

One could argue that either the nest operation on lists should discard the duplicates (making it no longer injective since order information is lost) or that unnesting should do so (which means that un nest no longer has a simple definition).

5 Multiple monads: research issues

So far, we have only shown that we can define the same calculus for each monad separately. However, ideally one should have several of these collection types available in one language, and be able to write mixed expressions. From simply combining the set and list versions of the calculus, we already get expressions like \( \text{map}. \text{map}_L.f : \text{PLB} \rightarrow \text{PLA} \) but there is no option yet to move from one datatype to another in general. To our knowledge, this has not been an issue for research in the database programming languages community.

Can general conversion functions between arbitrary strong ringads exist? Unfortunately, the answer to that question is “no”. For example, to define a deterministic conversion from bags to lists one needs an order on the element type, which need not exist. On the other hand, a rather pessimistic approach to this issue is based on the observation that all well-known examples of collection types can be viewed as implementations of sets. This implies that for each such type \( F \) a polymorphic function (i.e., a natural transformation) \( \text{setify}_F : \text{PA} \rightarrow \text{FA} \) exists. We could impose the existence of \( \text{setify} \) as an additional requirement on collection types – [4] gives a definition of \( \text{setify}_F \) in terms of the membership relation of \( F \), which is strongly related to the strength of \( F \). So, within the current context, this may not be a severe restriction. However, more polymorphic transformations are known to exist, e.g. there is also a polymorphic transformation from lists to bags. Generalizing this we could end up with something like a category of collection types (extending the Boom hierarchy), with polymorphic data type transformations as arrows and possibly the powerset type \( P \) as a terminal object with \( \text{setify}_F \) as its unique arrow.

A completely different class of useful operations for interfacing several data types is formed by operations that commute functors, i.e. that convert \( FG \)-structures into \( GF \)-structures. Our group has studied such operations [3], and called them “zips”, after the well known operator that turns a pair of lists of equal length into a list of pairs. The special case where \( G \) is the powerset functor \( P \) has been studied by de Moor [10] under the name of “cross operators”. Most zips are not definable in our language, an exception being \( \text{zip}, \times \), which is identical to \( \text{crossp} \) on its domain.\(^5\) (In general \( \text{cross}^F \) is not related

\(^5\)The cross product of one empty and one non-empty set is empty, but the zip of those two is undefined since they are not of the same shape.
to $\text{zip}_{F \times}$ though, as can be observed for $F$ being the list functor $L$.) Another interesting example is $\text{zip}_{F \times F}$ which is a kind of transpose operator: $\text{zip}_{L \times L}$ turns a list of $m$ lists of length $n$ into a list of $n$ lists of length $m$ in the obvious way.\textsuperscript{6}

A final example of the use of zips is in the approach to query languages for databases with partial information advocated by Libkin and Wong [17, 16]. They combine a collection type (sets or bags) with a version of sets that has a non-standard interpretation (but is otherwise identical), the so-called or-sets. An or-set conceptually represents one of the values in it. For normalisation of expressions containing tuples, ordinary sets/bags and or-sets, the \textit{flatten} and \textit{str} operations for or-sets can be used (check that these preserve the conceptual meaning of such expressions). At the heart of normalisation is an operation called $\alpha$ of type $\text{Or}(PA) \leftarrow \text{P}(\text{Or}A)$ which essentially translates conjunctive normal form into disjunctive normal form. It can be defined in terms of the corresponding operator for bags, $b_{\alpha} : \text{Or}(\text{Bag}A) \leftarrow \text{Bag}(\text{Or}A)$ which is $\text{zip}_{\text{Bag}, \text{Or}}$ on bags of non-empty or-sets. It should be noted, though, that unlike any of the other operators, $\text{zip}$ can have a complexity that is exponential in the size of its arguments.

Altogether, it seems that more research is needed into zips and other data type transformations before a more conclusive form of a query language with multiple collection types can be established. We expect that research done in formal program development on zips [3], the Boom hierarchy [19, 14] of datatypes, and polytypic programming in general [4, 11, 15] will provide a basis for this further research.

6 Concluding remarks

We have presented a monadic calculus for querying nested collections, inspired by (and in some sense equivalent to) the ones defined by Tannen, Buneman and Wong [6, 7] and Trinder [23]. Our presentation was designed to connect theories from formal program development with the state of the art in database query languages. Thus, a set up of functors over partial functions with associated natural transformations was chosen. Large portions of equational (point-free) calculi for category theory, for (total) functions [19, 5], and for binary relations [1] are directly applicable to our calculus. Much of these will translate directly to well-known or possibly even new optimisations of (nested) relational database queries. The calculus instantiated to a single monad appears to be complete and well understood, cf. [7]. However, having the monad (functor) as an explicit parameter induces the question of how these calculi could be combined for several monads, leading to a query language for databases involving multiple collection types. Our partial answer to this question showed that this strongly relates to several issues currently studied in the area of formal program development, most importantly to polytypism [4].

We hope that future research will continue the cross-fertilisation of these two areas, with investigation of data types, their operations, and transformations between them taking a central place.

\textsuperscript{6}For $\text{zip}_{F \times G}$ we have $\text{zip}_{F \times G} \circ \text{map}_{F \times \text{stl}_{G}} \circ \text{str}_{F} = \text{map}_{G} \circ \text{str}_{F} \circ \text{stl}_{G}$. This clearly relates the definition of \textit{cross} with the alternative one where \textit{the other index runs faster}.
Acknowledgements

Val Tannen’s support in our exploration of this research area is much appreciated. Our colleagues in the Mathematics of Program Construction section in Eindhoven and Simon Thompson at UKC gave many useful comments on this paper and other presentations we have given of this material. We thank Leonid Libkin for his helpful responses to our questions on or-sets.

References


<table>
<thead>
<tr>
<th>Computing Science Reports</th>
<th>Department of Mathematics and Computing Science</th>
</tr>
</thead>
<tbody>
<tr>
<td>In this series appeared:</td>
<td>Eindhoven University of Technology</td>
</tr>
</tbody>
</table>

- 93/01 R. van Geldrop
- 93/02 T. Verhoeff
- 93/03 T. Verhoeff
- 93/04 E.H.L. Aarts
  - J.H.M. Korst
  - P.J. Zwetereing
- 93/05 J.C.M. Baeten
  - C. Verhoef
- 93/06 J.P. Velthamp
- 93/07 P.D. Moerland
- 93/08 J. Verhoosel
- 93/09 K.M. van Hee
- 93/10 K.M. van Hee
- 93/11 K.M. van Hee
- 93/12 K.M. van Hee
- 93/13 K.M. van Hee
- 93/14 J.C.M. Baeten
  - J.A. Bergstra
- 93/15 J.C.M. Baeten
  - J.A. Bergstra
  - R.N. Böf
- 93/16 H. Schepers
  - J. Hoornan
- 93/17 D. Alstein
  - P. van der Stok
- 93/18 C. Verhoeff
- 93/19 G-J. Houbten
- 93/20 F.S. de Boer
- 93/21 M. Codish
  - D. Dams
  - G. Filé
  - M. Bruynooghe
- 93/22 E. Poll
- 93/23 E. de Kogel
- 93/24 E. Poll and Paula Severi
- 93/25 H. Schepers and R. Gerth
- 93/26 W.M.P. van der Aalst
- 93/27 T. KIoks and D. Kratsch
- 93/28 F. Kamareddine and R. Nederpelt
- 93/29 R. Post and P. De Bra
- 93/30 J. Deogun
  - T. KIoks
  - D. Kratsch
  - H. Müller

Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
A continuous version of the Prisoner's Dilemma, p. 17
Quicksort for linked lists, p. 8.
Deterministic and randomized local search, p. 78.
A congruence theorem for structured operational semantics with predicates, p. 18.
On the unavoidability of metastable behaviour, p. 29
Exercises in Multiprogramming, p. 97
A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
The Design of an Online Help Facility for ExSpect, p. 21.
Freeness Analysis for Logic Programs - And Correctness, p. 24
A Typechecker for Bijective Pure Type Systems, p. 28.
Relational Algebra and Equational Proofs, p. 23.
Pure Type Systems with Definitions, p. 38.
Multi-dimensional Petri nets, p. 25.
Finding all minimal separators of a graph, p. 11.
A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
GOLD, a Graph Oriented Language for Databases, p. 42.
On Vertex Ranking for Permutation and Other Graphs, p. 11.
<table>
<thead>
<tr>
<th>Page</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>93/31</td>
<td>W. Korver</td>
<td>Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.</td>
</tr>
<tr>
<td>93/33</td>
<td>L. Loyens and J. Moonen</td>
<td>ILLIAS, a sequential language for parallel matrix computations, p. 20.</td>
</tr>
<tr>
<td>93/34</td>
<td>J.C.M. Baeten and J.A. Bergstra</td>
<td>Real Time Process Algebra with Infinitesimals, p.39.</td>
</tr>
<tr>
<td>93/36</td>
<td>J.C.M. Baeten and J.A. Bergstra</td>
<td>Non Interleaving Process Algebra, p. 17.</td>
</tr>
<tr>
<td>93/37</td>
<td>J. Brunecke and J-P. Katoen</td>
<td>Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.</td>
</tr>
<tr>
<td>93/38</td>
<td>C. Verhoef</td>
<td>A general conservative extension theorem in process algebra, p. 17.</td>
</tr>
<tr>
<td>93/39</td>
<td>W.P.M. Neijten, E.H.L. Autri,</td>
<td>Job Shop Scheduling by Constraint Satisfaction, p. 22.</td>
</tr>
<tr>
<td></td>
<td>D.A.A. van Erp Taalman Kip,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>K.M. van Hoo</td>
<td></td>
</tr>
<tr>
<td>93/41</td>
<td>A. Bijlsma</td>
<td>Temporal operators viewed as predicate transformers, p. 11.</td>
</tr>
<tr>
<td>93/42</td>
<td>P.M.P. Rambags</td>
<td>Automatic Verification of Regular Protocols in PT Nets, p. 23.</td>
</tr>
<tr>
<td>93/43</td>
<td>B.W. Watson</td>
<td>A taxonomy of finite automata construction algorithms, p. 87.</td>
</tr>
<tr>
<td>93/44</td>
<td>B.W. Watson</td>
<td>A taxonomy of finite automata minimization algorithms, p. 23.</td>
</tr>
<tr>
<td>93/48</td>
<td>R. Gerth</td>
<td>Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.</td>
</tr>
<tr>
<td>94/01</td>
<td>P. America, M. van der Kamenen, R.P. Nederpelt, O.S. van Roosmalen, H.C.M. de Swart</td>
<td>The object-oriented paradigm, p. 28.</td>
</tr>
<tr>
<td>94/02</td>
<td>F. Kamareddine, R.P. Nederpelt</td>
<td>Canonical typing and Π-conversion, p. 51.</td>
</tr>
<tr>
<td>94/04</td>
<td>J.C.M. Baeten and J.A. Bergstra</td>
<td>Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.</td>
</tr>
<tr>
<td>94/06</td>
<td>T. Bazen, T. Kunz, M. Coffin, D. Taylor</td>
<td>Time and the Order of Abstract Events in Distributed Computations, p. 29.</td>
</tr>
<tr>
<td>94/08</td>
<td>O.S. van Roosmalen</td>
<td>A Hierarchical Diagrammatic Representation of Class Structure, p. 22.</td>
</tr>
<tr>
<td>94/09</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Process Algebra with Partial Choice, p. 16.</td>
</tr>
</tbody>
</table>
T. Verheoef

J. Peleska
C. Huizing
C. Petersohn

T. Kloks
D. Kratsch
H. Müller

R. Seljée

R.J.M. Vaessens
E.H.L. Aarts
J.K. Lenstra

R.C. Backhouse
H. Doornbos

S. Mauw
M.A. Reniers

F. Kamareddine
R. Nederpelt

B.W. Watson

R. Bloo
F. Kamareddine
R. Nederpelt

B.W. Watson

B.W. Watson

S. Mauw and M.A. Reniers

D. Dams
O. Grumberg
R. Gerth

C. W.A.M. van Overveld
M. Verhoeven

J.C.M. Baeten
J.A. Bergstra
Gh. Stefanescu

B.W. Watson
R.E. Watson

S. Mauw and H. Mulder

C.W.A.M. van Overveld
M. Verhoeven

J. Hooman

J.C.M. Baeten
J.A. Bergstra
Gh. Stefanescu

B.W. Watson
R.E. Watson

J.J. Vereijken

T. Laan

R. Bloo
F. Kamareddine
R. Nederpelt

J.C.M. Baeten
S. Mauw

F. Kamareddine
R. Nederpelt

T. Basten
R. Bol
M. Voorhoeve

A. Bijlsma
C.S. Scholten

The testing Paradigm Applied to Network Structure. p. 31.


A New Method for Integrity Constraint checking in Deductive Databases. p. 34.

Ups and Downs of Type Theory, p. 9.

Job Shop Scheduling by Local Search, p. 21.

Mathematical Induction Made Calculational. p. 36.

An Algebraic Semantics of Basic Message Sequence Charts. p. 9.

Refining Reduction in the Lambda Calculus, p. 15.

The performance of single-keyword and multiple-keyword pattern matching algorithms. p. 46.

Beyond β-Reduction in Church’s λ →, p. 22.


The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.

An algebraic semantics of Message Sequence Charts, p. 43.

Abstract Interpretation of Reactive Systems: Abstractions Preserving VCTL*, ECTL* and CTL*, p. 28.

Kω-free and W4-free graphs, p. 10.

On the foundations of functional programming: a programmer’s point of view, p. 54.


Correctness of Real Time Systems by Construction, p. 22.

Process Algebra with Feedback, p. 22.

A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.


A formalization of the Ramified Type Theory, p.40.

The Barendregt Cube with Definitions and Generalised Reduction, p. 37.

Delayed choice: an operator for joining Message Sequence Charts, p. 15.

Canonical typing and II-conversion in the Barendregt Cube, p. 19.

Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.

Point-free substitution, p. 10.
On the equivalence covering number of splitgraphs, p. 4.
Distributed Consensus and Hard Real-Time Systems, p. 34.
Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
Concatenation of Graphs, p. 7.
Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.
Verifying Sequentially Consistent Memory, p. 160.
Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek”, p. 43.
The λ-cube with classes of terms modulo conversion, p. 16.
On Π-conversion in Type Theory, p. 12.
Fixed-Point Calculus, p. 11.
Process Algebra with Propositional Signals, p. 25.
A short and flexible proof of Strong Normalization for the Calculus of Constructions, p. 27.
Listing simplicial vertices and recognizing chord-free graphs, p. 4.
Traces and Logic, p. 81
A Partial Order Approach to Branching Time Logic Model Checking, p. 20.
The Construction of a small Communication Library, p. 16.
Formalizing Process Algebraic Verifications in the Calculus of Constructions, p. 49.
Concrete process algebra, p. 134.
An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.
A Type Inference Algorithm for Pure Type Systems, p. 20.
A Quantitative Analysis of Iterated Local Search, p. 23.
Drawing Execution Graphs by Parsing, p. 10.
Preservation of Strong Normalisation for Explicit Substitution, p. 12.
Discrete Time Process Algebra, p. 20.
| 95/11 | R. Seljée | Deductive Database Systems and integrity constraint checking, p. 36. |
| 95/12 | S. Mauw and M. Reniers | Empty Interworkings and Refinement Semantics of Interworkings Revised, p. 19. |
| 95/14 | A. Ponse, C. Verhoef, S.F.M. Vlijmen (eds.) | De proceedings: ACP'95, p. |
| 95/16 | D. Dams, O. Grumberg, R. Gerth | Abstract Interpretation of Reactive Systems: Preservation of CTL*, p. 27. |
| 95/17 | S. Mauw and E.A. van der Meulen | Specification of tools for Message Sequence Charts, p. 36. |
| 95/19 | J.C.M. Baeten and J.A. Borgstra | Discrete Time Process Algebra with Abstraction, p. 15. |
| 95/20 | F. van Raamsdonk and P. Severi | On Normalisation, p. 33. |
| 95/21 | A. van Deursen | Axiomatising Early and Late Invar by Variable Elimination, p. 44. |
| 95/23 | W.M.P. van der Aalst | Petri net based scheduling, p. 20. |
| 95/25 | L. Feijs | Synchronous Sequence Charts In Action, p. 36. |
| 95/26 | W.M.P. van der Aalst | A Class of Petri nets for modeling and analyzing business processes, p. 24. |
| 95/32 | W.M.P. van der Aalst | Handboek simulatie, p. 51. |
| 95/33 | J. Engelfriet and J.J. Vereijken | Context-Free Graph Grammars and Concatenation of Graphs, p. 35. |
| 95/34 | J. Zwanenburg | Record concatenation with intersection types, p. 46. |
| 95/35 | T. Basten and M. Voorhoeve | An algebraic semantics for hierarchical PT Nets, p. 32. |
| 96/01 | M. Voorhoeve and T. Basten | Process Algebra with Autonomous Actions, p. 12. |
| 96/02 | P. de Bra and A. Aerts | Multi-User Publishing in the Web: DresS, A Document Repository Service Station, p. 12 |
| 96/04 | S. Mauw | Example specifications in phi-SDL. |
| 96/06 | W.M.P. van der Aalst and T. Basten | Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18. |
| 96/07 | M. Voorhoeve | Structural Petri Net Equivalence, p. 16. |
| 96/11 | T. Laan | AUTOMATH and Pure Type Systems, p. 30. |
| 96/12 | F. Kamareddine and T. Laan | A Correspondence between Nuprl and the Ramified Theory of Types, p. 12. |
| 96/13 | T. Borchgrys | Priorean Tense Logics in Modal Pure Type Systems, p. 61 |
96/14  S.H.J. Bos and M.A. Reniers  The /² C-bus in Discrete-Time Process Algebra, p. 25.
96/15  M.A. Reniers and J.J. Vereijken  Completeness in Discrete-Time Process Algebra, p. 139.
96/16  P. Hoogendijk and O. de Moor  What is a data type?, p. 29.