Executable Calculational Expressions

Francisco Cháves and Camilo Rocha

Escuela Colombiana de Ingeniería, Bogotá, Colombia

Abstract. The calculational style of E. W. Dijkstra and C. S. Scholten is a semi-formal style for the development, both in terms of verification and derivation, of correct programs. This calculational style heavily relies on the symbolic manipulation of expressions involving, for instance, arithmetic, quantifiers, and collections. This manuscript presents *Calculational*, an executable specification language for a broad class of calculational expressions in the style of Dijkstra & Scholten. The *Calculational* tool offers support for executable specifications, including: (i) quantifier expressions such as summation and universal/existential quantification, and (ii) data types such as lists, sets, and bags (i.e., multisets). The current implementation of *Calculational* is executable in the Haskell programming language, offering support also for higher order computation and pattern matching. This implementation is available for download as open source code and as a Haskell library too (which can be embedded in other programming languages). The main features of *Calculational*, some examples of its use —including the formal processing and querying of relational data—, and details of its implementation are also included.

1 Introduction

Nowadays, computers are part of planes, cars, refrigerators, TVs, phones, clocks, etc.; and software is at the heart of such devices. Bugs in software are common and they can have extremely seriuos consecuenses. Formal methods has been developed in order to minimize (or completely eradicate) the existence of software bugs by providing mathematical based techniques and tools.

The calculational style is a semi-formal style of program verification and derivation originated by the work of E. W. Dijkstra [7] and C. A. R. Hoare [10]. Its formal foundation was consolidated in the work of E. W. Dijkstra and C. S. Scholten [8], under the name of Dijkstra-Scholten logic. These foundations have been thoroughly and actively studied from different perspectives [2,12,21,17], and even extended to other logics [3]. Computability properties of the Dijkstra-Scholten logic have also been explored recently [18,19]. The calculational style has also had impact in the teaching of logic and discrete mathematics [9], and program construction [1]. This report presents *Calculational*, a language and an implementation for executing constructs commonly found in specifications in the style of Dijkstra & Scholten.

Consider a 0-based array $a[0..N)$ of integer values and the following specification [1, p. 187] for computing the summation of values in a:

`Precondition:` $0 \leq N$
`Postcondition:` $s = (\Sigma i \mid 0 \leq i < N : a[i])$

Computing the summation of values in an array is an elementary exercise in the derivation of algorithms. One viable approach is to use the technique commonly known as *changing a constant by a variable in the postcondition*, with the goal of obtaining a more general problem. In principle, this approach seems counter-intuitive, but in the end it is useful. In this particular case, it is convenient to use a fresh variable n instead of the constant N in the summation of values of a and in an intermediate state of the algorithm, s has the partial sums of the $a[i]$ elements $0 \leq i < n$. At the end, when n reach the N value, the desired summation has been computed. Actually, this can be done directly in *Calculational* by introducing a function (in the syntax of the Haskell programming language), say `sumA`, that takes as input the array a and the variable n as follows:

```
sumA :: (Num e) => Array Int e -> Int -> e
sumA a n = [calc| (Σ i <- [0 .. n-1] | : a!i) |]
```

Wrapper `[calc| ... ]` is used to embed calculational expressions in the current implementation of *Calculational*. Expression `(Σ i <- [0 .. n-1] | : a!i)` has the same syntactic structure and semantics of generalized summation in the style of Dijkstra & Scholten: it denotes the summation of the first n values of a. As it turns out and thanks to *Calculational*, function `sumA` is an executable specification of a calculational construct, namely, of a generalized sum. As such, the *executable* specification `sumA` a N solves the summation problem specified above.

The *Calculational* language and implementation feature support for many calculational constructs such as arithmetic and Boolean expressions, generalized operations (such as quantifiers and generalized unions), and several collections (such as arrays, lists, sets, and multisets). Generalized operations and collections can be defined with the help of multiple generators, pattern matching, and lambda abstractions. With *Calculational*, all of these specifications can be embedded in the hosting programming language and be executed with the expected semantics.

These kind of expressions could be used in assertions in order to verify if algorithms satisfy the desired preconditions or to test if a given postcondition is satisfied by an algorithm. On the other hand, these expressions can be used to construct correct programs to calculate results.

This report presents examples on the use of *Calculational* in several domains. These examples include linear search, sorting, combinatorial generation, and

graph-theoretic specifications. More elaborated examples on the processing and querying of relational information are also included.

The current implementation of *Calculational* has been done in the Haskell programming language and has been tested with the Glasgow Haskell Compiler (*gch*), one of the most popular implementation of this language. It is available as Haskell source code and also as a library that can be embedded in other programming languages. Internally, the implementation of *Calculational* heavily relies on Haskell constructs such as templates and quasi-quotation.

Outline. This manuscript is organized as follows. Section 2 presents *Calculational* language in BNF-like notation and Section 3 some examples of its use. Section 4 presents some complementary examples on relational data processing and querying. Section 5 presents implementation details and, finally, Section 6 presents some concluding remarks.

2 Calculational Language

This section presents the language supported by *Calculational* in the form of extended BNF-like expressions. The following conventions are adopted:

- Non-terminal symbols are enclosed by angle brackets (i.e., by '⟨' and '⟩').
- Terminal symbols are annotated with **boldface**.
- Curly brackets denote *zero or more repetitions* of the enclosed expression.
- Square brackets denote *at most one repetition* of the enclosed expression.
- A production rule of the form $\langle \mathtt{e} \rangle ::= \langle \mathtt{e_1} \rangle \ \{\oplus \ \langle \mathtt{e_1} \rangle\}$ denotes that '$\oplus$' is a left-associative operator.
- A production rule of the form $\langle \mathtt{e} \rangle ::= \langle \mathtt{e_1} \rangle \ [\oplus \ \langle \mathtt{e} \rangle]$ denotes that '$\oplus$' is a right-associative operator.
- Operator precedence is given by the presentation order of productions, i.e., an operator defined in the last production rule has more precedence than one defined earlier.

The set of expressions is denoted by $\langle \mathtt{expr} \rangle$, which is defined in the rest of the section.

2.1 Explicit Function Application

Explicit function application is denoted *à la* Haskell by **$**. Since **$** is the first operator defined, explicit function application has the least precedence among all operators in the language of *Calculational*.

⟨`expr`⟩ ::= ⟨`extBinOpExpr`⟩ [**$** ⟨`expr`⟩]

Note that **$** differs from the usual '**.**' in the Dijkstra-Scholten style. Therefore, an expression such as $f.x$ is written $f\$x$ in *Calculational*.

2.2 External Operators

Calculational offers support for operators at two levels. The first level consists of internal operators that correspond to operators which semantics is directly defined in *Calculational*. The second level consists of *external* (i.e., *alien*) operators defined by the environment in which *Calculational* is compiled or executed. Operationally, *Calculational* first tries to resolve a token operator against the internal operators and then, if the search fails, such a token is considered an alien operator which semantics is taken from the environment. Because of the lower precedence of external operators, the production rule that defines them comes before the one for internal operators.

External operators are denoted by ⟨`extOp`⟩ which represents any symbol not included in the following collection of *reserved* symbols:

: **|** **|:** **::** **$** **,** **..**

The following production rule defines expressions that can include external operators:

⟨`extBinOpExpr`⟩ ::= ⟨`boolExpr`⟩ [⟨`extOp`⟩ ⟨`extBinOpExpr`⟩]

Note that because ⟨`extBinOpExpr`⟩ is defined right-associative the precedence and associative behavior of Haskell operators is disregarded operationally. This means that if an expression contains several external operators they all are resolved in a right-associative way and with the same precedence. Therefore, parenthesis are required for inducing precedence over external operators.

2.3 Logical Expressions

Boolean expressions are the top-most type of expressions in *Calculational* and are denoted by ⟨`boolExpr`⟩. Boolean expressions are composed from the usual

Dijkstra-Scholten operators: equivalence, inequivalence, implication, consequence, conjunction, and disjunction.

$$
\begin{aligned}
\langle \texttt{boolExpr} \rangle &::= \langle \texttt{boolImpExpr} \rangle \, \{ \equiv \langle \texttt{boolImpExpr} \rangle \} \\
&\mid \langle \texttt{boolImpExpr} \rangle \, \{ \not\equiv \langle \texttt{boolImpExpr} \rangle \} \\
\langle \texttt{boolImpExpr} \rangle &::= \langle \texttt{boolConsExpr} \rangle \, [\Rightarrow \langle \texttt{boolImpExpr} \rangle] \\
\langle \texttt{boolConsExpr} \rangle &::= \langle \texttt{boolConjDisExpr} \rangle \, \{ \Leftarrow \langle \texttt{boolConjDisExpr} \rangle \} \\
\langle \texttt{boolConjDisExpr} \rangle &::= \langle \texttt{conjuntiveExpr} \rangle \, \langle \texttt{boolConjDisExprC} \rangle \\
\langle \texttt{boolConjDisExprC} \rangle &::= \{ \wedge \langle \texttt{conjuntiveExpr} \rangle \} \mid \{ \vee \langle \texttt{conjuntiveExpr} \rangle \})
\end{aligned}
$$

The Boolean operators have the usual semantics in *Calculational*. Note that in the production rule $\langle \texttt{boolExpr} \rangle$ conjunction and disjunction have the same precedence and cannot be mixed.

2.4 Conjunctive Expressions

Conjunctive expressions are a specific type of Boolean expressions and are denoted by $\langle \texttt{conjuntiveExpr} \rangle$.

$$
\begin{aligned}
\langle \texttt{conjuntiveExpr} \rangle &::= \langle \texttt{concatExpr} \rangle \, \{ \langle \texttt{relOp} \rangle \, \langle \texttt{concatExpr} \rangle \} \\
\langle \texttt{relOp} \rangle &::= \texttt{==} \mid \texttt{/=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \\
&\quad \mid \subset \mid \subseteq \mid \supset \mid \supseteq \mid \not\subset \mid \not\subseteq \mid \not\supset \mid \not\supseteq \\
&\quad \mid \in \mid \notin
\end{aligned}
$$

Sequences of expressions connected by *relational operators* in $\langle \texttt{relOp} \rangle$ have a conjunctive reading. For example, if $\bullet_i$ and e_j are, respectively, relational operators and expressions, for $0 \leq i < n$ and $0 \leq j \leq n$, then the Boolean expression

$$e_0 \ \bullet_0 \ e_1 \ \bullet_1 \ \cdots \ \bullet_{n-1} \ e_n$$

is interpreted as

$$(e_0 \bullet_0 e_1) \wedge (e_1 \bullet_1 e_2) \wedge \cdots \wedge (e_{n-1} \bullet_{n-1} e_n).$$

2.5 Sequences

Sequences of expressions in *Calculational* are denoted by $\langle \texttt{concatExpr} \rangle$. Two sequences are concatenated with the usual Haskell notation ++. A sequence can

be appended and extended with an element with, respectively, ▷ and ◁.

⟨concatExpr⟩ ::= ⟨appendExpr⟩ {++ ⟨appendExpr⟩}
⟨appendExpr⟩ ::= ⟨prependExpr⟩ { ▷ ⟨prependExpr⟩}
⟨prependExpr⟩ ::= ⟨countExpr⟩ [◁ ⟨prependExpr⟩]

2.6 Number of Occurrences in a Bag

The binary sharp operator # computes the number of occurrences of an expression in a finite bag.

⟨countExpr⟩ ::= ⟨maxminExpr⟩ {# ⟨maxminExpr⟩}

2.7 Maximum and Minimum

The binary maximum ↑ and binary minimum ↓ operators compute, respectively, the maximum and minimum of two expressions. It is noted that the two expressions being compared must be comparable.

⟨maxminExpr⟩ ::= ⟨addExpr⟩ ⟨maxminExprC⟩
⟨maxminExprC⟩ ::= {↑ ⟨infixExpr⟩} | {↓ ⟨infixExpr⟩}

2.8 Infix Expressions

Calculational supports the use of infix notation by following the usual Haskell convention of back-quotes.

⟨infixExpr⟩ ::= ⟨unionInterExpr⟩ {` ⟨identifierExpr⟩ ` ⟨unionInterExpr⟩}

Evaluation of infix expressions are performed as the evaluation of expressions involving external operators (see Section 2.2).

2.9 Union, Intersection, and Difference

The union ∪, intersection ∩, and difference \ operators are defined over all collections supported in *Calculational* (see Section 2.19).

⟨unionInterExpr⟩ ::= ⟨diffExpr⟩ ⟨unionInterExprC⟩
⟨unionInterExprC⟩ ::= {∩ ⟨diffExpr⟩} | {∪ ⟨diffExpr⟩}
⟨diffExpr⟩ ::= ⟨addExpr⟩ {\ ⟨addExpr⟩}

2.10 Arithmetic Expressions

The arithmetic expressions denoted by ⟨`addExpr`⟩ are defined as follows:

⟨`addExpr`⟩ ::= ⟨`multExpr`⟩ {⟨`addOp`⟩ ⟨`multExpr`⟩}
⟨`addOP`⟩ ::= `+` | `-`
⟨`multExpr`⟩ ::= ⟨`powExpr`⟩ {⟨`multOp`⟩ ⟨`powExpr`⟩}
⟨`multOP`⟩ ::= `*` | `/` | `÷` | `%`
⟨`powExpr`⟩ ::= ⟨`compExpr`⟩ {⟨`powOp`⟩ ⟨`compExpr`⟩}
⟨`powOP`⟩ ::= `^` | `**`

The `÷` represents integer division, `^` represents exponentiation with an integer number, and `**` represents exponentiation with a floating point exponent. Also, remainder operator is included with the infix construction with back-quotes (e.g., e_1 ‘mod‘ $e2$). Note that operator `%` (defined in Haskell's `Data.Ratio` library [15]) has a different meaning than in C-like programming languages: in Haskell the expression $e_1 \% e_2$, with e_1 and e_2 integer numbers, denotes the rational number $\frac{e_1}{e_2}$.

Because infix precedence is lower than arithmetic precedence, parenthesis must be used to set the correct precedence in a expression when necessary.

2.11 Function Composition and Function Application

Calculational uses `.` to denote function composition. Juxtaposition (i.e., the empty syntax) and `$` are both used for function application (see Section 2.1), i.e., expressions e_1 \$ e_2 and e_1 e_2 have the same meaning. Operator `$`, because it has lower precedence, is preferred over juxtaposition to avoid extra parenthesis when expressions e_1 or e_2 are complex.

⟨`compExpr`⟩ ::= ⟨`appExpr`⟩ [`.` ⟨`compExpr`⟩]
⟨`appExpr`⟩ ::= ⟨`factor`⟩ { ⟨`factor`⟩}

2.12 Factors

Factor expressions are the atomic building blocks in *Calculational*, also including the conditional and some unary expressions.

⟨factor⟩ ::= ⟨stringExpr⟩ | ⟨charExpr⟩ | ⟨numberExpr⟩ | ⟨emptyExpr⟩
| ⟨ifExpr⟩ | ⟨negExpr⟩ | ⟨notExpr⟩ | ⟨sharpExpr⟩
| ⟨identifierExpr⟩ | ⟨constructorExpr⟩ | ⟨lambdaExpr⟩
| ⟨parentExpr⟩ | ⟨setExpr⟩ | ⟨listExpr⟩ | ⟨bagExpr⟩

2.13 Literals

The following rules are defined for literals:

⟨stringExpr⟩ ::= "{⟨char⟩}"
⟨charExpr⟩ ::= '⟨char⟩'
⟨numberExpr⟩ ::= ⟨number⟩
⟨number⟩ ::= ⟨digit⟩ {⟨digit⟩}[. ⟨digit⟩ {⟨digit⟩}]
⟨emptyExpr⟩ ::= ∅

The non-terminals ⟨char⟩ and ⟨digit⟩ are, respectively, any Haskell character and any decimal number digit between 0 and 9, inclusive. Symbol ∅ represents the empty collection.

2.14 Identifiers

An identifier is a variable name beginning with a lowercase letter or a constructor name that begins with an uppercase letter. Both variable and constructor names are looked up in the environment in which *Calculational* is deployed.

⟨identifierExpr⟩ ::= ⟨nameSpace⟩ ⟨lower⟩ {⟨alphaNum⟩ | _}
⟨constructorExpr⟩ ::= ⟨nameSpace⟩ ⟨upper⟩ {⟨alphaNum⟩ | _)}
⟨nameSpace⟩ ::= {⟨alpha⟩ {⟨alphanum⟩ | _}.}

The non-terminal ⟨lower⟩ is any lowercase alphabetic letter, ⟨upper⟩ is any uppercase alphabetic letter, and ⟨alphaNum⟩ is any alphabetic letter or digit.

2.15 Conditionals

The '**if** e_B **then** e_T **else** e_F' expression has the type of e_T and e_F, which should be the same. Depending on the value of the condition e_B, the 'if-then-else' has the usual operational semantics: if e_B evaluates to *True*, then its value is that of e_T; otherwise, its value is that of e_F. In *Calculational* the **else** part is mandatory.

⟨ifExpr⟩ ::= ***if*** ⟨expr⟩ ***then*** ⟨expr⟩ ***else*** ⟨expr⟩

2.16 Unary Expressions

The unary operators supported by *Calculational* are defined as follows:

⟨negExpr⟩ ::= − ⟨factor⟩
⟨notExpr⟩ ::= (¬ | ∼) ⟨factor⟩
⟨sharpExpr⟩ ::= # ⟨factor⟩

Negation is the usual − unary minus symbol. Symbols ¬ are ∼ used for Boolean negation. Unary function symbol # is overloaded in *Calculational*. It can operate both as the Iverson's bracket operator (i.e., it returns 1 if the argument evaluates to *True* and 0 otherwise) and to denote the size of a finite collection (Section 2.19). Recall from Section 2.6 that symbol # is also used as a binary operator to denote the number of occurrences of an element in a finite bag.

2.17 Lambda Abstractions

Lambda abstractions are used in *Calculational* to represent anonymous functions, with the following syntax:

⟨lambdaExpr⟩ ::= \ ⟨patterns⟩ → ⟨expr⟩

The non-terminal ⟨patterns⟩ (see Section 2.18) is a list of patterns for pattern matching à la Haskell.

2.18 Pattern Matching

Pattern matching is supported in *Calculational* with the following syntax:

$$
\begin{aligned}
\langle \texttt{patterns} \rangle &::= \langle \texttt{pattern} \rangle \, \{ \langle \texttt{pattern} \rangle \} \\
\langle \texttt{pattern} \rangle &::= \langle \texttt{infixPattern} \rangle \\
\langle \texttt{infixPattern} \rangle &::= \langle \texttt{simpleP} \rangle \, \langle \texttt{patternOp} \rangle \, \langle \texttt{pattern} \rangle \\
\langle \texttt{simpleP} \rangle &::= \langle \texttt{patternId} \rangle \mid \langle \texttt{patternLiteral} \rangle \\
&\quad \mid \langle \texttt{constructorExpr} \rangle \, \langle \texttt{patterns} \rangle \\
&\quad \mid \mathbf{(} [\langle \texttt{pattern} \rangle \, \{ \mathbf{,} \, \langle \texttt{pattern} \rangle \}] \mathbf{)} \\
&\quad \mid \mathbf{[} [\langle \texttt{pattern} \rangle \, \{ \mathbf{,} \, \langle \texttt{pattern} \rangle \}] \mathbf{]} \\
\langle \texttt{patternOp} \rangle &::= \texttt{:} \{ \langle \texttt{symbol} \rangle \} \\
\langle \texttt{patternId} \rangle &::= \langle \texttt{identifierExpr} \rangle \, \mathbf{@} \, \langle \texttt{pattern} \rangle \\
\langle \texttt{patternLiteral} \rangle &::= \texttt{'} \langle \texttt{char} \rangle \texttt{'} \mid \texttt{"} \langle \texttt{char} \rangle \texttt{"} \mid \langle \texttt{number} \rangle
\end{aligned}
$$

The non-terminal $\langle \texttt{symbol} \rangle$ is any Haskell character symbol.

2.19 Collections

As in the Dijkstra-Scholten style, collections are at the heart of *Calculational*. The supported collections are sets, bags, and lists.

$$
\begin{aligned}
\langle \texttt{setExpr} \rangle &::= \mathbf{\{} \, \langle \texttt{collectionExpr} \rangle \, \mathbf{\}} \\
\langle \texttt{bagExpr} \rangle &::= \mathbf{\{|} \, \langle \texttt{collectionExpr} \rangle \, \mathbf{|\}} \\
\langle \texttt{listExpr} \rangle &::= \mathbf{[} \, \langle \texttt{collectionExpr} \rangle \, \mathbf{]} \\
\langle \texttt{collectionExpr} \rangle &::= \langle \texttt{comprehensionExpr} \rangle \mid \langle \texttt{extensionalExpr} \rangle \mid \epsilon
\end{aligned}
$$

Curly brackets { and } are set constructors, curly brackets with pipes {| and |} are bag constructors, and square brackets [and] are list constructors. Expression ϵ denotes the empty production rule and it is used to form empty collections. Hence, expressions

$\{\}$, $\{|\}$, $[\,]$

denote the empty set, empty bag, and empty list, respectively.

Definitions by extension. Collections can be defined by extension, both by explicit and implicit notation.

$$\begin{aligned}
\langle \texttt{extensionalExpr} \rangle &::= \langle \texttt{expr} \rangle \, [\langle \texttt{extensionalTailExpr} \rangle] \\
\langle \texttt{extensionalTailExpr} \rangle &::= \langle \texttt{impTailExpr} \rangle \\
&\quad \mid \texttt{,} \, \langle \texttt{expr} \rangle \, [\langle \texttt{impTailExpr} \rangle \mid \texttt{,} \, \langle \texttt{expTailExpr} \rangle] \\
\langle \texttt{impTailExpr} \rangle &::= \texttt{..} \, [\langle \texttt{expr} \rangle] \\
\langle \texttt{expTailExpr} \rangle &::= \langle \texttt{expr} \rangle \, [\texttt{,} \langle \texttt{expTailExpr} \rangle]
\end{aligned}$$

With explicit notation, for example, the set, the multiset, and the list of digits, respectively, are written as

$$\{0,1,2,3,4,5,6,7,8,9\},\ \{\!|0,1,2,3,4,5,6,7,8,9|\!\},\ [0,1,2,3,4,5,6,7,8,9].$$

With implicit notation the above-mentioned collections of digits are be expressed as

$$\{0\ ..\ 9\},\ \{\!|0\ ..\ 9|\!\},\ [0\ ..\ 9].$$

Implicit notation can also use an optional *step* value having the form $e_1, e_2, ..e_n$ meaning that elements are to be generated beginning with e_1, with step $e_2 - e_1$, and until e_n is reached. For example, the set of digits can be expressed by the range $0, 1, .., 9$. Also, the collections of even digits can be expressed by $\{0, 2\ ..\ 9\}$, $\{\!|0, 2\ ..\ 9|\!\}$ and $[0, 2\ ..\ 9]$. Each one of these collections contains the digits $0, 2, 4, 6, 8$. The expression e_n in *any* implicit notation is optional; it this value is not present, then the generated collection is bounded by the size of the underlying type.

Definitions by comprehension. Collection comprehension in *Calculational* is similar to the one initially proposed by Dijkstra and Scholten. Comprehension expressions consist of a binding expression lists, a range, and a body:

$$\begin{aligned}
\langle \texttt{comprehensionExpr} \rangle &::= \langle \texttt{bindExprList} \rangle \ \ (\texttt{:} \mid \mid)[\langle \texttt{expr} \rangle] \ [\texttt{:} \langle \texttt{expr} \rangle] \\
\langle \texttt{bindExprList} \rangle &::= [\langle \texttt{bindExpr} \rangle \, \{\texttt{,} \langle \texttt{bindExpr} \rangle\}] \\
\langle \texttt{bindExpr} \rangle &::= \langle \texttt{pattern} \rangle \leftarrow \langle \texttt{expr} \rangle
\end{aligned}$$

With the comprehension notation for collections, *Calculational* supports the following syntax for collections defined by comprehension:

$$\{p \leftarrow e_S : e_R : e\},\ \{\!|\ p \leftarrow e_S : e_R : e\ |\!\},\ [\ p \leftarrow e_S : e_R : e\ ].$$

The expression $p \leftarrow e_s$ is called the *generator* of the collection and p is a pattern whose variables are bound with each value v in the collection e_S. For each such v, if the expression e_R evaluated with the corresponding instance of p by v is *True*, then the value of e with that particular instance of p is accumulated

in the collection; otherwise, it is not accumulated. The range e_R can be empty meaning that the *True* value is assumed. If the body e is empty, then the resulting collection corresponds to the one represented by the pattern p in $p \leftarrow e_S$. Under this convention, the collections $\{p \leftarrow e_S : e_R\}$ and $\{p \leftarrow e_S : e_R : p_e\}$ are equivalent. The sub-index e in p_e indicates that the pattern p is automatically converted to an expression. It is important to note that the binding expression list can be empty in a comprehension. In this case, the empty collection is returned since there are no elements to consider in the comprehension.

Collections can have multiple generators separated by commas as in:

$$p_0 \leftarrow e_{S_0}, \ldots, p_n \leftarrow e_{S_n},$$

where the variables in the pattern p_i can appear in the expression e_{S_j} if $i < j$. In this case, each value v in the collection e_{S_i} is instantiated in the expression e_{S_j} as described above.

2.20 Parenthesized Expressions: Tuples and Quantifiers

Angular brackets are thoroughly used in the Dijkstra-Scholten style for quantifier expressions. However, *Calculational* uses parenthesis instead.

In *Calculational* there are two different types of parenthesized expressions, namely, (non-empty) tuples and quantifiers.

$$\begin{aligned}
\langle \texttt{parentExpr} \rangle &::= \texttt{(}\ \langle \texttt{parentsExpr} \rangle\ \texttt{)} \\
\langle \texttt{parentsExpr} \rangle &::= \langle \texttt{quantifierExpr} \rangle \mid \langle \texttt{tupleExpr} \rangle \\
\langle \texttt{tupleExpr} \rangle &::= \langle \texttt{expr} \rangle\ [\texttt{,}\ \langle \texttt{tupleExpr} \rangle]
\end{aligned}$$

Tuples are expressions separated by commas and the singleton tuple is automatically downgraded to its only operand.

The following production rules introduce the syntax for quantifier expressions:

$$\langle \texttt{quantifierExpr} \rangle ::= \langle \texttt{quantifierOp} \rangle\ \langle \texttt{comprehensionExpr} \rangle$$

The non-terminal $\langle \texttt{quantifierOp} \rangle$ corresponds to any of the operators presented in Table 1.

Quantifier expressions are written in a similar way to collection expressions, having the form:

$$(\oplus\ p \leftarrow e_S : e_R : e)$$

Operator $\oplus$ is a binary associative and commutative operator with type T (i.e., this operator has typing $T \times T \to T$) and identity element u (i.e., $(T, \oplus, u)$ is an Abelian monoid); expression e has also type T and p is a pattern. In this case, the above quantifier expression has type T.

Operationally, a quantifier expression is evaluated in the same way a collection expression. The only difference with the semantics of collections is that in a quantifier expression the values are not accumulated but are operated with $\oplus$. Similar to the situation with collections, quantifier expressions can have multiple generators. If the range of a quantifier expression is unsatisfiable by the particular values of the quantification, then the result is the identity element for the corresponding type and operator.

Table 1 presents the list of operators $\oplus$ currently supported by *Calculational*.

Operator	Text alternative	Description
$+$ Σ	sum	Summation
$*$ Π	product	Product
$\#$		Apply the # operator to the body and sums it
/\ $\wedge$	$\forall$ for all	For all quantifier
\/ $\vee$	$\exists$ exists	Exists quantifier
=== $\equiv$		Generalized equivalence
=/= $\not\equiv$		Generalized inequivalence
++	concat	Generalized concatenation
$\cup$	union	Generalized union
$\cap$	intersection	Generalized intersection
$\uparrow$	max	Maximum
$\downarrow$	min	Minimum

Table 1. Operators for quantifier expressions supported by *Calculational*.

3 Examples

This section presents examples on the use of *Calculational* as both an specification and programming language based on the ideas of Dijkstra and Scholten. These fully executable examples are written in the Haskell programming language, but they can also be implemented in any other programming language in which *Calculational* is available.

3.1 Array Summation

The following specification for the summation of a 0-based array a of N elements is taken from [1, p. 187]:

`Pre:` $0 \leq N$
`Pos:` $s = (\Sigma i \mid 0 \leq i < N : a[i])$

Using the postcondition above as a starting point, this specification can be made automatically executable in *Calculational*. The main idea is to create a summing function, say `sumA`, having a and n as parameters as shown below.

```
sumA :: (Num e) => Array Int e -> Int -> e
sumA a n = [calc| (Σ i <- [0 .. n-1] | : a!i) |]
```

The calculational fragment is embedded in Haskell by using quasiquoters, a way to embed domain specific fragments inside a Haskell program. In this case, the quasiquoter is for *Calculational* is:

```
  [calc| ... ]
```

Calculational expressions can be combined with functions available in the hosting programming environment. For instance, for the above example Haskell's **indices** function can be used to get the indices of the array a, as shown below.

```
sumB :: (Ix i,Num e) => Array i e -> e
sumB a = [calc| (Σ i <- indices a | : a!i) |]
```

3.2 Polynomial Evaluation

Another example from [1, p. 187] is the specification of a program for computing the value of a polynomial, given the value for its variable and an array a with its coefficients.

Pre: $0 \leq N$
Pos: $s = \big(\Sigma i \mid 0 \leq i < N : a[i] * x^i\big)$

In *Calculational* this evaluation can be achieved with the following executable specification that corresponds to the definition of a function `evalPoly`:

```
evalPoly :: (Ix b, Num a, Integral b) => Array b a -> a -> a
evalPoly a x = [calc| (Σ i <- indices a | : (a!i)*x^i) |]
```

3.3 Linear Search

This subsection presents several examples of linear search algorithms.

Index of an element in an array. When searching for the index of an element in an array, it is important to consider situations in which such an element is not present in the array. For theses cases, a new data type `Infty`, that includes lower and upper bounds for `Value` (i.e., the values in the array), can be introduced to handle failed searches.

```
data Infty a = NegInfty
             | Value {getValue :: a}
             | PosInfty
           deriving (Show,Eq,Ord)
```

The **`data`** clause is used in Haskell to define a new sum type, also known as disjoint type (sum types can be easily represented in object oriented languages [14]). In the case above, the **`deriving`** clause makes the defined type automatically an instance of Haskell's **`Eq`**, **`Ord`**, and **`Show`** classes. Class **`Eq`** allows the comparison of two elements of the type for equality, class **`Ord`** makes the values of the type a total order (with minimum `NegInfty` and maximum `PosInfty`), and class **`Show`** allows for the conversion of these values to an **`String`**.

The following *Calculational* code snippet computes the minimum index of an element x in an array a. When the search succeeds, then the minimum index is returned; otherwise, `PosInfty` is returned.

```
indexOf :: (Ix b, Eq a) => Array b a -> a -> Infty b
indexOf a x = [calc| (↓ i <- indices a | (a!i)==x : Value i) |]
```

Index of an element that satisfies a predicate. A generalization of the linear search in Subsection 3.3 to find the index of an element x that satisfies a given predicate p can be done as follows:

```
indexOfBy :: (Ix b) => Array b a -> (a -> Bool) -> Infty b
indexOfBy a p = [calc| (↓ i <- indices a | p (a!i) : Value i) |]
```

By instantiating the property p in `indexOfBy` with the equality predicate, then an alternative linear search solution to the problem in Subsection 3.3 can be obtained:

```
indexOf' :: (Ix b, Eq a) => Array b a -> a -> Infty b
indexOf' a x = indexOfBy a (== x)
```

Variations of these definitions can be obtained for sequences in general. All that is needed is that elements in such collections are paired with indices (e.g., via Haskell's **`zip`** function).

```
indexOfByList :: [a] -> (a -> Bool) -> Infty Int
indexOfByList xs p =
    [calc| (↓ (x,i) <- zip xs [0 .. ] | p x : Value i) |]

indexOfList :: (Eq a) => [a] -> a -> Infty Int
indexOfList xs x = indexOfByList xs (== x)
```

Deciding if an array contains an element satisfying a predicate. Typically, some algorithms require a container a such as an array to contain at least one element satisfying a given property p. These assertions can be easily specified as a *Calculational* quantifier expression as follows:

```
isElemBy :: (Ix b) => Array b a -> (a -> Bool) -> Bool
isElemBy a p = [calc| (∃ i <- indices a |: p (a ! i) ) |]
```

If the interest is in determining if an element x is an element of an array a, then the following definition can be given:

```
isElem :: (Ix b,Eq a) => Array b a -> a -> Bool
isElem a x = isElemBy a (== x)
```

As noted before, these specifications can be adapted without much effort to lists and other sequences.

Deciding if all elements in an array are nonzero. Since *Calculational* has support for universal quantification, then deciding if all elements in an array are nonzero can be specified easily. First, a general definition can be given in which there is a parametric predicate. Then, this definition can be used by instantiating such a predicate with the "nonzero" predicate. Both definitions are shown below:

```
allBy :: (Ix b) => Array b a -> (a -> Bool) -> Bool
allBy a p = [calc| (∀ i <- indices a |: p (a ! i) ) |]

nonZero :: (Ix b,Num a,Eq a) => Array b a -> Bool
nonZero a = allBy a (/= 0)
```

3.4 Dutch National Flag

The "Dutch National Flag" problem was initially proposed by E. W. Dijkstra [7]. It consists of ordering a sequence of values representing the colors of the Dutch national flag: blue, white, and red.

Consider the data type `FlagColor` representing the three colors mentioned above:

```
data FlagColor = Blue
               | White
               | Red
               deriving (Eq,Ord, Show)
```

The following Haskell algorithm embeds *Calculational* code to solve the Dutch national flag problem. The main idea is to convert the input sequence into a bag and then, by using *Calculational* support for counting elements in a bag, construct an ordered version of the input sequence.

```
dutchFlag :: [FlagColor] -> [FlagColor]
dutchFlag xs = rep Blue ++ rep White ++ rep Red
    where ms = [calc| {| x <- xs | : x |} |]
          rep c = [calc| replicate (c # ms) c |]
```

An alternative but simpler solution can also be found by using *Calculational* and the predefined order imposed on the colors in `FlagColor` as follows:

```
dutchFlag :: [FlagColor] -> [FlagColor]
dutchFlag xs = toList [calc| {| x <- xs | : x |} |]
```

3.5 Set Partitions

The following formula is proposed in [9] to characterize a partition S of a set T:

$$(\forall\, u, v \mid u \in S \wedge v \in S \wedge u \neq v \;:\; u \cap v = \emptyset) \wedge (\cup\, u \mid u \in S \;:\; u) = T.$$

The first conjunct in the partition formula indicates that the sets in the partition S are pair-wise disjoint. The second conjunct indicates that the union of all sets in the partition S is the entire set T.

The following code snippet shows a Haskell program that uses *Calculational* to automatically verify the set partition formula.

```
isPartition :: (Ord a) => Set (Set a) -> Set a -> Bool
isPartition s t = [calc|
    (∀ u <- s, v <- s | u /= v : u ∩ v == ∅)
                          ∧ (∪ u <- s | : u) == t
|]
```

3.6 Power Set of a Finite Set

The power set $\mathcal{P}(S)$ of a set S is the set of all subsets of S. The power set can be constructed in many different ways. This section presets three alternative recursive constructions that can be easily derived from definitions with the help of *Calculational*.

The first approach is to generate the power set following the recursive characterization found in [23]. The base case corresponds to the power set of the empty set (denoted in *Calculational* as `[calc| ∅ |]`. The inductive case corresponds to the union of the set S with the collection of *all* power sets of sets obtained from S by removing one element. Note that the union of all these sets is easily specified with quantifier expressions in *Calculational*.

```
powerSet :: Ord a => Set a -> Set (Set a)
powerSet s =
    if s == [calc| ∅ |]
    then [calc| {∅} |]
    else [calc| {s} ∪ (⋃ e <- s | : powerSet (s \ {e})) |]
```

The second approach is to inductively construct the power set by layers. The idea is to begin with the only subset of T having no elements, i.e., the empty set. Then, recursively, construct the subsets of T having size $k + 1$ based on the already created sets of size k. The resulting executable specification, shown below, uses *Calculational* collections having multiple generators that ultimately help in reducing the size of the specification.

```
powerSet :: Ord a => Set a -> Set (Set a)
powerSet s = [calc| (⋃ i <- [ 0 .. n ] | : t i ) |]
   where t 0 = [calc| { ∅ } |]
         t k = [calc| { u<-t(k-1),x<-s | x ∉ u : {x} ∪ u } |]
         n = [calc| # s |]
```

Both of `powerSet` versions presented so far have the defect of generating the subsets of T more than once. The goal is to finally eliminate superfluous generation of subsets by using memoization techniques for functional programming

such as the ones in [16]. The following code snippet presents an improved variant of the second `powerSet` definition in which each subset of T is constructed exactly once.

```
powerSet :: Ord a => Set a -> Set (Set a)
powerSet s = [calc| (⋃ i <- [ 0 .. n ] | : mem ! i ) |]
   where mem = listArray (0,n) [ t i | i <- [0..n] ]
         t 0 = [calc| { ∅ } |]
         t k = [calc| { u<-mem!(k-1),x<-s | x∉u : {x} ∪ u } |]
         n = [calc| # s |]
```

In the latter version of `powerSet` the `mem` construct is used as a memoization mechanism. This results in a bookkeeping that helps in avoiding unnecessary recursion thus generating each subset of T exactly once. In terms of calculations, this last version is the most efficient among the three versions of function `powerSet` presented so far.

3.7 Common Letters

In this example the interest is in finding the letters common to every word in a collection of words. An straightforward solution is to take a generalized union over the set of words for each word in the given collection. This can be specified in *Calculational* as the following function `commonLetters`:

```
commonLetters :: Foldable t => t String -> Universe Set Char
commonLetters s =  [calc|
     (∩ u <- s | : Container { x <- u | : x } )
|]
```

Function `commonLetters` depends on data type `Universe`, which is needed for handling an empty set of words as input and it is defined in the same spirit of data type `Infty` in Subsection 3.3.

```
data Universe m a = Container { getContainer :: m a }
                  | Universe
                  deriving (Eq, Ord, Read, Show)
```

The `Universe` data type has two arguments, namely, `m` and `a`. The `m` argument is a variable that denotes the type of the collection and the `a` argument is the type of its elements. The type expression `m a` denotes the type of a collection of elements of type `a`. Because `m` is a type variable, it could be instantiated in any collection or container type such as **`List`**, `Set` or `Bag`.

In this case of function `commonLetters`, constant `Universe` is used as the neutral element of the intersection operator. In particular, in the case in which `u <- s`

yields the empty collection of words, the quantifier evaluates to the constant `Universe`.

3.8 Degree of Vertices in a Graph

Given a directed graph $G = (V, E)$:

- the *outdegree* of a vertex $u \in V$ is the number of edges $(u', v') \in E$ such that $u' = u$.
- the *indegree* of a vertex $v \in V$ is the number of edges $(u', v') \in E$ such that $v' = v$;

Consider the following definitions of functions `outdegree` and `indegree` in *Calculational*, where both definitions use bags (Subsection 2.19) and the **#** operator for bags (Subsection 2.6).

```
outdegree :: (Foldable c, Ord a) => c (a, a) -> a -> Int
outdegree edges u = [calc| # { (u',v') <- edges | u == u' } |]

indegree :: (Foldable c, Ord a) => c (a, a) -> a -> Int
indegree edges v = [calc| # { (u',v') <- edges | v == v' } |]
```

An alternative solution is to associate to each vertex in V the collection of vertices that connect to it and connect from it in E, respectively. Then, each of one of the problems at hand can be solved by counting the corresponding set of vertices associated to it.

```
outvertices :: (Foldable c, Ord a) => c (a, a) -> MultiSet a
outvertices edges = [calc| {| (u,_) <- edges |: u  |} |]

invertices :: (Foldable c, Ord a) => c (a, a) -> MultiSet a
invertices edges = [calc| {| (_,v) <- edges |: v  |} |]
```

Note that wildcard _ is a pattern that matches with any term but does not do result in any binding.

The following are the alternative definitions for functions `outdegree` and `indegree` that use functions `outvertices` and `invertices`, respectively.

```
outdegree :: (Foldable c, Ord a) => c (a, a) -> a -> Int
outdegree edges u = [calc| u # outvertices edges |]

indegree :: (Foldable c, Ord a) => c (a, a) -> a -> Int
indegree edges v = [calc| v # invertices edges |]
```

Finally, if the given graph is undirected, it does not make sense to talk about outgoing or incoming edges for a vertex. Instead, it is possible to compute the *degree* of a vertex v, i.e., the number of edges $\{u', v'\} \in E$ such that $v = u'$ or $v = v'$. Function `degree` below compute the degree of a vertex in a graph.

```
degrees :: (Foldable c, Ord a) => c (Set a) -> MultiSet a
degrees edges = [calc| {| e <- edges, u <- e |: u  |} |]

degree :: (Foldable c, Ord a) => c (Set a) -> a -> Int
degree edges u = [calc| u # degrees edges |]
```

3.9 Transitive Closure

A binary relation R over a set A (i.e., $R \subseteq A \times A$) is *transitive* iff for any $x, y, z \in A$ if $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$. Such a definition has a straightforward translation in *Calculational* as follows:

```
isTransitive :: (Ord a) => Set (a, a) -> Bool
isTransitive r = [calc|
    (∀ (x,y) <- r, (y',z) <- r | y == y' : (x,z) ∈ r)
|]
```

Given a binary relation $R \subseteq A \times A$ it is natural to consider its *transitive closure* R^+ defined to be the least relation R^+ such that $R \subseteq R^+$ and R^+ is transitive. Traditionally, the transitive closure R^+ of a relation R is defined mathematically as follows[24]:

$$\left(\cup\, i \in \{1, 2, 3, \dots\} \mid :\ R^i\right)$$

Note that if R is transitive, then $R = R^+$. The goal is to specify, with the help of *Calculational*, the transitive closure of a given binary (homogeneous) relation R.

The following are the definitions of *relational composition* the *power of a relation* found in [9]:

$$R \circ S = \{(x, y) \in R, (y, z) \in S \ \mid : (x, z)\}$$

$$\begin{aligned} R^1 &= R \\ R^i &= R \circ R^{i-1} \qquad \text{, for } i > 1. \end{aligned}$$

With the help of the infix notation, these definitions can be expressed in *Calculational* as follows:

```
o :: (Ord a,Ord b,Ord c) => Set (a, b) -> Set (b,c) -> Set (a,c)
r `o` s = [calc| { (x,y) <- r,(y',z) <- s | y == y' : (x,z) } |]

power :: (Ord a,Ord n,Num n) => Set (a, a) -> n -> Set (a,a)
power r 1 = r
power r i | i > 1 = r `o` power r (i - 1)
```

With the help of *Calculational*, the transitive closure for a given *finite* relation over a *finite* set can be computed by the function `transitiveClosure` below with tail recursion and generalized union.

```
transitiveClosure :: (Ord a) => Set (a, a) -> Set (a,a)
transitiveClosure rel = transClos rel rel
    where transClos rs u = if isTransitive u
                           then u
                           else transClos rs' [calc| rs' ∪ u |]
                where rs' = (rel `o` rs)
```

Note that this specification does not terminate if the input set a is infinite. The variable `rs'` is used to avoid multiple computations in the expression

`transClos rs' [calc| rs' ∪ u |]`.

4 Database Examples

This section presents examples on how to query and process data in a relational setting. This is particularly useful for application of formal methods for databases where data structure is represented by relations. As a matter of fact and as pointed in [9], databases are an important application of the theory of relations.

Consider the data types `Person` and `Employee` that represent, respectively, a person and an employee. A person has an identifier (attribute `idPerson`), a name (attribute `name`), an age (attribute `age`), and a gender (attribute `gender`). An employee has a identifier (attribute `idEmploye`), it is a person (attribute `idPersonE`), and it has a salary (attribute `salary`).

```
data Person = Person { idPerson :: IdPerson,
                       name :: String,
                       age :: Integer,
                       gender :: Gender }
              deriving (Show,Eq,Ord)

data Employee = Employee { idEmployee :: IdEmployee,
```

```
                        idPersonE :: IdPerson,
                        salary :: Integer }
            deriving (Show,Eq,Ord)
```

In the rest of the section, it is assumed that `persons` and `employees` are variables denoting collections of, respectively, people and employees.

Joins. Joins are very common queries in relational databases. As an example, consider the query to find the people having a salary within a given range:

```
salaryBetween :: Integer -> Integer -> Set (String,Integer)
salaryBetween a b = [calc|
  { p <- persons, e <- employees |
    personId p == personIdE e /\ a <= salary e <= b :
    (name p,salary e) } |]
```

The join condition is given by the constraint `personId p == personIdE e` requiring that the person and employee must be the same.

This query has a straightforward translation into SQL as follows:

```
SELECT p.name, e.salary
    FROM persons as p, employees as e
    WHERE p.personId = e.personIdE AND a <= e.salary AND e.
        salary <= b
```

Aggregate operators. A relational aggregation operator in relational databases can be seen mathematically as a quantifier. Consider the following *Calculational* expression that computes the sum of the payroll:

```
payrollValue :: Integer
payrollValue =  [calc| (Σ e <- employees : : salary e ) |]
```

Function `payrollValue` can be automatically translated into the following SQL expression, where `sum` is assumed to be an aggregate function calculating to calculate the sum of a given collection:

```
SELECT sum(e.salary)
    FROM employees as e
```

It is also common in relational databases to arrange similar data into groups and then aggregate each group's values. The following example shows how to

use sets and a quantifier expressions in *Calculational* to obtain the frequency of each salary:

```
salaryFrequence :: Set (Integer,Integer)
salaryFrequence = [calc|
    { e <- employees | :
      (salary e,(# e' <- employees |: salary e' == salary e)) }
|]
```

An alternative version can be obtained by using bags as follows:

```
salaryFrequence :: MultiSet Integer
salaryFrequence = [calc| {| e <- employees : : salary e |} |]
```

A resulting SQL query is the following:

```
SELECT e.salary, count(*)
    FROM employees as e
    GROUP BY e.salary
```

As a final example, consider the search for the maximum salary in the payroll. In *Calculational*, this query can be specified as follows:

```
maxSalaryByAge :: Set (Integer,Infty Integer)
maxSalaryByAge = [calc|
   { p <- persons | : (age p,(↑ e <- employees |
          personIdE e == personId p : Value (salary e))) } |]
```

An SQL equivalent expressions is the following:

```
SELECT p.age,max(e.salary)
    FROM person as p
    LEFT JOIN employees as e
    ON p.personId = e.personIdE
    GROUP BY p.age
```

5 Haskell Implementation Overview

The version of *Calculational* presented in this document has been implemented in Haskell using the template meta-programming facilities in [20]. This section presents details on the implementation of *Calculational*'s quantifiers and

collections, because they present the highest level of implementation complexity.

The implementation of quantifiers and collections in *Calculational* uses the `Monoid` and `Foldable` classes available in Haskell's prelude.

The `Monoid` class is for types with an associative binary operator `mappend` with neutral element `mempty`.

```
class Monoid a where
    mempty :: a
    mappend :: a -> a -> a
```

For example, data type `List` is an instance of `Monoid` with the neutral element being the empty list `[]` and list concatenation as the binary operator.

```
instance Monoid [a] where
        mempty  = []
        mappend = (++)
```

In Haskell it is allowed for a class to be an instance of at most one class. This deficiency can be solved by using the `newtype` declaration that wraps a type creating a new type that is taken into account at compiling time and still does not yield significant runtime overhead when the wraps are ignored. By following this approach, data type `Num` can be an instance of the `Monoide` class and, at the same time, have the choice between the sum, the product, and any other associative binary operation. For example, the `sum` and `product` monoids are declared in the `Data.Monoid` module as follows:

```
-- | Monoid under addition.
newtype Sum a = Sum { getSum :: a }
        deriving (...)

instance Num a => Monoid (Sum a) where
        mempty = Sum 0
        Sum x `mappend` Sum y = Sum (x + y)

-- | Monoid under multiplication.
newtype Product a = Product { getProduct :: a }
        deriving (...)

instance Num a => Monoid (Product a) where
        mempty = Product 1
        Product x `mappend` Product y = Product (x * y)
```

These definitions indicate that the type `Sum a` and the type `Product a` are instances of the class `Monoid` whenever the constrain `Num a` is satisfied. This

constraint implies that the type a, that is wrapped, must be an instance of `Num`. For `Sum a` to be an instance of `Monoid`, its neutral element `mempty` is number zero wrapped with type constructor `Sum`; the binary operation `mappend` unwraps the arguments and wraps the result of the sum of the two arguments. Similar considerations apply for the type `Product` so it can be an instance of `Monoid`.

The other class that is used is `Foldable`. In particular, function `foldMap` is thoroughly used in *Calculational*:

```
class Foldable t where
    ...
    -- | Map each element of the structure to a monoid,
    -- and combine the results.
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty
    ...
    -- | Right-associative fold of a structure.
    foldr :: (a -> b -> b) -> b -> t a -> b
    ...
```

For the purpose of showing how quantifiers and collections are implemented in *Calculational*, it is important to understand how function `foldr` is defined. As example, consider the specific implementation of **`foldr`** for lists:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b []     = b
foldr f b (a:as) = f a (foldr f b as)
```

Expressions `[]` and `(a:as)` denote, respectively, the empty list and the list with head `a` and tail `as`.

In abstract, `foldr` can be used to transform a collection cs of elements $x_1, x_2 \ldots, x_n$ of type a into a type b, as follows:

$$\begin{aligned} &\mathit{foldr}\ (\otimes)\ b\ \emptyset = b \\ &\mathit{foldr}\ (\otimes)\ b\ cs = x_1 \otimes (x_2 \otimes (\cdots \otimes (x_n \otimes b) \cdots)) \end{aligned}$$

Consider the following definition of function `foldMap` that uses `foldr`, `foldMap`, instantiates b with the neutral element, instantiates $\otimes$ with `mappend . f`, instantiates `mappend` with $\oplus$, and instantiates `mempty` by u (note that `mappend` is associative and u is its neutral element):

$$\begin{aligned} &\mathit{foldMap}\ f\ \emptyset = u \\ &\mathit{foldMap}\ f\ cs = f\ x_0 \oplus f\ x_1 \oplus \cdots \oplus f\ x_n \end{aligned}$$

For the implementation of *Calculational*, the following variation of function `foldMap` is used:

```
bind :: (Foldable t, Monoid m) =>  t a -> (a -> m) -> m
bind = flip foldMap
```

5.1 Quantifiers

A quantifier of the form

```
(⊕ x1 <- s1,x2 <- s2,..., xn x<- sn | r : e)
```

is codified following ideas similar to the ones in [22,11,13], as follows:

```
s1 `bind` \p1 ->
s2 `bind` \p2 ->
...
sn `bind` \pn ->
if r then e else mempty
```

where s_i are collections and p_i are patterns, $1 \leq i \leq n$. Operationally, the pattern p_i instantiate his variables with each value v in the collection s_i and the instantiated variables are used in the range `r` and in the expression `e`. When the range `r` with the pattern variables instantiated is not satisfied, the neutral element is computed (*False* case of the **`if`** conditional); otherwise, the `e` expression with the pattern variables instantiated is computed (*True* case of conditional).

To show how translation works, consider the case of one collection s_1 with elements $u_1, u_2, \ldots, u_k$, and for simplicity consider pattern p_1 as variable x:

```
s1 `bind` \x -> if r then e else mempty
```

the preceding expression is equivalent to

```
   if r[x := u1] then e[x := u1] else mempty
⊕ if r[x := u2] then e[x := u2] else mempty
...
⊕ if r[x := uk] then e[x := uk] else mempty
```

The expression $e[x := u_i]$ denotes the substitution of free occurrences of x in e by u_i.

For the case of two collections s_1 with elements $u_1, u_2, \ldots, u_k$ and s_2 with elements $v_1, v_2, \ldots, v_l$, consider patterns p_1 and p_2 as variables x and y, then:

```
s1 `bind` \x ->
s2 `bind` \y ->
if r then e else mempty
```

is expanded to:

```
   if r[y := v1][x := u1] then e[y := v1][x := u1] else mempty
⊕  if r[y := v2][x := u1] then e[y := v2][x := u1] else mempty
...
⊕  if r[y := vl][x := u1] then e[y := vl][x := u1] else mempty
⊕  if r[y := v1][x := u2] then e[y := v1][x := u2] else mempty
⊕  if r[y := v2][x := u2] then e[y := v2][x := u2] else mempty
...
⊕  if r[y := v1][x := uk] then e[y := v1][x := uk] else mempty
⊕  if r[y := v2][x := uk] then e[y := v2][x := uk] else mempty
...
⊕  if r[y := vl][x := uk] then e[y := vl][x := uk] else mempty
```

Because some monoids require wrapped values and return an unwrapped result, a function `w` is used with expression `e`, then resulting expression is converted to `w e` and a function `uw` with the result. For the expressions that we must wrap and unwrap we use the constructor and pattern matching; for expression that does not have to be wrapped we use the identity function for `w` and `uw`.

For example, the expanded translation for collection s with elements $u_1, u_2, \ldots, u_k$, the pattern p as variable x, and the `Sum` monoid type defined before, with the `mappend` function denoted by $\oplus$, is:

```
getSum(if r[x := u1] then Sum(e[x := u1]) else Sum 0
⊕  if r[x := u2] then Sum(e[x := u2]) else Sum 0
...
⊕  if r[x := uk] then Sum(e[x := uk]) else Sum 0)
```

5.2 Collections

The implementation of collections uses a similar translation to that of quantifiers 5.1 because lists, sets, and bags are monoids.

The main implementation difference between these three types of containers is that they have different union and intersection. For this purpose, classes `UnionClass` and `IntersectionClass` have been defined with a type of family constraints [5,4].

```
class UnionClass m a where
    type UC m a :: Constraint
    munion :: UC m a => m a -> m a -> m a

class IntersectionClass m a where
    type IC m a :: Constraint
    mintersection :: (IC m a) => m a -> m a -> m a
```

Types `UC m a` and `IC m a` correspond to specific constraints over the types `m` and `a`. Note that functions `munion` and `mintersection` are overloaded for union and intersection, each with its own constraints.

The following code presents the union definition for lists, sets, and bags. The definition for the intersection operation can be obtained in a similar way.

```
instance UnionClass [] a where
      type UC [] a = Eq a
      munion = L.union

instance UnionClass Set.Set a where
      type UC Set.Set a = Ord a
      munion = Set.union

instance UnionClass MultiSet.MultiSet a where
      type UC MultiSet.MultiSet a = Ord a
      munion = MultiSet.union
```

Since there are at least two instances for class `Monoid`, new data types `Union` and `Intersection` are introduced. The code for `Union` is shown below.

```
newtype Union m = Union { getUnion :: m }
                deriving (Eq, Ord, Read, Show, Bounded)

instance (UC m a,UnionClass m a,Monoid (m a))
                  => Monoid (Union (m a)) where
      mempty = Union mempty
      s1 `mappend` s2 = Union (getUnion s1 `munion` getUnion s2)
```

Finally, the `Universe` extension (which adds an upper bound to containers) is made an instance of `UnionClass`. Note that constraint `UC` has the specific constraints of each collection data type.

```
instance (UC m a,UnionClass m a) => UnionClass (Universe m) a
    where
      type UC (Universe m) a = UC m a
```

```
Container xs `munion` Container ys = Container (xs `munion
    ` ys)
_ `munion` _ = Universe
```

6 Concluding Remarks

This manuscript presented *Calculational*, an executable specification language for a broad class of calculational expressions in the style of Dijkstra & Scholten. The *Calculational* tool offers support for executable specifications, including quantifier expressions such as summation and universal/existential quantification, and data types such as lists, sets, and bags. An overview of the *Calculational* language has been presented and several examples were given illustrating the language's main features. Also, a case study on the processing and querying of relational data with the help of *Calculational* was included in the paper. Finally, an overview of the current implementation of *Calculational* in the Haskell programming language was presented, with specific details on the implementation of collections and generalized operations.

As future work, it seems promising to extend *Calculational* with support for symbolic expressions so that calculational expressions can contain variables. Currently, *Calculational* offers support for ground expressions. This new feature can be useful specially for the derivation of algorithms in the style of Dijkstra & Scholten. This goal suggest to explore alternative implementations of *Calculational*'s language in other declarative programming languages such as Maude [6].

References

1. R. Backhouse. *Program Construction: Calculating Implementations from Specifications*. John Wiley & Sons, Mar. 2003.
2. L. Bijlsma and R. Nederpelt. Dijkstra-Scholten predicate calculus: Concepts and misconceptions. *Acta Informatica*, 35(12):1007–1036, 1998.
3. J. Bohórquez. Intuitionistic logic according to Dijkstra's calculus of equational deduction. *Notre Dame Journal of Formal Logic*, 49(4):361–384, 2008.
4. M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. *SIGPLAN Not.*, 40(9):241–253, Sept. 2005.
5. M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. *SIGPLAN Not.*, 40(1):1–13, Jan. 2005.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
7. E. Dijkstra. *A Discipline of Programming*. Prentice-Hall series in automatic computation. Prentice-Hall, 1976.

8. E. Dijkstra and C. Scholten. *Predicate calculus and program semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.
9. D. Gries and F. B. Schneider. *A logical approach to discrete math*. Texts and Monographs in Computer Science. Springer, 1993.
10. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
11. G. Hutton. *Programming in Haskell*. Cambridge University Press, Jan. 2007.
12. V. Lifschitz. On calculational proofs. *Annals of Pure and Applied Logic*, 113(1-3):207–224, 2001.
13. M. Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011.
14. M. Odersky and P. Wadler. Pizza into java: Translating theory into practice. In P. Lee, F. Henglein, and N. D. Jones, editors, *POPL*, pages 146–159. ACM Press, 1997.
15. S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.
16. F. Rabhi and G. Lapalme. *Algorithms; A Functional Programming Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.
17. C. Rocha. The formal system of Dijkstra and Scholten. In *Logic, Rewriting, and Concurrency*, Lecture Notes in Computer Science, 2015 (to appear).
18. C. Rocha and J. Meseguer. A rewriting decision procedure for Dijkstra-Scholten's syllogistic logic with complements. *Revista Colombiana de Computación*, 8(2), 2007.
19. C. Rocha and J. Meseguer. Theorem proving modulo based on Boolean equational procedures. In R. Berghammer, B. Möller, and G. Struth, editors, *Relations and Kleene Algebra in Computer Science, 10th International Conference on Relational Methods in Computer Science, and 5th International Conference on Applications of Kleene Algebra, RelMiCS/AKA 2008, Frauenwörth, Germany, April 7-11, 2008. Proceedings*, pages 337–351, 2008.
20. T. Sheard and S. P. Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, Dec. 2002.
21. G. Tourlakis. On the soundness and completeness of equational predicate logics. *Journal of Logic and Computation*, 11(4):623–653, 2001.
22. P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM.
23. Wikipedia. Power set — Wikipedia, the free encyclopedia, 2015. [Online; accessed 05-June-2015].
24. Wikipedia. Transitive closure — Wikipedia, the free encyclopedia, 2015. [Online; accessed 05-June-2015].