RZ 3828 (# Z1208-002) 08/10/2012
Computer Science 23 pages

Research Report

Language Definition for a Notation of Computational Problems

R. Jongerius‡, P. Stanley-Marbell*

‡IBM Netherlands
1006 CE Amsterdam
Netherlands

*IBM Research – Zurich
8803 Rüschlikon
Switzerland

LIMITED DISTRIBUTION NOTICE

This report will be distributed outside of IBM up to one year after the IBM publication date.
Some reports are available at http://domino.watson.ibm.com/library/Cyberdig.nsf/home.

Research
Almaden • Austin • Brazil • Cambridge • China • Haifa • India • Tokyo • Watson • Zurich

Language Definition for a Notation of Computational Problems

Rik Jongerius
IBM Netherlands*
r.jongerius@nl.ibm.com

Phillip Stanley-Marbell
IBM Research—Zürich
pst@zurich.ibm.com

August 10, 2012

Contents

*Work performed while at IBM Research—Zürich

1 Introduction

Computer applications usually implement several *algorithms* to perform the task intended by a programmer. However, these algorithms only describe methods for solving *computational problems* (CPs). Usually, these CPs can be solved by a variety of algorithms, and each algorithm can be implemented in a variety of ways [1]. As an example, consider the computational problem of sorting integers; algorithms to solve this problem include MergeSort, QuickSort, and others. A programmer is left with the task of selecting the best-performing algorithm and implementation for a given hardware platform and input data size. However, not all design parameters might be known or even fixed at design time, making this a non-trivial, and possibly even impossible task.

The problem of *algorithmic choice* can be addressed by moving the task of selecting algorithms and implementations from the programmer to the runtime system. A runtime system can base the decision of which algorithm to execute on current system parameters and past performance measurements of the algorithms. Such a runtime system, however, needs a method to identify candidates for replacement among the algorithms and implementations solving the same CP.

This document describes NCP, a notation to capture the semantic properties of computational problems, independent of algorithms. A CP is a 3-tuple, as defined in Definition 1. The set S_D describes the possible inputs, or *domain* of the problem, and S_R the possible outputs, or *range*. The relation $\mathcal{R}$ defines which outputs are valid given the input of the problem.

Definition 1. *Computational Problem (CP). A computational problem, CP(S_D, S_R, $\mathcal{R}$) is a 3-tuple representing an input set S_D and output set S_R, which are related by the relation $\mathcal{R}$.* ♢

The main intention of the notation is to enable algorithmic choice after application compilation. However, the potential usage of the notation is not restricted to this single objective. It is envisioned that the notation can be used, among other things, for:

Algorithmic choice. Code sections in executable binaries can be annotated with a description of the CPs described in NCP. A runtime system may then use the annotated CP descriptions of two independent sections of code (which might be in other libraries or binaries) to verify if both sections solve the same problem, and, if so, possibly replace one by the other.

Documentation. The notation can be used to give a human-understandable description of the *purpose* of a piece of (possibly highly optimized) source code. It may be easier to understand a description of the semantics written in NCP than to extract this from an algorithmic implementation.

Analysis. The NCP description permits interesting new CP-level analyses. Using *descriptive complexity theory*, it is, for example, possible to determine the computational complexity of a CP based on the logic operators used in the NCP description. Another potential analysis may be to make statements about inherent parallelism in a given computational problem.

Verification. The description of a CP in NCP captures the semantics of the computation an algorithmic implementation performs. This may allow one to verify correctness of an algorithmic implementation based on the CP description.

Problem solving. A CP may be solved using search heuristics (analogous to those used by SAT and SMT solvers). The NCP notation can be compiled to Sal [2], permitting Svm to be used to solve the CP description. A potential future method would be to automatically synthesize the semantic description into a software algorithm or hardware accelerator.

In the remainder of this document, the term *problem* is strictly used for problem definitions written in the notation described in this reference manual. The term *application* is used for host applications written in any other language, implementing algorithms which are annotated, with a *problem*, for algorithmic replacement.

2 NCP Language Overview

NCP is a declarative language for describing computational problems, which is intended to enable algorithmic choice. It is a language with operators capturing at least a second-order logic language. Based on the problem descriptions, a decision can be made to determine if two machine-code implementations of a problem can be replaced by each other without changing the behavior of the application. The basic unit considered for replacement on the machine-code level is between a function's *call* instruction and its *return* instruction. Replacement may then be performed by changing the jump of the call instruction to the address of the replacement routine.

The programmer captures the semantics of a CP in the language. The problem definition may be developed together with the algorithm used in the application, or may be developed after the fact. The application can be programmed in any high-level programming language, for example, C. Both the problem definition and the algorithm have the same inputs and outputs. However, the algorithm describes how the output can be calculated step by step from the input, while the problem only describes the relation between input and output.

2.1 Low-level representation

For algorithmic replacement, comparing the computational semantics (the relation) of two implementations is not sufficient to guarantee correct operation of the application after replacement. The system has to ensure that two implementations not only solve the same problem, but also expect the same data types, sizes, and memory locations. As such, a low-level representation incorporating this additional information is needed. For example, a *mapping table*, which maps variables to physical locations, could be contained in the low-level representation. The current implementation of NCP, however, does not yet provide such a mapping table.

2.2 Relation to Sal/Svm

An effort is being made to compile NCP into the Set assembly language (Sal) [2], a low-level language for representing CPs. The current NCP compiler can compile a subset of the language to Sal (a few examples of NCP problems compiled to Sal can be found in Appendix E). NCP was designed to provide a higher-level interface to the concepts embodied in Sal; the relation between NCP and Sal can be seen as anologous to that between a high-level language, such as C, and a machine language, such as MIPS assembler.

There are two reasons to compile NCP problems into Sal problems. The Set virtual machine (Svm) is Sal's runtime system, providing a means to execute NCP problems by compiling them down to Sal. Compiling into Sal thus allows verification of the correctness of a CP description by verifying if the generated output is as expected. Second, if a correct CP description is available, this also allows verification of the associated algorithmic implementations, as it allows evaluating whether a data input-output pair satisfies the relation.

While Sal is capable of capturing the semantics of computational problems, NCP can also capture the data structures of input and output of CPs as required for algorithmic choice.

2.3 Variables

Variables in an NCP problem are either *physical* or *virtual* (Definitions 2 and 3) and can be either *free* or *bound* (Definitions 4 and 5, which are identical to the work of Abelson et al. [3]).

Definition 2. *Physical variables. A variable* `v` *is a physical variable if it appears in an NCP problem and has a counterpart in the actual machine-code implementation of the computational problem. The mapping table contains an entry mapping the variable to the physical representation on the hardware platform.* ♢

Definition 3. *Virtual variables. A variable* `v` *is a virtual variable if it appears in an NCP problem, but does not have a counterpart in an actual machine-code implementation. The mapping table contains no entry for the variable.* ♢

Definition 4. *Bound variables. A variable is bound in an expression if consistent renaming of the variable throughout the expression does not change the meaning of the expression.* ♢

Definition 5. *Free variables. A variable which is not bound in an expression is a free variable.* ♢

---

3 NCP Language Definition

The NCP language is used to describe the semantic properties of a computational problem (CP). It captures the relation between the domain and range of the CP. The notation is strongly-typed, has several constructs such as predicates and function expanding, to simplify CP definitions. The complete grammar of the notation in EBNF [4] can be found in Appendix A.

3.1 Lexical elements

A problem consists of a sequence of characters. Groups of characters form tokens according to the language specification. Tokens are separated by whitespace, or any of the non-alphanumeric tokens. Whitespace consist of the characters '\␣', '\n' '\r', and '\t'. The non-alphanumeric tokens are:

```
==   !=   <   >   <=   >=   >=<  +   -   *   /   %   ^   (   )   {
}    [    ]   ,   :    ::   ;    ::= =   _   '   "   #
```

The reserved keywords are:

```
include     define      typedef     domain      range
relation    struct      int         nat         real
char        bool        or          and         not
forall      exists      sum         prod        min
max         from        to          with
```

A few keywords are reserved for future functionality and, as such, can not be used for identifiers:

```
string
```

Comments start with `//` and end with a newline character '\n'. Constant values are tokenized by the lexer and their syntax diagrams are shown in Figure 1.

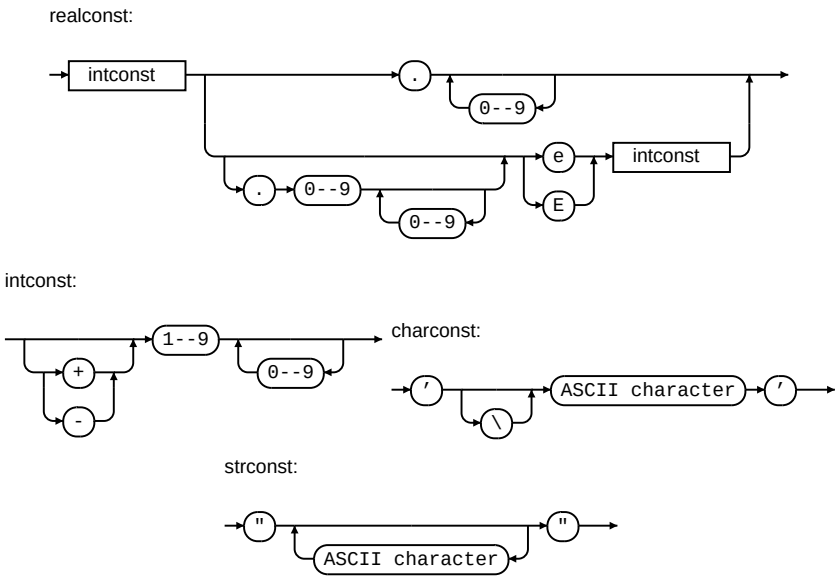

Figure 1: Syntax diagrams for signed real, signed integer, character, and string constants.

3.2 Problem structure

The syntax diagram in Figure 2 and code in Listing 1 show the basic structure of a computational problem in NCP. The first part of the problem in the listing, before the `typedef` keyword, is called the *header*. If needed, the header can include Boolean predicate, arithmetic function, and structure prototypes. The header can contain C-style `#include` and `#define` directives, which are processed by a preprocessor before actual parsing starts.

An `#include` statement can be used to include library interfaces. Libraries implement arithmetic functions or Boolean predicates, and, after inclusion of their interfaces, these functions and predicates are available to use in an NCP problem. Currently, only the `math` library is available (see Appendix B).

Listing 1: Basic structure of an NCP problem.

```
//optional header
#include "lib"

typedef ::
 //Optional typedef area
domain ::
 //Domain area
range ::
 //Range area
relation ::
 //Relation area
```

The `#include` statement in the header in Listing 1 is followed by an optional type definition area. This area always starts with the keyword `typedef` and contains additional type definitions for data structures. The type definition area can appear once anywhere after the last `include` statement and before the `relation` keyword.

The domain area, range area, and relation area are always required (however, they may be empty) and have a fixed order in which they appear in the problem. The domain area and range area, starting with the keywords `domain` and `range` respectively, describe the input and output variables of the CP. A detailed description can be found in Section 3.9. The relation area starts with the `relation` keyword and defines the relation between domain and range; see Section 3.10.

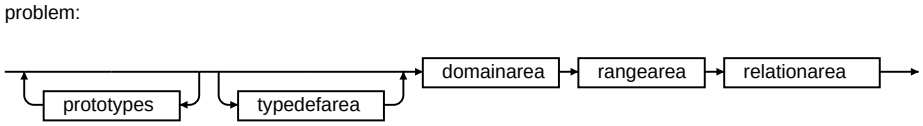

Figure 2: Syntax diagram for the structure of a problem.

Listing 2: NCP problem representation for linear programming.

```
1  fun constraint(y : int<32>[N]) : bool<1>
   fun positivity(y : int<32>[N]) : bool<1>

   domain ::
5   N : int<32>                    //Number of elements in x
    M : int<32>                    //Number of problem constraints
    A : int<32>[M,N]
    b : int<32>[M]
    c : int<32>[N]
10 range ::
    x : int<32>[N]
   relation ::
    cost : int<32>
    i : int<32> = <0 to N-1>       //Temporary used as iteration variable
15  j : int<32> = <0 to M-1>       //Temporary used as iteration variable

    //Objective function
    exists cost { cost == max for x with constraint(x) and positivity(x) { sum i { c[i] * x[i] }
        } };

20  //Problem constraints
    fun constraint(y : int<32>[N]) : bool<1> ::= forall j { sum i { a[j,i] * y[i]} <= b[j] };

    //Non-negativity constraint
    fun positivity(y : int<32>[N]) : bool<1> ::= forall i { y[i] >= 0 } ;
```

3.3 Example: linear programming

Before introducing the language constructs and semantics, an example is used to introduce a problem in the notation. Linear programming models are widely used in industry. Linear programming is a method to optimize linear objective functions and can be expressed as:

$$\begin{aligned} &\text{maximize } \mathbf{c}^\top \mathbf{x} \\ &\text{subject to } A\mathbf{x} \leq \mathbf{b} \\ &\qquad \text{and } \mathbf{x} \geq 0 \end{aligned}$$

where $\mathbf{x}$ is the vector of variables to optimize and A, $\mathbf{b}$, and $\mathbf{c}$ are respectively a coefficient matrix and two coefficient vectors. The *objective function* is the expression to be maximized and $\mathbf{x}$ is subjected to *problem constraints* and a *non-negativity constraint*.

Listing 2 shows the problem capturing the semantics of linear programming. The problem does not need libraries or type definitions, and, as such, the `#include` and `typedef` statements are not present. The header on lines 1 and 2 contain two Boolean predicate prototypes, which are defined later on lines 21 and 24.

Lines 4 through 9 contain the domain area and define the inputs of the problem. There are two 32-bit integer scalars: `N`, the number of elements in $\mathbf{x}$, and `M`, the number of problem constraints. `A` is an `M`×`N` matrix of 32-bit integers with the coefficients A. The arrays `b` and `c` are the coefficient vectors $\mathbf{b}$ and $\mathbf{c}$. Variables in the domain area are *bound* and *physical* variables (see Definitions 4 and 2 given previously).

The range area on lines 10 and 11 defines the outputs of the problem, and the variables defined therein are, just as the domain variables, *physical* variables. The range area is, in terms of syntax, identical to the domain area. In the example, the only output variable is `x`, representing $\mathbf{x}$ in the linear programming problem. However, in contrast to variables in the domain area, variables defined as the range are *free* variables (Definition 5) and have to be bound by a variable-binding operator.

The remainder of the problem is in the relation area. First, four temporary, *free* and *virtual* variables (Definition 3) are declared. The variables `i` and `j`, on lines 14 and 15 can only have values from 0 to, respectively, `N`−1 or `M`−1. All other variables can take values from -2^{31} to $2^{31}-1$, the range of a

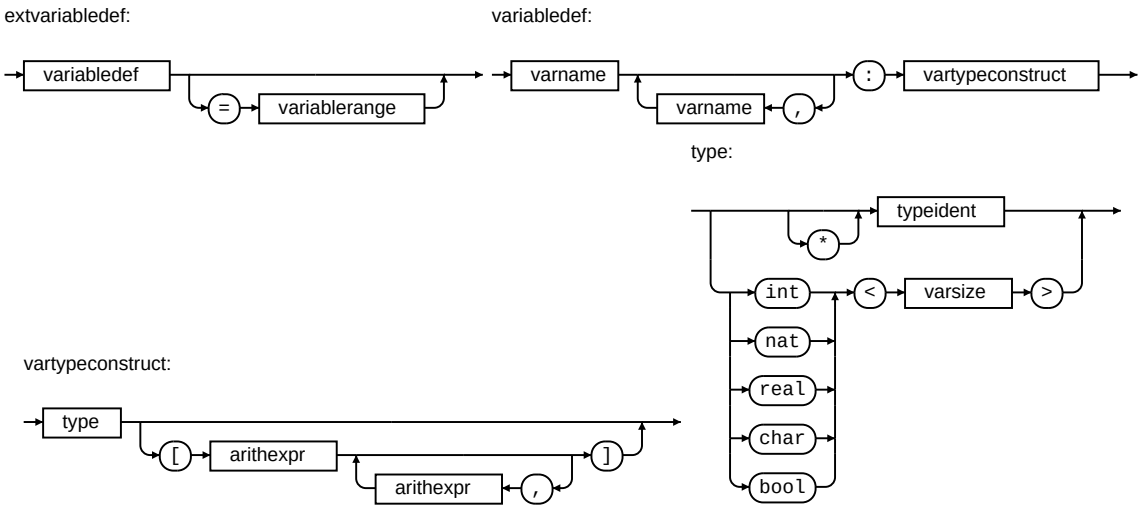

Figure 3: Syntax diagram for variable declarations. Multiple variables with the same type can be defined by separating variable names with commas. The non-terminal "typeident" is a string with the name of a structure.

32-bit signed integer, as discussed in more detail in Section 3.6. The relation area consists of Boolean expressions which describe the relation between the domain and range, as well as Boolean predicates `constraint(...)` and `positivity(...)`. If, for a given set of input and output values, these expressions all evaluate to TRUE, the input, output-pair is valid.

Line 18 represents the objective function. It binds three variables `cost`, `x`, and `i`, the variable-binding operators being `exists`, `max for`, and `sum`. If this Boolean expression evaluates to TRUE, then the output `x` maximizes the objective function given the input. However, `x` should also be subjected to the problem constraints and the non-negativity constraint. Therefore, using the `with` keyword, `x` is subjected to the two Boolean predicates on lines 21 and 24.[1]

3.4 Preprocessor

Before parsing of a problem starts, the source code is first processed by a preprocessor[2]. Two directives are available with the following syntax[3]:

```
#include "filename"
#define identifier token-string
```

The `#include` statement can be used to merge source files. The `#define` statement allows giving a meaningful name to constants, every occurrence of `identifier` in the code is replaced with `token-string`.

3.5 Variable declarations

Variables can be declared in various parts of an NCP problem. A variable declaration consists of the variable name, its type and size, and an optional construct limiting the range of values a variable can represent. Unlike in imperative programming languages, variables are not storage containers. Variable declarations do not allocate storage; they simply associate a type with an identifier.

The NCP grammar productions in Figure 3 show that the type of a variable can be any of the five basic types (see Section 3.6) or the name of a structure; asterisks indicate pointers to structures. The

[1] Note that the function's variable `y` is never bound using a variable-binding operator. `y` is substituted with the bound variable `x` when the predicates are applied.

[2] The compiler performs a system call to GCC to invoke its preprocessor. However, GCC adds additional lines of code to the problem if `#include` and `#define` directives are used. As a result, line numbers in error-messages correspond with lines in the preprocessed file and not in the original problem.

[3] As the current preprocessor is GCC, the full C syntax and other preprocessor directives might also be accepted. However, these are not part of the NCP language and might not be supported in future versions of the compiler (this includes macro-expansions via the `#define` directive).

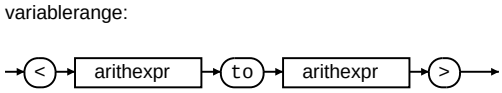

Figure 4: The possible values a variable can represent, can be limited by applying a range limit.

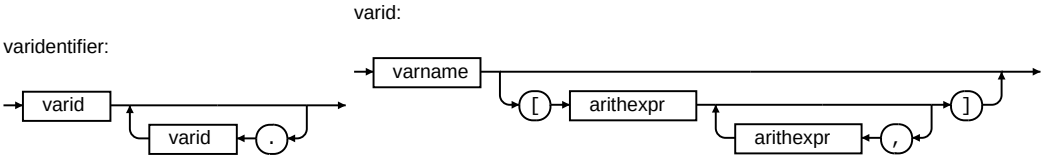

Figure 5: Syntax diagram for a variable identifier used in expressions.

size of a variable in bits is set by the non-terminal "varsize", an unsigned integer. In the case of a multi-dimensional array, the size of each dimension, separated by a comma, is an arithmetic expression (covered in Section 3.12) and is put between brackets `[` and `]`.

Figure 4 shows the optional production for the range limit. This limit can be used to set valid input or output values of the problem or to limit the range of any of the variable-binding operators.

Variables can be used in expressions using the syntax in Figure 5. Elements in structures can be accessed using a dot, '.'.

Variables can be defined in different areas and are lexically scoped. Variables in the domain, range, and relation area are scoped from the moment that they are declared until the end of the problem definition. This makes it possible to use domain variables to indicate the size of arrays, even of arrays in structures as the type definition area can be placed before or after any other area, or between any two areas.

3.6 Available types

Currently, the language supports five different basic types: `bool`, `char`, `nat` (natural numbers including zero), `int` (integers) and `real`. The basic types can be divided into two categories: the Boolean type (`bool`) and the arithmetic types (the others). Operators operate either on the Boolean types or on the arithmetic types (they cannot be mixed) and the resulting type of the operator depends on the operand types according to Table 1. The only exception are the comparison operators, which are always of type `bool` irrespective of their operand types, which can be of arithmetic type.

Table 1: Type inference for non-comparison operators.

First type	Second type	Cast to
char	nat	nat
char	int	int
char	real	real
nat	real	real
nat	int	int
int	real	real

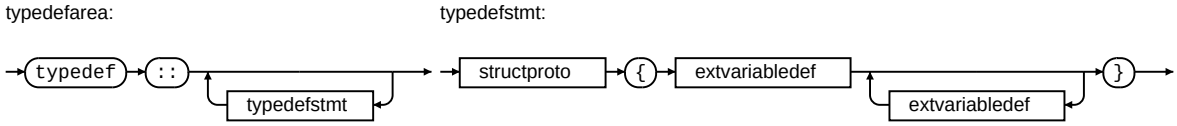

Figure 6: Syntax diagrams related to the type definition area.

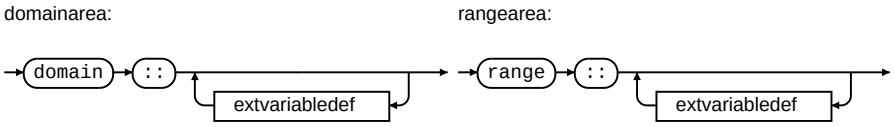

Figure 7: Syntax diagrams for domain and range area.

3.7 Typecasting for Sal

The Sal language contains four types, `bool`, `integers`, `reals`, and `strings`. The NCP types are cast to the corresponding Sal types according to Table 2, when compiling NCP to Sal.

Table 2: Typecasting rules for Sal.

NCP type	Sal type
`bool`	`bool` (or `integers = <0 ... 1>` if used as type for a domain or range variable)
`char`	`integers` (smallest element in the Svm-universe is 0, using ASCII values)
`nat`	`integers` (smallest element in the Svm-universe is 0)
`int`	`integers`
`real`	`reals`

The Sal language does not allow using arithmetic operators with differently typed operands. As such, evaluating the problem on Svm requires typecasting or type promotion. If an expression contains a single type with different sizes, all variables are promoted to the largest size. If an expression contains mixed types, all variables are cast to a single type according to the rules in Table 2, after inferring the type of an expression with the rules in Table 1.

3.8 Type definition area

In the type definition area, starting with the `typedef` keyword, additional data structures can be defined from the basic types. Figure 6 shows the syntax diagrams for the type definition area as well as for a type definition statement. Each structure has a type name and one or more member variables. Note that a structure may include pointers to itself, enabling linked lists, binary trees, and other advanced structures (this is, however, not yet supported by the compiler).

3.9 Domain and range areas

The domain and range area, starting with `domain` and `range` keywords respectively, define the inputs and outputs of a CP. In terms of syntax they are both identical; the meaning of variables, however, is different. The domain area defines bound and physical variables, and these variables are used without variable-binding operators. On the other hand, the range area defines free and physical variables and they have to bound using the `exists` variable-binding operator. The syntax diagrams for the domain and range area can be found in Figure 7. Variables are defined as in Section 3.5.

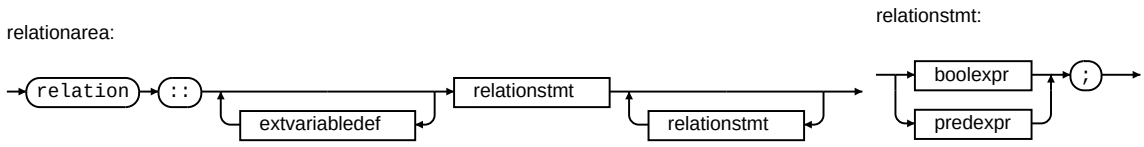

Figure 8: Syntax diagrams for the relation area.

3.10 Relation area

The relation area, Figure 8, captures the relation between domain and range of a computational problem. A relation area starts with variable definitions; these variables are both free and virtual. Following the variable definitions are relational statements, separated by semicolons. These statements can either be a Boolean expression (see Section 3.12), a Boolean predicate, or an arithmetic function (Section 3.14).

The set S_E is defined as the set of Boolean expressions representing the relation $\mathcal{R}$ from Definition 1. S_E is thus a set of individual expressions s which completely define the valid set of input and output values for the CP. The complete set of input and output values is an enumeration of all possible values for the 2-tuple (S_D, S_R). A 2-tuple (S_D, S_R) is a valid 2-tuple of input and output values iff each Boolean expression $s \in S_E$ given S_D and S_R evaluates to TRUE, thus iff $\bigwedge_{s \in S_E} s(S_D, S_R) = \text{TRUE}$. If the set S_E is the empty set $\emptyset$, the problem will evaluate to FALSE for any 2-tuple (S_D, S_R).

3.11 Operators

Three types of operators are available for use in Boolean and arithmetic expressions. Table 3 lists the Boolean operators, Table 4 the comparison operators, and Table 5 the arithmetic operators. The syntax diagrams with production names are found in Figure 9.

Table 3: Boolean operators.

Operator	Description	Precedence
`or`	Logical OR	Low
`and`	Logical AND	High
`not`	Logical NOT	High

Table 4: Comparison operators.

Operator	Description
`==`	Equals
`!=`	Not equals
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`>=<`	Is a permutation of

Table 5: Arithmetic operators.

Operator	Description	Precedence
`+`	Add	Low
`-`	Subtract	Low
`*`	Multiply	High
`/`	Division	High
`%`	Modulo	High
`^`	Exponentiation	High

All comparison operators are defined on scalar values. However, the comparison operators `==`, `!=`, and `>=<` also operate on multi-dimensional arrays. Two multi-dimensional arrays are considered equal if:

- both arrays have an equal number of dimensions;
- the respective dimensions in each array have an equal number of elements;
- elements with the same index in each array have equal values.

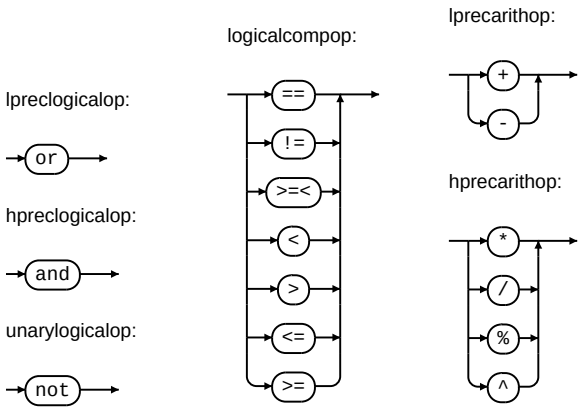

Figure 9: Syntax diagrams and production names for all operators.

For the permutation operator >=<, two multi-dimensional arrays are considered a permutation of each other if:

- both arrays have an equal number of dimensions;
- the respective dimensions in each array have an equal number of elements;
- both arrays have elements with equal values, but do not necessarily have the same index.

3.12 Boolean and arithmetic expressions

The relation $\mathcal{R}$, of a CP definition, is composed of one or more Boolean expressions. Boolean expressions can be built using comparisons of arithmetic expressions. *Expressions* are composed of *terms*, which are linked together via low precedence operators. Terms are made up of *factors*, which are joined using high precedence operators.

Figure 10 shows the syntax diagrams related to Boolean expressions. Boolean expressions can be a single comparison of two arithmetic expressions or constants. However, more elaborate expressions can be made using Boolean variables, operators, predicates (see Section 3.14), or variable-binding operators (see Section 3.13). The Boolean variable-binding operators are `forall` or `exists`.

Arithmetic expressions have similar syntax diagrams as the Boolean expressions, as shown in Figure 11. An arithmetic factor can be a variable or a constant, or combinations joined with arithmetic operators. Four variable-binding operators are available: `sum`, `prod`, `min for`, and `max for`.

3.13 Variable-binding operators

The syntax (Figure 12) of all variable-binding operators is identical. However, the type of the quantified expression depends on the type of the operator, Boolean operators have to be followed by a Boolean expression and the arithmetic operators by an arithmetic expression.

Each variable-binding operator can bind one or more variables. The range over which a variable is bound, is determined by either the range limit set when the variable was defined (see Section 3.5) or by an optional condition in the construct of the binding operator. There are two constructs available: using the keywords `from` and `to`, or using the keyword `with`.

The keywords `from` and `to` set the range between two arithmetic expressions. The keyword `with` sets valid values for the variable using a Boolean expression, values for which the Boolean expression evaluates to TRUE are valid. Note that Boolean expressions can be Boolean predicates. If multiple variables are bound with the same operator, they are all subject to the same range limit if any of the constructs is used.

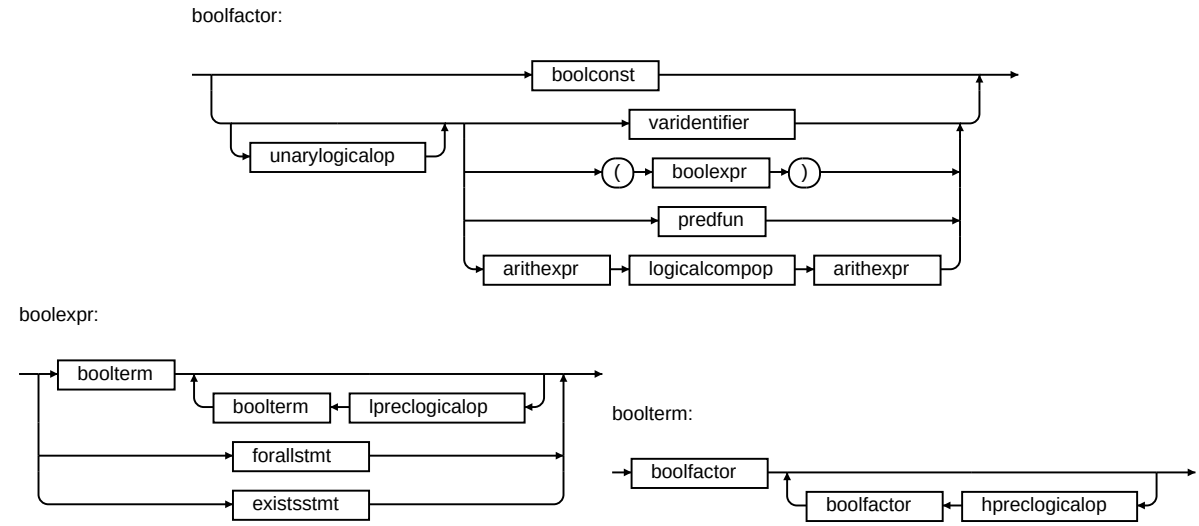

Figure 10: Syntax diagrams for Boolean expressions.

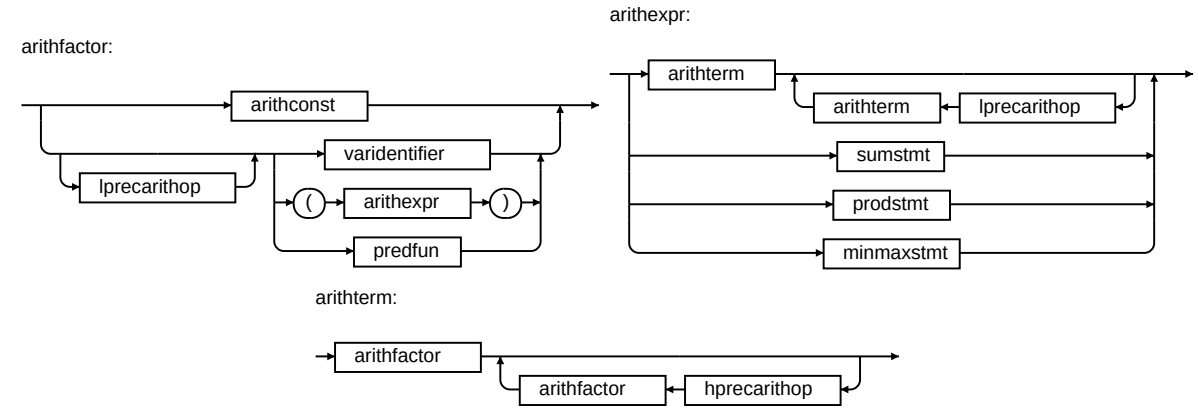

Figure 11: Syntax diagrams for arithmetic expressions.

The six variable-binding operators can be interpreted as follows. Let us define a variable `i` over the range $0 \leq i < 5$:

```
i : int<32> = <0 to 4>
```

There is a Boolean predicate `p` and an arithmetic function `f` (see Section 3.14 for details):

```
fun p(i : int<32>) : bool<1> ::= //some Boolean predicate of i
fun f(i : int<32>) : int<32> ::= //some arithmetic function of i
```

The following Boolean statements will always evaluate to TRUE:

```
forall i { p(i) } == p(0) and p(1) and p(2) and p(3) and p(4)
exists i { p(i) } == p(0) or p(1) or p(2) or p(3) or p(4)
sum i { f(i) } == f(0) + f(1) + f(2) + f(3) + f(4)
prod i { f(i) } == f(0) * f(1) * f(2) * f(3) * f(4)
```

The `min for` and `max for` operators can be expressed using an additional variable `r` and temporary variable `t`:

```
t : int<32> = <0 to 4>
r : int<32>
```

The operation

```
exists r { r == min for i { f(i) } } ;
```

is equal to:

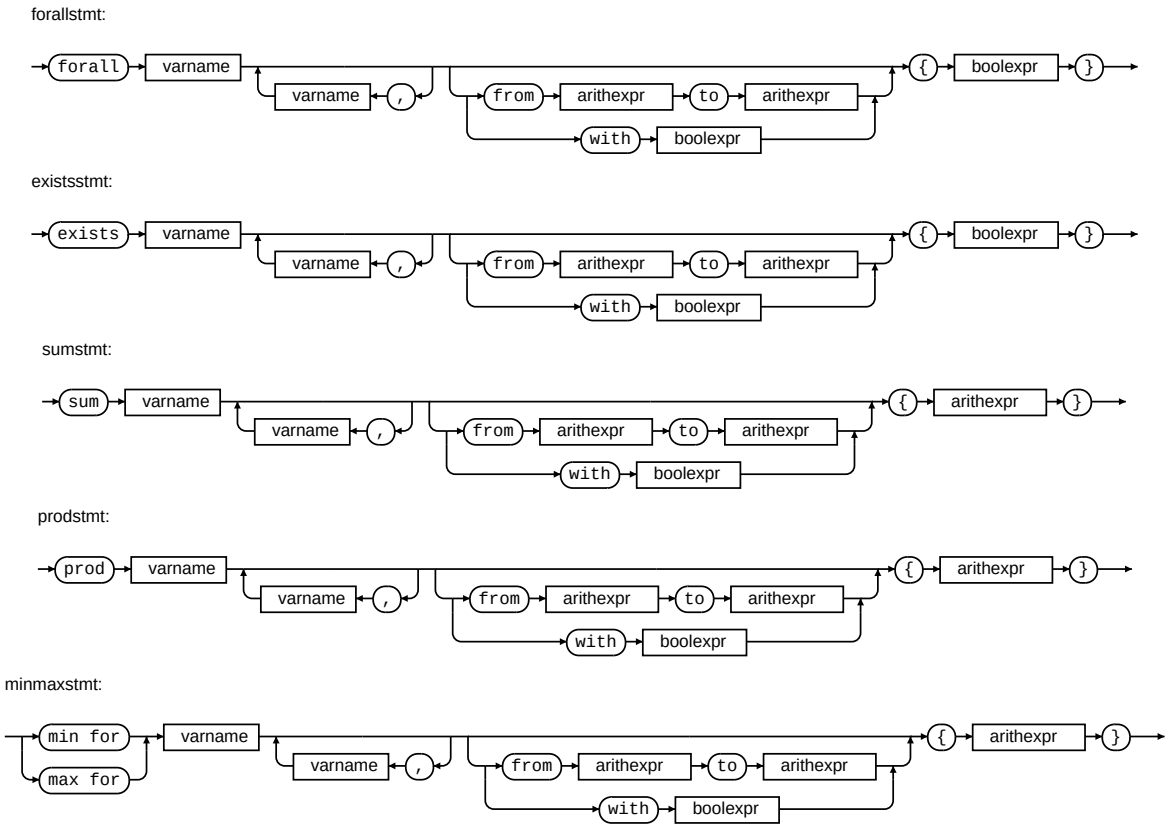

Figure 12: Syntax diagrams for variable-binding operators.

```
fun minelement(i : int<32>) : bool<1> ::= forall t { f(i) <= f(t) } ;
exists r { exists i with minelement(i) { r == f(i) } } ;
```

Rewriting for the `max for` operator is trivial.

Variable-binding operator examples

In order to get a better understanding on how to use the variable-binding operators and the `from`, `to`, and `with` keywords, several examples are given. Let us consider integer variables `x` and `y` (x and y) without a range, a Boolean predicate ($p(x)$), and an arithmetic function ($f(x)$):

```
x,y : int<32>
```

```
fun p(x : int<32>) : bool<1> ::= //some Boolean predicate of x
fun f(x : int<32>) : int<32> ::= //some arithmetic function of x
```

The examples are given in Table 6, they use the variables, predicate, and function defined above.

3.14 Arithmetic functions and Boolean predicates

For ease of programming, the language supports arithmetic functions and Boolean predicates which can be used in arithmetic and Boolean expressions respectively and are expanded during compilation. The syntax of both are identical and they are used in the same way. However, Boolean predicates always evaluate to type `bool`, while arithmetic functions evaluate to any of the other arithmetic types. The syntax diagram is given in Figure 13; they can be used in expressions using the syntax in Figure 14.

A predicate or function definition starts with the keyword `fun`, followed by a unique identifier. This is followed by a set of variable definitions, between parenthesis, for use in the expression. These variables are bound in the expression as they are replaced with (explicitly) bound variables when the predicate or function is used. After the colon, the type to which the function evaluates is specified. The actual

expression follows after the `::=`. The expression can use any of the constructs available, but note that recursion is not allowed.

Table 6: Examples of variable-binding operators in NCP.

Mathematical representation	**NCP representation**
$\forall_x\, p(x)$	`forall x { p(x) }`
$\forall_{x,x\geq 0}\, p(x)$	`forall x with x >= 0 { p(x) }`
$\exists_{x,0\leq x<64}\, p(x)$	`exists x from 0 to 63 { p(x) }`
$\exists_x\, \exists_y\, p(x+y) \wedge \neg p(x-y)$	`exists x,y { p(x+y) and not p(x-y) }`
$\sum_{x=1}^{10} f(x)$	`sum x from 1 to 10 { f(x) }`
$\prod_{x\geq 0} f(x)$	`prod x with x >= 0 { f(x) }`
$\min_x f(x)$	`min for x { f(x) }`
$\max_{0\leq x<32} f(x)$	`max for x from 0 to 31 { f(x) }`

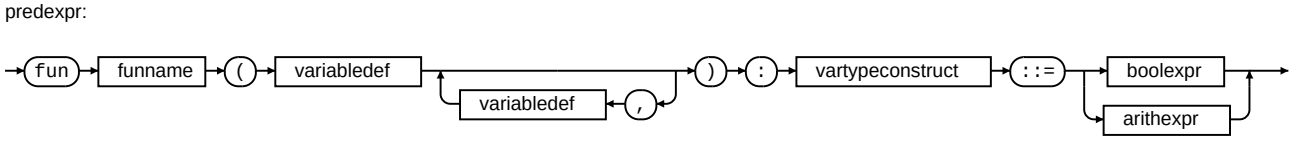

Figure 13: Syntax diagram of an arithmetic function or Boolean predicate.

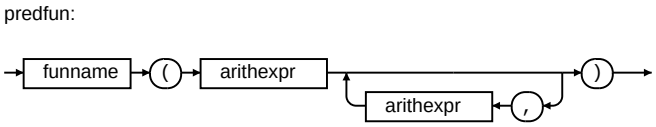

Figure 14: Syntax diagram for using an arithmetic function or Boolean predicate in expressions.

References

[1] R. Jongerius, P. Stanley-Marbell, and H. Corporaal, "Quantifying the common computational problems in contemporary applications," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, 2011.

[2] P. Stanley-Marbell, "Sal/svm: an assembly language and virtual machine for computing with non-enumerated sets," in *Proceedings of the 2010 workshop on Virtual Machines and Intermediate Languages*, VMIL '10, pp. 1:1–1:10, ACM, 2010.

[3] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*. MIT Press, sixth ed., 1985.

[4] N. Wirth, "What can we do about the unnecessary diversity of notation for syntactic definitions?," *Communications of the ACM*, vol. 20, 1977.

[5] N. Immerman, *Descriptive Complexity*. Springer-Verlag, 1998.

A NCP Language Grammar

Listing 3: NCP language grammar in EBNF.

```
digit = "0--9" .
uimm = "1--9" { digit } .
intconst ::= [ "+" | "-" ] uimm .
drealconstext = "." {digit} .
erealconstext = ["." digit { digit }] ("e" | "E") intconst .
realconst ::= intconst (drealconstext | erealconstext) .
char = "ASCII character" .
charconst ::= "'" ["\\"] char "'" .
strident ::= char { char } .
strconst ::= "\"" { char } "\"" .
boolconst ::= "true" | "false" .

intconstlist ::= intconst { "," intconst } .
realconstlist ::= realconst { "," realconst } .
charconstlist ::= strconst | ( charconst { "," charconst } ) .
constlist = intconstlist | realconstlist | charconstlist .
arrayconst ::= strconst | ("{" constlist "}") | ("{" arrayconst { "," arrayconst } "}") .

problem ::= {prototypes} [typedefarea] domainarea rangearea relationarea .
prototypes ::= structproto | predproto .
typedefarea ::= "typedef" "::" {typedefstmt} .
domainarea ::= "domain" "::" {domainstmt} .
rangearea ::= "range" "::" {rangestmt} .
relationarea ::= "relation" "::" {extvariabledef} relationstmt {relationstmt} .

typedefstmt ::= structproto "{" extvariabledef {extvariabledef} "}" .
structproto ::= typeident ":" "struct" .
domainstmt = extvariabledef .
rangestmt = extvariabledef .
relationstmt ::= (boolexpr | predexpr) ";" .

extvariabledef ::= variabledef ["=" variablerange] .
variabledef ::= varname { "," varname } ":" vartypeconstruct .
vartypeconstruct ::= type ["[" extvarlen "]"] .
variablerange ::= "<" arithexpr "to" arithexpr ">" .
type ::= [ "*" ] typeident | basetype"<"varsize">" .
basetype = "int" | "nat" | "real" | "char" | "bool" .
varsize ::= uimm .
extvarlen = varlen {"," varlen} .
varlen = arithexpr .

incname = strconst .
typeident ::= strident .
varname ::= strident .
funname ::= strident .

logicalcompop ::= "==" | "!=" | ">=<" | "<" | ">" | "<=" | ">=" .
lpreclogicalop ::= "or" .
hpreclogicalop ::= "and" .
unarylogicalop ::= "not" .
boolfactor ::= boolconst | [unarylogicalop] ( varidentifier | "(" boolexpr ")" | predfun |
    arithexpr logicalcompop arithexpr ).
boolexpr ::= boolterm {lpreclogicalop boolterm} | forallstmt | existsstmt .
boolterm ::= boolfactor {hpreclogicalop boolfactor} .

arithconst ::= intconst | realconst | charconst | arrayconst .
lprecarithop ::= "+" | "-" .
hprecarithop ::= "*" | "/" | "%" | "^" .
arithfactor ::= arithconst | [lprecarithop] (varidentifier | "(" arithexpr ")" | predfun) .
arithexpr ::= arithterm {lprecarithop arithterm} | sumstmt | prodstmt | minmaxstmt .
arithterm ::= arithfactor {hprecarithop arithfactor} .

predproto = "fun" funname "(" variabledef {"," variabledef} ")" ":" vartypeconstruct .
predexpr ::= predproto "::=" (boolexpr | arithexpr) .
predfun ::= funname "(" arithexpr {"," arithexpr} ")" .

forallstmt ::= "forall" varname {"," varname} [condstmt] "{" boolexpr "}" .
existsstmt ::= "exists" varname {"," varname} [condstmt] "{" boolexpr "}" .
sumstmt ::= "sum" varname {"," varname} [condstmt] "{" arithexpr "}" .
prodstmt ::= "prod" varname {"," varname} [condstmt] "{" arithexpr "}" .
minmaxstmt ::= ( "min for" | "max for" ) varname {"," varname} [condstmt] "{" arithexpr "}" .
condstmt = condrangestmt | condwithstmt .
condrangestmt = "from" arithexpr "to" arithexpr .
condwithstmt = "with" boolexpr .

varidentifier ::= varid { "." varid }.
varid ::= varname ["[" arithexpr {"," arithexpr } "]"] .
```

B Library: math

The math library contains several mathematical functions. Once the header file for library is included, the following variables and arithmetic predicates are available for use in problem definitions:

```
//Defines:
#define PI 3.141592

//Arithmetic functions accepting integers:
fun absi(x : int<32>) : int<32>
fun sqrti(x : int<32>) : int<32>

//Arithmetic functions accepting reals:
fun cos(x : real<32>) : real<32>
fun sin(x : real<32>) : real<32>
fun tan(x : real<32>) : real<32>
fun acos(x : real<32>) : real<32>
fun asin(x : real<32>) : real<32>
fun atan(x : real<32>) : real<32>
fun atan2(y, x : real<32>) : real<32>

fun cosh(x : real<32>) : real<32>
fun sinh(x : real<32>) : real<32>
fun tanh(x : real<32>) : real<32>

fun exp(x : real<32>) : real<32>
//fun frexp(x) : real_32 //C version requires additional return of an int
fun ldexp(x : real<32>, exp : int<32>) : real<32>
fun log(x : real<32>) : real<32>
fun log10(x : real<32>) : real<32>
//fun modf(x : real<32>) : real<32> //C version requires additional return of an int

//fun pow(base, exp : real<32>) : real<32> not needed, as there is the ^ operator
fun sqrt(x : real<32>) : real<32>

fun ceil(x : real<32>) : real<32>
fun fabs(x : real<32>) : real<32>
fun floor(x : real<32>) : real<32>
fun fmod(num, den : real<32>) : real<32>
```

C NCP Design Choices

During the design of the NCP language, several design choices were made which influenced the appearance of the language. Some of the choices are explained in the following subsections.

C.1 Declarative, second-order logic language

The main intention of the language is to capture the semantic properties of a computation independent of algorithms. As a result, the language is chosen to be a purely declarative language without any control flow statements. Recursion via Boolean predicates or arithmetic functions is not allowed, as it would require control flow to terminate the recursion.

The operators available in the language allow to express second-order logic systems; it is known that this is sufficient to capture the semantics of problems in PH, the polynomial-time hierarchy [5]. As it is known that

$$P \subseteq NP \subseteq PH, \tag{1}$$

it is possible to express a sufficiently large amount of problems in the language. The choice for second-order logic is made as it contains no operators which imply control flow and the set of problems which can be expressed is sufficiently large for the language to be of use.

C.2 Readability

The language is intended to be used by programmers, and, as such, it is favored to be human readable. Several of the available constructs were added to improve readability.

Because of the design of the language, problems tend to be written in a single line of code. For larger problem definitions, this quickly becomes unmanageable. As such, both Boolean predicates and

arithmetic expressions can be used to structure a problem. Listing 4 shows two different implementations of the objective function for the linear programming problem, with and without predicates.

Listing 4: Motivational example for predicates and functions.

```
//Objective function without predicates:
exists cost {
    cost == max for x
        with forall j { sum i { a[j,i] * x[i]} <= b[j] } and forall i { x[i] >= 0 }
            { sum i { c[i] * x[i] } }
} ;

//Or the equivalent objective function with predicates:
exists cost {
    cost == max for x with constraint(x) and positivity(x) { sum i { c[i] * x[i] } }
} ;

//Problem constraints
fun constraint(y : int<32>[N]) : bool<1> ::= forall j { sum i { a[j,i] * y[i]} <= b[j] };

//Non-negativity constraint
fun positivity(y : int<32>[N]) : bool<1> ::= forall i { y[i] >= 0 } ;
```

The `for` keyword in `max for` and `min for` was added to improve readability of the operator. First, consider the following sentence:

```
exists x with x > 0 { x == f(i) }
```

It could literally be read as "there exists an x with x larger than 0 such that x is equal to some function $f(i)$", which correctly describes the operator. However, the following (erroneous) statement can be misleading:

```
max x with x > 0 { f(x) }
```

In a similar way as with the `exists` keyword, this could be read as "maximize x with x larger than 0...", but then we are "left" with the function $f(x)$. The operator is however intended to maximize the function $f(x)$. Adding the `for` keyword clarifies the statement:

```
max for x with x > 0 { f(x) }
```

Which could correctly read as "maximize for x with x larger than 0 the function $f(x)$".

C.3 Boolean operators

The NCP language defines three Boolean operators: `or`, `and`, and `not`. The actual keywords are chosen over the C-style operators `||`, `&&`, and `!` for readability. As problems are intended to be written by programmers, editors with syntax highlighting can be used to highlight the keywords. Listing 5 shows three candidate approaches from which the first was chosen.

Listing 5: Tentative options for Boolean operators.

```
with i >= 0 and i < 50 and j >= 0 and not j >= 25
with i >= 0 && i < 50 && j >= 0 && !(j >= 25)
with (i >= 0) && (i < 50) && (j >= 0) && !(j >= 25)
```

D Examples

The following are examples of various computational problems defined in NCP.

D.1 Discrete cosine transform—type II

```
#include "math.cpd"

domain ::
 N : int<32>          //# samples
 x : real<32>[N]   //time domain samples
range ::
 X : real<32>[N]    //freq domain samples
relation ::
 k : int<32>
 n : int<32>

 exists X {
   forall k from 0 to N-1 {
   X[k] == sum n from 0 to N-1 { (x[n] * cos(PI / N * (n + 0.5) * k)) }
  }
 } ;
```

D.2 8x8 2D discrete cosine transform—type II

```
#include "math.cpd"

domain ::
 x : real<32>[8,8]  //time domain samples
range ::
 X : real<32>[8,8]    //freq domain samples
relation ::
 Y : real<32>[8,8]    // half transform
 k,l : int<32> = <0 to 7>
 n : int<32> = <0 to 7>

 exists Y,X {
  forall l, k {
   Y[l,k] == sum n { (x[l,n] * cos(PI / 8 * (n + 0.5) * k)) }
  } and
  forall l, k {
   X[l,k] == sum n { (Y[n,k] * cos(PI / 8 * (n + 0.5) * l)) }
  }
 } ;
```

D.3 k-Means clustering

```
#include "math.cpd"

domain ::
 N : int<32>          //# points
 D : int<32>          //# dim
 k : int<32>          //# clusters
 p : int<32>[N,M]     //points
range ::
 c : int<32>[N] = <0 to k-1>  //cluster assignment, each point has to be assigned to one of
     the k clusters
relation ::
 i, j, z : int<32>
 cost : int<32>

 means(x,y : int<32>, c : int<32>[N]) : int<32> ::=
                                    ( sum z with (z >= 0 and z < N and c[z] == x) { p[z,y] } /
                                      sum z with (z >= 0 and z < N and c[z] == x) { 1 } );

 exists cost {
     cost == min for c {
         sum i from 0 to N-1 { sum j from 0 to M-1 { abs(means(c[i],j,c) - p[i,j]) } } ^ 2
     }
 } ;
```

D.4 Integer sorting

```
domain ::
 N : int<32>
 x : int<32>[N]
range ::
 y : int<32>[N]
relation ::
 n : int<32>

 exists y { forall n from 0 to N-2 { y[n] <= y[n+1] } and y >=< x } ;
```

D.5 Minimum-cost network flow

```
typedef ::
 edge : struct {
  capacity : real<32>
  flow : real<32>
  cost : real<32>
 }
domain ::
 V : nat<32>                    //# Vertices
 inmatrix : edge[V,V]           //Adjacency matrix with edges
 source : nat<32>               //Index of source vertex in adjacency matrix
 sink : nat<32>                 //Index of sink vertex
 d : real<32>                   //Required flow
range ::
 outmatrix : edge[V,V]          //Output matrix with edges
relation ::
 cost : real<32>
 u : nat<32> = <0 to V-1>       //Range u, v, and w over all vertices
 v : nat<32> = <0 to V-1>
 w : nat<32> = <0 to V-1>

 //Capacity constraint: A flow can never exceed the capacity
 fun capacity_constraint(mat : edge[V,V]) : bool<1> ::=
    forall u,v { mat[u,v].flow <= mat[u,v].capacity } ;

 //Skew symmetry: flow u->v is -flow v->u
 fun skew_symmetry(mat : edge[V,V]) : bool<1> ::=
    forall u,v { mat[u,v].flow == -mat[v,u].flow } ;

 //Flow conservation: Kirchoff's Law, total flow of a vertex (except source/sink) is 0
 fun flow_conservation(mat : edge[V,V]) : bool<1> ::=
    forall u with u != source and u != sink { sum w { mat[u,w].flow } == 0 } ;

 //Required flow:
 fun required_flow(mat : edge[V,V]) : bool<1> ::=
    sum w { mat[sink,w].flow } == d and sum w { mat[w,source].flow } == d ;

 //Capacity and cost of in and out matrix are identical, flow depends on several constraints
 fun all_constraints(outmatrix : edge[V,V]) : bool<1> ::=
    forall u,v { outmatrix[u,v].capacity == inmatrix[u,v].capacity } and
    forall u,v { outmatrix[u,v].cost == inmatrix[u,v].cost } and
    capacity_constraint(outmatrix) and
    skew_symmetry(outmatrix) and
    flow_conservation(outmatrix) and
    required_flow(outmatrix) ;

 //Objective: minimize the total cost of the output flow
 exists cost {
  cost == min for outmatrix with all_constraints(outmatrix) {
   sum u,v { outmatrix[u,v].cost * outmatrix[u,v].flow }
 } } ;
```

E Examples of compilation to Sal

Currently, a subset of the NCP problem language can be compiled to Sal. Listings 6 and 7 give an example of a 3×3 integer matrix multiplication CP. The Sal implementation in Listing 7, generated using the compiler, shows registers `I0` to `I17` on lines 2 to 21. These I-registers contain the domain variables `A` and `B`, the `#`-character is a placeholder which the user replaces with input data. After execution, the terminating statement on line 55 prints the values of the range variable `C` on the screen.

Listing 6: Integer matrix multiplication in NCP.

```
1  domain ::
    A : int<32>[3*3] = <0 to 4>
    B : int<32>[3*3] = <0 to 4>
   range ::
5   C : int<32>[3*3] = <0 to 16>
   relation ::
    n,m,k: int<32> = <0 to 2>

   exists C { forall n,m {
10  C[n*3+m] == sum k { A[n*3+k] * B[k*3+m] }
   } } ;
```

Listing 7: Compiled version of integer matrix multiplication to Sal.

```
1  -- A
   I0 = #
   I1 = #
   I2 = #
5  I3 = #
   I4 = #
   I5 = #
   I6 = #
   I7 = #
10 I8 = #

   -- B
   I9 = #
   I10 = #
15 I11 = #
   I12 = #
   I13 = #
   I14 = #
   I15 = #
20 I16 = #
   I17 = #

   -- A universe
   U0 : integers = <0 ... 4>
25 U1 = U0><U0><U0><U0><U0><U0><U0><U0><U0

   -- B universe
   U2 : integers = <0 ... 4>
   U3 = U2><U2><U2><U2><U2><U2><U2><U2><U2
30
   -- C universe
   U4 : integers = <0 ... 16>
   U5 = U4><U4><U4><U4><U4><U4><U4><U4><U4

35 -- n universe
   U6 : integers = <0 ... 2>

   -- m universe
   U7 : integers = <0 ... 2>
40
   -- k universe
   U8 : integers = <0 ... 2>

   -- A input set
45 S0 = {(I0, I1, I2, I3, I4, I5, I6, I7, I8)} : U1

   -- B input set
   S1 = {(I9, I10, I11, I12, I13, I14, I15, I16, I17)} : U3

50 P0 = forall n:U6[1] forall m:U7[1] (_:U5[(1 + ((n * 3) + m))] == sum k:U8[1] from
                    0 to 2 step 1 of (imem[0 + (((n * 3) + k))] * imem[9 + (((k * 3) + m))]))

   S2 = (P0 : U5)

55 echo "Output S2 = " print enum S2
```

Listings 8 and 9 show a second example of an NCP problem compiled to Sal[4]. This example uses variables of type `real` and the `min for` operator. The Sal program this time contains registers `R0` to `R9`, on lines 2 to 13, for the two input arrays. The Sal floating-point comparison tolerances (after the `delta` keyword on lines 16, 20, and 24 and using the `@` symbol on line 38) are requested by the compiler during the compilation process.

Listing 8: Example problem in NCP using the `min for` operator.

```
1  domain ::
    cost : real<32>[5] = <0 to 1>
    multiplier : real<32>[5] = <1 to 5>
   range ::
5   optcost : real<32> = <0 to 5>
   relation ::
    i : int<32> = <0 to 4>

    exists optcost { optcost == min for i { cost[i] * multiplier[i] } } ;
```

Listing 9: Compiled version of the NCP problem in Listing 8.

```
1  -- cost
   R0 = #
   R1 = #
   R2 = #
5  R3 = #
   R4 = #

   -- multiplier
   R5 = #
10 R6 = #
   R7 = #
   R8 = #
   R9 = #

15 -- cost universe
   U0 : reals = <0.000000 ... 1.000000 delta 0.100000*iota>
   U1 = U0><U0><U0><U0><U0

   -- multiplier universe
20 U2 : reals = <1.000000 ... 5.000000 delta 0.100000*iota>
   U3 = U2><U2><U2><U2><U2

   -- optcost universe
   U4 : reals = <0.000000 ... 5.000000 delta 0.100000*iota>
25
   -- i universe
   U5 : integers = <0 ... 4>

   -- cost input set
30 S0 = {(R0, R1, R2, R3, R4)} : U1

   -- multiplier input set
   S1 = {(R5, R6, R7, R8, R9)} : U3

35 P1 = forall jria:U5[1]
                ((rmem[0 + (i)] * rmem[5 + (i)]) <= (rmem[0 + (jria)] * rmem[5 + (jria)]))

   P0 = exists i:U5[1] ((_:U4[1] == @0.050000 (rmem[0 + (i)] * rmem[5 + (i)])) & P1)

40 S2 = (P0 : U4)

   echo "Output S2 = " print enum S2
```

[4]This can be classified as a *difficult-to-name CP* according to [1]. From the NCP code it is clear what problem the NCP description solves, but what *name* would one give such a CP?

F Compiler Command Line Reference

The program to compile NCP problem definitions to any of the supported target languages is CPC (the Computational Problem Compiler). The program supports various command line options. The compiler can be called as follows:

```
cpc [-v0|-v1|-v2|-p] [-I dir] [-d file] [-n file] [-s file] infile
```

F.1 Options

-v0 -v1 -v2

Verbose mode. `-v0` and `-v1` outputs additional information regarding the compiler steps. `-v2` enables, in addition to the information from `-v1`, printing of GNU Bison debugging information.

-I dir

Add the directory `dir` to the list of directories to search for header files (*.cpd).

-d file

Output the AST in dot-format to the file `file`.

-n file

Output the problem in the NCP language to the file `file`. This option is used for debugging the compiler.

-s file

Output the problem in Sal to file `file`.

-p

Disable check for unused variables.

G Vim Syntax Highlighting

Syntax highlighting for high-level NCP problems can be added to vim. In order to enable this, a file describing the syntax is required by vim. The file `~/.vim/syntax/cp.vim` must be created (the location of the file may change with different operating systems or distributions) and the contents of the file are given in Listing 10.

Listing 10: Contents of `cp.vim` to enable syntax highlighting in vim.

```
" Vim Syntax File
" CP notation
"
" IBM Research
"
" Author: jri
"

if exists("b:current_syntax")
 finish
endif

syn keyword CPBlockKeywords typedef domain range relation
syn keyword CPTypedefKeywords struct nat int real char adjmatrix bool
syn keyword CPRelationKeywords and or not forall exists sum prod from to with min max for fun
syn keyword CPBoolKeywords true false

syn match CPComment "//.*$"

syn match CPInclude "^#\=include.*$"
syn match CPDefine "^#\=define.*$"

syn match CPConst '\d\+'
syn match CPConst '[-+]\d\+'

syn match CPConst '\d\+\.\d*f'
syn match CPConst '[-+]\d\+\.\d*f'

let b:current_syntax = "cp"

hi def link CPInclude PreProc
hi def link CPDefine PreProc
hi def link CPBlockKeywords Statement
hi def link CPTypedefKeywords Type
hi def link CPRelationKeywords Statement
hi def link CPBoolKeywords Constant
hi def link CPComment Comment
hi def link CPConst Constant
hi def link CPInt Type
```

The file `~/.vim/filetype.vim` must be created or modified such that files with the `.cp` extension are automatically highlighted. An example file is given in Listing 11.

Listing 11: Example `filetype.vim` for linking the `.cp` extension with the correct syntax file.

```
if exists("did_load_filetypes")
 finish
endif
augroup filetypedetect
 au BufRead,BufNewFile *.cp setfiletype cp
 au BufRead,BufNewFile *.cpd setfiletype cp
augroup end
```