The Omega Calculator and Library, *version 1.1.0*

Wayne Kelly, Vadim Maslov, William Pugh,
Evan Rosser, Tatiana Shpeisman, Dave Wonnacott

November 18, 1996

1 Introduction

This document gives an overview of the Omega library and describes the Omega Calculator, a text-based interface to the Omega library. A separate document describes the C++ interface to the Omega library and we are still working on a third document that describes some of the algorithms used in implementing the Omega library.

The Omega library manipulates integer tuple relations and sets, such as

$$\{[i, j] \to [j, j'] : 1 \le i < j < j' \le n\} \text{ and } \{[i, j] : 1 \le i < j \le n\}$$

Tuple relations and sets are described using Presburger formulas[KK67, Sho77, Coo72, Coo71] a class of logical formulas which can be built from affine constraints over integer variables, the logical connectives $\neg$, $\wedge$ and $\vee$, and the quantifiers $\forall$ and $\exists$. The best known upper bound on the performance of an algorithm for verifying Presburger formulas is $2^{2^{2^n}}$ [Opp78], and we have no reason to believe that our method provides better worst-case performance. However, we have found it to be reasonably efficient for our applications.

The following relation, which maps 2-tuples to 1-tuples: $\{[i, j] \to [i] : 1 \le i, j \le 2\}$ represents the following set of mappings: $\{[1, 1] \to [1],\ [1, 2] \to [1],\ [2, 1] \to [2],\ [2, 2] \to [2]\}$. In addition to variables in the input and output tuples, the Presburger formulas may also contain free variables. This allows parameterized relations to be described. For example, n and m are free in $\{[i, j] \to [i] : 1 \le i \le n \wedge 1 \le j \le m\}$. The language allows new relations to be defined in terms of existing relations by providing a number of relational operators. The relational operations provided include intersection, union, composition, inverse, domain, range and complement. For example, $\{[p] \to [2p, q]\}$ `compose` $\{[i, j] \to [i]\}$ evaluates to $\{[i, j] \to [2i, q]\}$.

Relations are simplified before being displayed to the user. This involves transforming them into disjunctive normal form and removing redundant constraints and conjuncts. During simplification, it may be determined that the relation contains no points or contains all points, in which case the simplified constraints will be False or True respectively. For example, $\{[i] \to [i]\}$ `intersect` $\{[i] \to [i + 1]\}$ evaluates to $\{[i] \to [i'] : \texttt{False}\}$.

The copyright notice and legal fine print for the Omega calculator and library are contained in the README and omega.h files. Basically, you can do anything you want with them (other than sue us); if you redistribute it you must include a copy of our copyright notice and legal fine print.

2 Omega Calculator invocation, syntax and semantics

The calculator reads from the file given as the first argument, or standard input, and prints results on standard output. You can specify several command line flags. Using `-D`mk, where m is a character and k is a digit, sets the debugging level to k for module m. Using `-G`g sets the maximum number of inequalities in a conjunct to g. Using `-E`e sets the maximum number of equalities in a conjunct to E. In version 1.1.0, we intend to change our data structures so that these will not need to be specified. There is also a limit on the maximum number of variables in a conjunct, but this cannot be changed at run-time. It is given by `maxVars` in `oc.h`. We also intend to make this go away.

The input is a series of statements terminated by semicolons. A # indicates that the rest of the line is a comment. The syntax `<<`*filename*`>>` can occur anywhere (and indicates that the text of the file is to be included here. The statements can have one of the forms listed in Figure 1. *RelExpr* is a short form of *Relational Expression*.

Syntax	Semantics
`symbolic` *varList*	Defines the variable names as symbolic constants that can be used in all later expressions.
var := *RelExpr*	Computes the relational expression and binds the variable name to the result.
RelExpr	Computes and prints the relational expression.
`codegen` ...	This is described in Section 5.
RelExpr `subset` *RelExpr*	Prints True if the first relation is a subset of the second, otherwise prints False.

Figure 1: Omega Calculator statements

Figures 2 and 2 show a sample session with the Omega Calculator. Lines starting with # are input to the Omega calculator, the other lines are output from the calculator.

Some relational operations may not preserve the names of input and output variables. If this happens, the variables get default names: `In_`*n* for input variables and `Out_`*n* for output variables, where n is a position of variables in its tuple. When printing, primes are added to variables to distinguish between multiple variables with the same name. In input, primes may also be used to distinguish between multiple variables with the same name: the primes are stripped before being passed to the Omega library.

3 Relations

3.1 Building relations

A relation is an operand of a relational expression. Its syntax is:

{ `[` *InputList* `]` `->` `[` *OutputList* `]` `:` *formula* }

InputList and *OutputList* are lists of tuple elements. A tuple element can be:

var	The corresponding tuple variable is given this name. If a variable with that name is already in scope, an equality is added forcing this tuple variable to be equal to the value of the previous definition.
exp	The tuple variable is unnamed and forced to be equal to the expression.
exp:*exp*	The tuple variable is unnamed and forced to be greater than or equal to the first expression and less than or equal to the second.
exp:*exp*:*int*	The tuple variable is unnamed and forced to be greater than or equal to the first expression and less than or equal to the second, and the difference between the tuple variable and the first expression must be an integer multiple of the integer.
`*`	The tuple variable is unnamed.

The *formula* is optional. If it is omitted, no constraints other than those introduced by the input and output expressions are imposed upon the relation's variables. Otherwise, the formula describes additional constraints on variables used in the relation.

3.2 Sets

In addition to relations, the system can represent *sets*.

When a relation is declared with only one tuple, as in:

{ `[` *SetList* `]` `:` *formula* }

then the relation becomes a set. The variables that are used to describe a set (*SetList*) are called *set variables* rather than input or output variables.

```
# R := { [i] -> [i'] : 1 <= i,i' <= 10 && i' = i+1 };
# R;
{[i] -> [i+1] : 1 <= i <= 9}

# inverse R;
{[i] -> [i-1] : 2 <= i <= 10}
# domain R;
{[i]: 1 <= i <= 9}
# range R;
{[i]: 2 <= i <= 10}

# R compose R;
{[i] -> [i+2] : 1 <= i <= 8}

# R+;
{[i] -> [i'] : 1 <= i < i' <= 10}
#              # closure of R = R union (R compose R) union (R compose R ...

# complement R;
{[i] -> [i'] : i <= 0} union
 {[i] -> [i'] : 10 <= i} union
 {[i] -> [i'] : 1 <= i <= 9, i'-2} union
 {[i] -> [i'] : 1, i' <= i <= 9}

# S := {[i] : 5 <= i <= 25};
# S;
{[i]: 5 <= i <= 25}

# R(S);
{[i]: 6 <= i <= 10}
#            # apply R to S

# R \ S;
{[i] -> [i+1] : 5 <= i <= 9}
#           # restrict domain of R to S

# R / S;
{[i] -> [i+1] : 4 <= i <= 9}
#           # restrict range of R to S

# (R\S) union (R/S);
{[i] -> [i+1] : 4 <= i <= 9}
# (R\S) intersection (R/S);
{[i] -> [i+1] : 5 <= i <= 9}
# (R/S) - (R\S);
{[4] -> [5] }

# S*S;
{[i] -> [i'] : 5 <= i <= 25 && 5 <= i' <= 25}
#             # cross product
```

Figure 2: Example of the Omega Calculator in action: part 1

```
# D := S - {[9:16:2]} - {[17:19]};
# D;
{[i]: 5 <= i <= 8} union
 {[i]: Exists ( alpha : 2alpha = i && 10 <= i <= 16)} union
 {[i]: 20 <= i <= 25}

# T :=  { [i] : 1 <= i <= 11 & exists (a : i = 2a) };
# T;
{[i]: Exists ( alpha : 2alpha = i && 2 <= i <= 10)}

# Hull T;

{[i]: 2 <= i <= 10}

# Hull D;

{[i]: 5 <= i <= 25}

# codegen D;
for(t1 = 5; t1 <= 8; t1++) {
  s1(t1);
  }
for(t1 = 10; t1 <= 16; t1 += 2) {
  s1(t1);
  }
for(t1 = 20; t1 <= 25; t1++) {
  s1(t1);
  }

# codegen {[i,j] : 1 <= i+j,j <= 10};
for(t1 = -9; t1 <= 9; t1++) {
  for(t2 = max(-t1+1,1); t2 <= min(-t1+10,10); t2++) {
    s1(t1,t2);
    }
  }
```

Figure 3: Example of the Omega Calculator in action: part 2

3.3 Presburger formula operations

As mentioned above, the tuples belonging to the relation are defined by a *Presburger formula*. This formula is built from constraints using the operations described in Figure 4.

Name	Notation		
And	*formula* `&` *formula*	*formula* `&&` *formula*	*formula* `and` *formula*
Or	*formula* `|` *formula*	*formula* `||` *formula*	*formula* `or` *formula*
Not	`!` *formula*		`not` *formula*
Exists	`exists` $(v_1, ..., v_n$: *formula*)		
Forall	`forall` $(v_1, ..., v_n$: *formula*)		
Parentheses	(*formula*)		
Constraint	*constraint*		

Figure 4: Presburger formula syntax

3.4 Constraints and arithmetic and comparison operations

A Presburger formula is built from constraints using the operations described in the previous subsection. In this subsection we describe the syntax of individual constraints.

A constraint is a series of expression lists, connected with the arithmetic relational operators `=, !=, >, >=, <, <=`. An example is `2i + 3j <= 5k,p <= 3x-z = t`.

When an constraint contains a comma-separated list of expressions, it indicates that the same constraints should be introduced for each of the expressions. The constraint `a,b <= c,d > e,f` translates to the constraints

$$a \leq c \wedge a \leq d \wedge b \leq c \wedge b \leq d \wedge c > e \wedge c > f \wedge d > e \wedge d > f$$

Expressions can be of the forms (where *var* is a variable, *integer* is an integer constant, and e, e_1, and e_2 are expressions): *var*, *integer*, e, *integer* e, $e_1 + e_2$, $e_1 - e_2$, $e_1 * e_2$, $-e$, (e).

An important restriction is that all expressions in the constraints must be affine functions of the variables. For example, $2 * x$ is legal, $x * 2$ is legal, but $x * x$ is illegal.

4 Relational and set operations

A relational expression is an expression over individual relations. The relational operations defined in the system are listed in Figures 5 and 6. Here r, r_1, r_2 are relations and s, s_1, s_2 are sets.

5 Code Generation

The Omega Calculator incorporates an algorithm for generating code for multiple, overlapping iteration spaces [KPR95]. Each iteration space has an associated statement or block of statements. The syntax is

`codegen` [*effort*] $IS_1, IS_2, \ldots, IS_n$ [`given` *known*]

Each iteration space IS_i can be specified either as a set representing the iteration space, or as an original iteration space and a transformation, T:IS, where IS is the original iteration space and T is a relation defining an affine, 1-1 mapping to a new iteration space. That is, given a point in the original iteration space, the mapping specifies the point in the new iteration space at which to execute that iteration.

The effort value specifies the amount of effort to be used to eliiminate sources of overhead in the generated code. Sources of overhead include if statements an `min` and `max` functions in loop bounds. If not specified, the effort level is 0. The different effort levels are:

-2 Minimal possible effort. Loop bounds may not be finite.

-1 Forces finite bounds

Name	Syntax	Explanation
Union	$r = r_1$ `union` r_2	$x \to y \in r$ iff $x \to y \in r_1$ or $x \to y \in r_1$
	$s = s_1$ `union` s_2	$x \in s$ iff $x \in s_1$ or $x \in s_1$
Intersection	$r = r_1$ `intersection` r_2	$x \to y \in r$ iff $x \to y \in r_1$ and $x \to y \in r_1$
	$s = s_1$ `intersection` s_2	$x \in s$ iff $x \in s_1$ and $x \in s_1$
Difference	$r = r_1 - r_2$	$x \to y \in r$ iff $x \to y \in r_1$ and $x \to y \notin r_1$
	$s = s_1 - s_2$	$x \in s$ iff $x \in s_1$ and $x \notin s_1$
Complement	$r =$ `complement` r_1	$x \to y \in r$ iff $x \to y \notin r_1$
	$s =$ `complement` s_1	$x \in s$ iff $x \notin r_1$
Composition	$r = r_1$ `compose` r_2	$x \to y \in r$ iff $\exists y$ s.t. $x \to y \in r_2$ and $y \to z \in r_1$
Application	$s = r_1(s_2)$	$x \in s$ iff $\exists y$ s.t. $x \to y \in r_1$ and $y \in s_2$
	$r = r_1(r_2)$	Equivalent to r_1 `compose` r_2
Join	$r = r_1.r_2$	Equivalent to r_2 `compose` r_1
		$x \to y \in r$ iff $\exists y$ s.t. $x \to y \in r_1$ and $y \to z \in r_2$
Inverse	$r =$ `inverse` r_1	$x \to y \in r$ iff $y \to x \in r_1$
Domain	$s =$ `domain` r_1	$x \in s$ iff $\exists y$ s.t. $x \to y \in r_1$
Range	$s =$ `range` r_1	$y \in s$ iff $\exists x$ s.t. $x \to y \in r_1$
Restrict Domain	$r = r_1 \setminus s_2$	$x \to y \in r$ iff $x \to y \in r_1$ and $x \in s_2$
Restrict Range	$r = r_1 \;/\; s_2$	$x \to y \in r$ iff $x \to y \in r_1$ and $y \in s_2$
Gist	$r =$ `gist` r_1 `given` r_2	Computes gist of relation r_1 given relation r_2.

Figure 5: Relational operations, part 1

0 Forces finite bounds and tries to remove `if`'s within most deeply nested loops (at a possible cost of code duplication).

1 removes `if`'s within most deeply nested loops and loops one short of being most deeply nested.

2 ... and loops two short of being most deeply nested.

x ... and loops x short of being most deeply nested.

The known information, if specified, is used to simplify the generated code. The generated code will not be correct if known is not true. Currently, the known relation needs to be a set with the same arity and the transformed iteration space.

A discussion of program transformations using this framework is given in [KP93, KP94b, KP94a].

The following is an example of code generation, given three original iteration spaces and mappings. The transformed code requires the traditional transformations loop distribution and imperfectly nested triangular loop interchange. Below, the program information has been extracted and presented to the Omega Calculator in relation form.

Original code

```
      do 30 i=2,n
10      sum(i) = 0.
        do 20 j=1,i-1
20        sum(i) = sum(i) + a(j,i)*b(j)
30      b(i) = b(i) - sum(i)
```

Schedule (for parallelism)

$$T10 : \{\ [\ i\ ] \to\ [\ 0, i, 0, 0\ ]\ \}$$

$$T20 : \{\ [\ i, j\ ] \to\ [\ 1, j, 0, i\ ]\ \}$$

$$T30 : \{\ [\ i\ ] \to\ [\ 1, i-1, 1, 0\ ]\ \}$$

Omega Calculator output:

Name	Syntax	Explanation
Hull	$r =$ `hull` r_1	Computes an single convex region that contains all of r_1. Not as tight as the convex hull, but intended to be fairly fast.
Convex Hull	$r =$ `ConvexHull` r_1	Computes the convex hull [Sch86, Wil93] of r_1. May be prohibitively expensive to compute and/or induce numeric overflow of coefficients (leading to a core dump)
Affine Hull	$r =$ `AffineHull` r_1	Computes the affine hull [Sch86, Wil93] of r_1. May be prohibitively expensive to compute and/or induce numeric overflow of coefficients (leading to a core dump)
Linear Hull	$r =$ `LinearHull` r_1	Computes the linear hull [Sch86, Wil93] of r_1. May be prohibitively expensive to compute and/or induce numeric overflow of coefficients (leading to a core dump)
Conic Hull	$r =$ `ConicHull` r_1	Computes the conic or cone hull [Sch86, Wil93] of r_1. May be prohibitively expensive to compute and/or induce numeric overflow of coefficients (leading to a core dump)
Farkas Lemma	$r =$ `farkas` r_1	Applies Farkas's lemma [Sch86, Wil93] to r_1. In the result, the values of the variables correspond to legal values for coefficients of equations implied by the convex hull of r_1. Also happens to represent the convex hull of r_N using a points and rays representation. See [Wil93] for a more detailed explaination.
Convex Check	$r =$ `ConvexCheck` r_1	Returns the convex hull of r_1 if we can easily determine that it is equal to r_1, otherwise returns r_1.
Pairwise Convex Check	$r =$ `PairwiseCheck` r_1	Checks to see if any two conjuncts in r_1 can be replaced exactly by their convex hull (doing so if possible).
Transitive Closure	$r = r_1+$	Least fixed point of $r^+ \equiv r \cup (r \circ r^+)$
	$r = r_1+$ `within` b	Least fixed point of r^+ for dependence relation r within iteration space b
Conic Closure	$r = r_1@$	Gives a simple dependence relation r such that any linear schedule for all of the dependences in r is non-negative if and only if it is legal for all of the dependences in r_1. Note that $r_1+ \subseteq r_1@$. We previously refered to this as affine closure [KP95].
Cross-product	$r = r_1$ `*` r_2	$x \to y \in r$ iff $x \in r_1$ and $y \in r_2$
Create superset	$r =$ `supersetof` r_1	r is inexact with lower bound r_1
Create subset	$r =$ `subsetof` r_1	r is inexact with upper bound r_1
Create upper bound	$r =$ `upper_bound` r_1	r is an exact relation and is an upper bound on r_1 (all UNKNOWN constraints in r_1 are interpreted as TRUE)
Create lower bound	$r =$ `lower_bound` r_1	r is an exact relation and a lower bound on r_1 (all UNKNOWN constraints in r_1 are interpreted as FALSE)
Get an example	$r =$ `example` r_1	$r \subseteq r_1$ and all variables in r are single-valued
Symbolic example	$r =$ `sym_example` r_1	$r \subseteq r_1$ and all non-symbolic variables in r are single-valued

Figure 6: Relational operations, part 2

```
# Omega Calculator [v1.1.0, Nov 96]:
# #
# # Example of code generation from Omega Calculator documentation
# #
#
# T10:={[i] -> [0,i,0,0]};
#
# T20:={[i,j] -> [1,j,0,i]};
#
# T30:={[i] -> [1,i-1,1,0]};
#
#
# Symbolic n;
#
# IS10 := {[i]: 2 <= i <= n};
#
# IS20 := {[i,j]: 2 <= i <= n && 1 <= j <= i-1};
#
# IS30 := IS10;
#
#
# codegen T10:IS10,T20:IS20,T30:IS30;
for(t2 = 2; t2 <= n; t2++) {
  s1(t2);
  }
for(t2 = 1; t2 <= n-1; t2++) {
  for(t4 = t2+1; t4 <= n; t4++) {
    s2(t4,t2);
    }
  s3(t2+1);
  }
```

6 Inexact Relations

The special constraint UNKNOWN represents one or more additional constraints that are not known to the Omega Library. Such constraints can arise from uses of uninterpreted function symbols or transitive closure (as described below), or when the user explicitly requests it. Such relations require conservative treatment from the library (e.g., subtracting a conjunct containing UNKNOWN from a relation must return that relation, since the unknown constraints might make the conjunct unsatisfiable.)

The `upper_bound` and `lower_bound` operations can be used to produce exact relations from inexact relations. They are produced by treating UNKNOWN constraints as TRUE or FALSE, respectively.

7 Presburger Arithmetic with Uninterpreted Function Symbols

The Omega Calculator allows certain restricted uses of *uninterpreted function symbols* in a Presburger formula. Functions may be declared in the `symbolic` statement as

`symbolic` *Function* (*Arity*)

where *Function* is the function name and *Arity* is its number of arguments. Functions of arity 0 are symbolic constants.

Functions may only be applied to a prefix of the input or output tuple of a relation, or a prefix of the tuple of a set. The function application may list the names of the argument variables explicitly (*not yet supported*), or use the abbreviations *Function(*`In`*)*, *Function(*`Out`*)*, and *Function(*`Set`*)*, to describe the application of a function to the appropriate length prefix of the desired tuple.

Our system relies on the following observation: Consider a formula F that contains references $f(i)$ and $f(j)$, where i and j are free in F. Let F' be F with f_i and f_j substituted for $f(i)$ and $f(j)$. F if satisfiable iff $((i = j) \Rightarrow (f_i = f_j)) \wedge F'$ is satisfiable. For more details, see [Sho79].

Presburger Arithmetic with uninterpreted function symbols is in general undecidable, so in some circumstances we will have to produce approximate results (as we do with the transitive closure operation) [KPRS95].

The following examples show some legal uses of uninterpreted function symbols in the Omega Calculator:

```
# symbolic p(2), n, m;
#
# R  := { [ir,jr] : 1 <= ir <= n && 1 <= jr <= m };
#
# W1 := { [iw,jw] : 1 <= iw <= n && 1 <= jw <= m && p(Set) >= 0 };
#
# W2 := { [iw,jw] : 1 <= iw <= n && 1 <= jw <= m && p(Set) <  0 };
#
# Exposed := R intersection complement ( W1 union W2 );
#
# Exposed;

{[In_1,In_2]  : FALSE }

#
#
# symbolic f(1);
#
# R1 := { [i] -> [j] : 1 <= i = j <= 100 && f(In) <= f(Out)};
#
# R2 := { [i] -> [j] : 1 <= i <= j <= 100 && f(In) = f(Out)};
#
#
# R1 intersection R2;

{[i] -> [i] : 1 <= i <= 100}

#
# R1 union R2;

{[i] -> [j] : f(j) = f(i) && 1 <= i < j <= 100} union
 {[i] -> [i] : 1 <= i <= 100}

#
# R1 intersection complement R2;

{[i] -> [j]  : FALSE }

#
# R1;

{[i] -> [i] : 1 <= i <= 100}
```

8 Reachability

Consider a graph where each directed edge is specified as a tuple relation. Given a tuple set for each node representing starting states at each node, the library can compute which nodes of the graph are reachable from those start states, and the values the tuples can take on.

The syntax is:

```
reachable ( [list of nodes] ) { [node:startstates] | node_i->node_j:transition] }
```

For example,

```
reachable (a,b,c)
  { a->b:{[1]->[2]},
    b->c:{[2]->[3]},
    a:{[1]}};
```

The transitions and start states may be any expression that evaluates to a relation.

You can also compute a tuple set containing the reachable values a t given node; for example:

```
R := reachable of c in (a,b,c)
                { a->b:{[1]->[2]},
                  b->c:{[2]->[3]},
                  a:{[1]}};
```

The current implementation is very straightforward and can be very slow.

9 Current limitations

The transitive closure operation will not work on a relation with uninterpreted function symbols of arity > 0. Any operation that requires the projection of input or output variables (such as composition) may return inexact results if variables in the argument list of a function symbol are projected.

References

[Coo71] D. C. Cooper. Programs for mechanical program verification. In B. Meltzer and D. Michie, editors, *Machine Intelligence 6*, pages 43–59. American Elsevier, New York, 1971.

[Coo72] D. C. Cooper. Theorem proving in arithmetic with multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 91–99. American Elsevier, New York, 1972.

[KK67] G. Kreisel and J. L. Krevine. *Elements of Mathematical Logic.* North-Holland Pub. Co., 1967.

[KP93] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Dept. of Computer Science, University of Maryland, College Park, April 1993.

[KP94a] Wayne Kelly and William Pugh. Determining schedules based on performance estimation. *Parallel Processing Letters*, 4(3):205–219, September 1994.

[KP94b] Wayne Kelly and William Pugh. Finding legal reordering transformations using mappings. In *Lecture Notes in Computer Science 892: Seventh International Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994. Springer-Verlag.

[KP95] Wayne Kelly and William Pugh. Using affine closure to find legal reordering transformations. *International Journal of Parallel Programming*, 1995.

[KPR95] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *The 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Virginia, February 1995.

[KPRS95] Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. Transitive closure of infinite graphs and its applications. In *Eighth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.

[Opp78] D. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323–332, July 1978.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, Chichester, Great Britain, 1986.

[Sho77] Robert E. Shostak. On the sup-inf method for proving presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.

[Sho79] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

[Wil93] D. Wilde. A library for doing polyhedral operations. Technical Report Internal Publication 785, IRISA, Rennes, France, Dec 1993.