The Omega Library

Version 1.1.0

Interface Guide

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

omega@cs.umd.edu
http://www.cs.umd.edu/projects/omega

November 18, 1996

Contents

1 Introduction **3**
- 1.1 About this release . . . 3
  - 1.1.1 Changes since the last release . . . 3
- 1.2 What is the Omega Library? . . . 4
- 1.3 What are tuple relations and sets? . . . 4
- 1.4 What are Presburger formulas? . . . 4
- 1.5 What are uninterpreted function symbols ? . . . 5
- 1.6 What is the UNKNOWN constraint? . . . 5
- 1.7 Manipulating integer tuple relations and sets . . . 5

2 Compiling And Running Programs With The Omega Library **6**
- 2.1 Introduction . . . 6
- 2.2 Avoiding Name Collisions . . . 6
- 2.3 Compiling Templates . . . 7
- 2.4 G++ 2.7.2 . . . 7
- 2.5 Debugging information . . . 8
- 2.6 Enabling one-pass linking . . . 8

3 Primitive Data Structures **9**
- 3.1 Collections and Iterators . . . 9
- 3.2 Sequences . . . 10
- 3.3 Lists and Tuples . . . 10
- 3.4 Generators . . . 11

4 Building New Relations **12**
- 4.1 Creating relations . . . 12
- 4.2 Building formulas . . . 13
- 4.3 Referring to variables . . . 15
- 4.4 Building atomic constraints . . . 16
- 4.5 Finalization . . . 20

5 Querying Existing Relations **21**
- 5.1 Relations during and after queries . . . 21
- 5.2 Printing . . . 21
- 5.3 Simplification and satisfiability . . . 22
- 5.4 Querying variables . . . 22
- 5.5 Iterating through a `DNF` . . . 24
- 5.6 Inexact relations . . . 25

6 Creating New Relations From Old **27**
6.1 Important warning . . . 27
6.2 Upper and Lower bounds . . . 27
6.3 Binary relational operations . . . 27
6.4 Unary relational operations . . . 29
6.5 Advanced operations (Hulls, Farkas lemma, ...) . . . 30
6.6 Low level relational operations . . . 31
6.7 Relational functions that return boolean values . . . 32
6.8 Generating code from relations . . . 32
6.9 Reachability . . . 34
6.10 Avoiding copy overhead . . . 34
6.11 Compressing relations . . . 34
6.12 Reclaiming memory used by `Relations` . . . 34

7 The Uniform Library **35**
7.0.1 Introduction . . . 35
7.0.2 Usage . . . 35

Chapter 1

Introduction

1.1 About this release

This user manual corresponds to version 1.1 of the Omega Library. While we have tested the code extensively, it is still research software, and we are constantly working on and improving the code. In other words, you should expect that there are still undiscovered bugs in this software.

If you use this library, drop us a line and let us know what you're using it for, and what you would like to see in future releases.

1.1.1 Changes since the last release

There are several important changes from the last release.

Inexactness changes

The most visible change to programmers is that relations which contain the constraint UNKNOWN are now handled more sanely. As a result, functions which query whether a relation is satisfiable or is a tautology have changed to reflect the fact that you cannot always get a definitive answer in the presence of UNKNOWN. In addition, there are new functions for checking subsets, plus two new functions for taking upper and lower bounds of relations. Users may have to make some changes to their code in order to compile with the new release. You can safely make the following substitutions and have your code work as with version 1.0:

- `Relation::is_satisfiable` to `Relation::is_upper_bound_satisfiable`
- `Relation::is_tautology` to `Relation::is_obvious_tautology`
- `Is_Subset` to `Must_Be_Subset`

Variables types replaced by variable kinds

It used to be that variables has *types*, such as input variable, output variable, global variable, and so on. To prepare the way for variables having types such as integer, enumeration and rational, and avoid the confusion that would be caused by having both *types* floating around, the previous uses of variable types were changed to variable *kinds*.

Old Name	New Name	File
Var_Type	Var_Kind	pres_gen.h
Global_Type	Global_Kind	pres_var.h
Var_Type Var_Decl::type()	Var_Kind Var_Decl::kind()	pres_var.h
void set_type(Var_Type v)	void set_kind(Var_Kind v)	pres_var.h
Global_Type Global_Var_Decl::type()	Global_Kind Global_Var_Decl::kind()	pres_var.h

Convex Hull

We now provide an exact convex hull computation, as well as variations such as affine hull, linear hull and conic hull [Sch86].

The Uniform library

This distribution includes the Uniform library, a source to source parallelizing transformation system.

Reachability

The library now includes a function to compute graph reachablity problems.

1.2 What is the Omega Library?

The Omega Library is a set of C++ classes for manipulating integer tuple relations and sets. We use the Omega Library in many of our research projects in the area of compilation for high-performance computers. Current applications include dependence analysis, program transformations, generating code from transformations, and detecting redundant synchronization. It is also the basis for the Omega Calculator, which is described separately. This manual describes how to use the Omega library in your programs.

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.

1.3 What are tuple relations and sets?

An integer k-tuple is simply a point in $\mathcal{Z}^k$. An *integer tuple relation* is a mapping from tuples to tuples. The tuple that is being mapped from is refered to as the input tuple and the tuple that is being mapped to is refered to as the output tuple. All the integer tuple relations we consider map from k-tuples to k'-tuples for some fixed k and k'. We refer to k and k' as the input and output arities respectively. A *integer tuple set* is a set of k-tuples, for some fixed k. We'll often abbreviate integer tuple relations and integer tuple sets as relations and sets respectively.

The sets and relations that we use may be infinite or may depend on the values of other variables, so it is generally not possible to describe them simply by enumerating their tuples or pairs of tuples. Instead, we introduce variables that correspond to each of the positions in the input and output or set tuples and construct a formula that has these and other variables as free variables. These variables are refered to as input, output or set variables as appropriate. For example, we would represent the set of all even numbers as: $\{ [i] \mid \exists \alpha \text{ s.t. } i = 2\alpha \}$ In this case we introduce a single set variable because we are describing a set of 1-tuples. The formula in this case is $\exists \alpha$ s.t. $i = 2\alpha$, note that i is the only free variable in this formula. If we wanted to represent the set of all positive even numbers less than n we would use: $\{ [i] \mid \exists \alpha \text{ s.t. } i = 2\alpha \ \wedge 0 < i < n \}$ We refer to variables such as n as symbolic or global variables. They are global in the sense that can be used in more than one relation and they represent the same value in all relations. This becomes important when we apply relational operations that combine variables and constraints from different relations. As another example, to represent the relation corresponding to integer addition we would use: $\{ [i_1, i_2] \rightarrow [j_1] \mid i_1 + i_2 = j_1 \}$

The Omega library can represent relations and sets that can be described by Presburger formulas (possibly with limited uses of uninterpreted function symbols).

1.4 What are Presburger formulas?

Presburger formulas [KK67] are those formulas that can be constructed by combining affine constraints on integer variables with the logical operations $\neg$, $\wedge$ and $\vee$, and the quantifiers $\forall$ and $\exists$. The affine constraints can be either equality constraints or inequality constraints (abbreviated as EQs and GEQs respectively). For example, the formulas in the previous section are Presburger formulas. There are a number of algorithms for testing the satisfiability of arbitrary Presburger formulas [KK67, Coo72, PW93]. 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 will provide better worst-case performance. However, our method may be more efficient for many simple cases that arise in our applications.

1.5 What are uninterpreted function symbols ?

Presburger arithmetic can be extended to allow *uninterpreted function symbols*. Formulas in this extended class may contain terms representing the application of a function to a list of argument terms. For example, the following extended presburger formula contains an uninterpreted function symbol F: $1 \leq i \leq i' \leq \mathtt{n} \wedge F(i) \neq F(i') \wedge 1 \leq j, j' \leq \mathtt{m}$ The functions are termed "uninterpreted" because the only thing we know about them is that they are functions: two applications of a function to the same arguments will produce the same value.

Applications of function symbols can be used to represent values that vary with the values of certain other variables. For example, when we perform dependence analysis with the Omega Library, we would use a scalar global variables to represent the value of a symbolic constant in the program, and a function of arity 2 to represent the values taken on by a non-linear expression nested in two loops (see [PW94] for a more detailed discussion of, and examples of, the use of function symbols to represent non-linear terms).

Downey ([Dow72]) proved that full Presburger arithmetic with uninterpreted function symbols is undecidable. We therefore restrict our attention to a subclass of the general problem, and produce approximations whenever a formula is outside of the subclass. Whenever an approximation is produced, it is marked as either an upper bound or lower bound as appropriate. We currently restrict our attention to formulas in which all function symbols are free, and functions are only applied to a prefix of the input or output tuple. For example, a binary function could be applied to the first two input or the first two output variables of a Relation (or the first two variables of a set's tuple).

1.6 What is the UNKNOWN constraint?

A conjunction may contain the constraint UNKNOWN. UNKNOWN indicates the existence of one or more other constraints in the Presburger formula that aren't known. Such constraints could arise from the need to approximate (as from some cases of uninterpreted function symbols or transitive closure), or because the user added it to the formula. A relation which includes the UNKNOWN constraint is said to be inexact. Inexactness can be removed by taking an upper or lower bound on the inexact relation. Inexactness is discussed further in sections 4.4, 5.3 and 5.6, and creating upper and lower bounds is discussed in section 6.2.

1.7 Manipulating integer tuple relations and sets

A common way to use the library is to build relations and/or sets describing your particular problem, perhaps combine them using the relational operators, and then query them in some way (checking information about a particular variable, checking to see if its formula is satisfiable, or just printing them to the screen.)

In Chapter 4, we describe how to build new relations and sets. In Chapter 5,we describe how to examine existing relations. This includes generating a user readable string representation of a set or relation, and examining the conjuncts and constraints of a relation once it has been simplified and converted into disjunctive normal form (DNF). In Chapter 6, we describe how to create new relations from old relations by applying relational operations such as intersection, union, domain, range, and composition.

In Chapter 2, we discuss some issues related to the compilation of the Omega Library and programs that use it. In Chapter 3, we describe some of the primitive data structures used in the library (such as strings and tuples).

Chapter 2

Compiling And Running Programs With The Omega Library

2.1 Introduction

To use the Omega Library in an existing C++ application, it is necessary to:

- Use the makefile included in the release to make libomega.a if it doesn't already exists.
- Include `-lomega` as an option to the linker.
- Include `-Ldir` as an option to the linker, where `dir` is a directory that contains `libomega.a`.
- Add the line `#include "omega.h"` to each application source file that references Omega library data structures or functions.
- If the include file `omega.h` and the associated `omega` subdirectory have not been placed in a standard include directory on your system, use `-Idir/include` as an option to the compiler, where `dir` is a directory that contains the include directory from this release.
- Make any modifications that are needed to get the Omega Library and your code to compile with compatible compilers (see Section 2.3).

2.2 Avoiding Name Collisions

Some of the global names in our library are simple enough that they may conflict with names used in your code or some other library you use. We have taken the following steps to ensure that such conflicts can be resolved easily - these are based on vague memories of something from the documentation of the `InterViews` project, so to the degree that the approaches are similar, that project should get the credit. In a header file that defines an obvious name, such as `String`, we replace (via `#define`) that name with some less obvious name, such as `Omega_String`. This ensures that these obvious names do not appear in the library, even though we use them in our code.

In case of conflict, just `#undef` the obvious name after the inclusion of `omega.h`, but before the inclusion of the header file that causes the conflict. For example, if you need to use both the Omega Library and Motif (which defines the name `String`), first `#include omega.h`, then `#undef String` (or, alternatively, `#include omega/basic/leave_String.h`, which does this), and then `#include` the necessary Motif header files.

If you find a conflict in a header that we have not prepared as described above, let us know so that we can fix this in the next release. In the meantime, it should be fairly easy to install this fix yourself and re-compile.

2.3 Compiling Templates

This software should compile correctly with either g++ 2.5.8 or g++ 2.6.3, as long as the proper flags are used (`-fno-implicit-templates` and `-DDONT_INCLUDE_TEMPLATE_CODE` for g++ 2.6.3). We take advantage of some of the new features of g++ 2.6.3 to ensure that the code for templates is not duplicated (this reduces the size of the executable by about $\frac{2}{3}$). This makes the compilation a bit more complicated: this section describes some of the techniques we use to make it work reasonably well.

A parameterized class's declaration is in a "`.h`" file and the code for its member functions in a "`.c`" file. The "`.c`" files for parameterized classes are kept in the include directory, as they are `#include`-d rather than compiled separately. Each class' "`.h`" file also contains a macro that expands to list all the other templates associated with a template you might be using. These macros, including those for types used by the Omega Library, should all be used expanded in one file, to avoid redundant definitions. The file "`include/omega/PT-omega.c`" contains a list of all the templates used by the Omega Library, and can be used to simplify the construction of a single file that contains all parameterized types. For example, the file "`PT.c`" is used in the compilation of the Omega Calculator: It `#includes` `omega/PT-omega.c` and explicitly lists the other parameterized types used by the calculator.

The functions of these features depend on the compiler being used:

G++ 2.5.8

The declaration and definition of all template functions must be given in every header file. To accomplish this, the header file for a given class **#include**s the "`.c`" file with the code for that class's member functions. The "`.c`" file should not be compiled directly. "`PT-omega.c`" and "`PT.c`" are not needed - fortunately g++ 2.5.8 ignores their contents.

G++ 2.6.3

The definitions of the external template functions must *not* be included in multiple source files. We therefore use `-fno-implicit-templates` and `-DDONT_INCLUDE_TEMPLATE_CODE` when compiling - the latter prevents inclusion of the "`.c`" files by the headers. The file "`PT.c`" `#undefs` `DONT_INCLUDE_TEMPLATE_CODE` before including the headers, so it will get the function definitions. It `#includes` "`PT-omega.c`", so this one source file will contain a copy of each template needed anywhere in the Omega Calculator. Note that our instantiate macros contain some redundancies, and the compiler will issue warnings about them, so warnings for `PT.c` are turned off in the makefile.

2.4 G++ 2.7.2

The library can also be compiled with G++ 2.7.2, using the same options as for G++ 2.6.3. Due to some changes in the C++ language standard, the compilation produces lots of warnings, but correct code is produced.

The first change is to the scope of variables declared inside a `for` loop. Most of our code has been adapted to use syntax that works with both the old and new rules. We use the `-fno-for-scope` option to allow the old syntax in places where it hasn't been fixed. Eventually, we will convert all such loops, but until most compilers use the new scoping, we want to allow the use of older compilers.

The second change is to prohibit passing an rvalue as a non-const reference parameter. We often re-use parts of `Relations` in relational operators, which destroys them; to avoid this destruction when passing a relation, we use a `copy` function. Unfortunately, this results in a warning, since the result of `copy` is an rvalue. We hope to address this problem in a future release.

Other C++ compilers

Please let us know if you succeed in getting our code to compile with something other than g++.

2.5 Debugging information

The library has some internal debugging output that you can turn on. It may not be useful to you, since it makes reports about internal functions of the library.

All debugging goes to the file `DebugFile`, which defaults to `stderr`. You can assign a `FILE*` to that variable to redirect debugging output. To turn off all debugging, call the function `all_debugging_off()`. To use the flags as they are used in the calculator (without the calculator debugging flag) call the function `process_pres_debugging_flags(char *arg,int &j)` where `arg` points to a character string with [letter][digit] pairs, each [digit] is between 1 and 4, and `j` is an offset into the string at which to start processing flags. If the digit is omitted, it is assumed to be 1. The letters mean:

- a: all debugging flags
- c: omega core internal debugging
- g: code generation debugging
- p: Presburger formula debugging (simplification, creation)
- r: relational operators
- o: print results upon exiting omega core
- t: transitive closure debugging

As an example, your program might use the `-D` flag to introduce Omega library debugging flags, and a use of this might be `-Dp1r1`. Your code should recognize the `-D` flag and pass a pointer to the beginning of the `p1r1` string to `process_pres_debugging_flags`.

2.6 Enabling one-pass linking

The g++ compilers perform linking in one pass. This puts some restrictions on the structure of the library being linked. Namely, it should be possible to topologically sort all the library object files according to the call graph. This restriction doesn't apply to the virtual functions, which are always linked in. Unfortunately, the logical structure of the Omega library does not allow to us distribute functions among the files in a such a way that this restriction holds. To resolve this problem, we force some functions to be always linked into the executable (just as the virtual functions are always linked in.) This is implemented in the `lib_hack.c` file, whose only purpose is to enable this trick. The target `check_order` in the makefile allows you to check that a correct ordering of functions in the library exists (accounting for the functions that are always linked). You never need to do this if you only use the functions the library provides; only users who extend the library should ever need to check this.

Chapter 3

Primitive Data Structures

The library uses a few generic data structures extensively. All of these data structures are templatized, so they can be used with any type that defines the comparison operations they require. This section provides an introduction to the most frequently used operations defined on these data structures.

3.1 Collections and Iterators

All of the one dimensional collection types we use are derived from the `Collection` class. The elements contained in a collection can be examined via the creation of an iterator - different collections will have different kinds of iterators, all of which are derived from class `Iterator`. Any collection must be able to allocate (via **new**) the appropriate type of iterator. The iterator can be dereferenced (via **operator ***) to access the current element or incremented (via **operator ++**) to move to the next element. If an iterator is used where a boolean is required (e.g. in the conditional expression of a **for** loop), the **operator bool()** function will return **true** iff there are more elements to be iterated over. The class declarations include the following lines:

```
template<class T> class Collection {
public:
        Iterator<T> *new_iterator();
    Any_Iterator<T>  any_iterator();

                int size() const;
};

template<class T> class Iterator {
public:
          T &  operator*();
       void    operator++();
               operator bool() const;
};
```

We can therefore enumerate the elements of any collection as follows:

```
int sum(Collection<int> &c)
    {
    Iterator<int> *i;
    int s = 0;

    for (i = c.new_iterator(); (*i); (*i)++)
        s += *(*i);
```

```
        delete i;
        return s;
        }
```

As it is easy to forget that the `new_iterator` must later be **delete**d, we have provided the type `Any_Iterator`, which handles this issue automatically.

```
int sum(Collection<int> &c)
    {
    int s = 0;

    for (Any_Iterator i = c.any_iterator(); i; i++)
        s += *i;

    return s;
    }
```

Caveats

If a collection is changed (elements are added or removed), all the behavior of any iterators on that collection is, in general, not defined.

Both the `new_iterator` and `any_iterator` functions cause heap allocation. If we know what kind of collection we are working with, we can avoid the overhead inherent in this allocation by using a specialized iterator for that type of collection (such as a `List_Iterator` for a `List`, as in Section 3.3).

Note that iterators can also be manipulated via an alternate set of functions: `live()`, `curr()`, and `next()`. These are particularly useful in a debugger, where the other syntax can be difficult to use.

Many classes in the Presburger library have iterators to walk through the structure – iterators over conjunctions in a formula in disjunctive normal form, iterators over constraints in a conjunction, or iterators over variables in a constraint. These will be discussed in more detail later.

3.2 Sequences

Collections in which the elements are ordered are derived from class `Sequence`. The elements of a sequence can be accessed via subscripting, though the efficiency of this operation varies with the particular kind of sequence. The `index` function gives the index of an element - for any value `v` in a list `l`, `l[l.index(v)] == v`.

Note that subscripts start an 1, not 0; the `index` function returns 0 if the element is not in this list.

3.3 Lists and Tuples

The most commonly used collections are `List` and `Tuple`; you may also encounter `Bag`, `Ordered_Bag` and `Map` (which is not derived from `Collection`), although no public functions return these types.

`List`s are sequences with $O(n)$ access time for the n^{th} element. They add the following operations to those of class `Sequence`:

```
template<class T> class List : public Sequence<T> {
public:
         int length() const;
         int empty() const;

           T &front();
```

```
// insertion/deletion on a list invalidates any iterators
// that are on/after the element added/removed

           T remove_front();
        void prepend(const T &item);
        void append(const T &item);
        void ins_after(List_Iterator<T> i, const T &item);
        void del_front();
        void del_after(List_Iterator<T> i);
        void clear();

        void join(List<T> &consumed);
};
```

Tuples are essentially re-sizable arrays: they provide $O(1)$ time access to elements. They provide bounds checking.

```
template <class T> class Tuple : public Sequence<T> {
public:
    Tuple<T>& operator=(const Tuple<T>&);

    int length() const;
    int  empty() const;

    void delete_last();
    void append(const Tuple<T> &);
    void append(const T &);
    void join(Tuple<T> &consumed);
    void clear();
};
```

These classes have associated iterator classes: `List_Iterator` and `Tuple_Iterator`, which can be initialized with a `List` or `Tuple` to be iterated over, allowing iteration without the overhead of heap allocation that is incurred when we use the more general `new_iterator` or `any_iterator` functions:

```
        int sum(List<int> &l)
            {
            int s = 0;

            for (List_Iterator<int> i = l; i; i++)
                s += *i;

            return s;
            }
```

3.4 Generators

Generators are similar to iterators in that they provide us with sequential access to a collection of values. Iterators provide read and write access to elements that are actually stored in some collection. The sequence of values provided by a generator need not actually be stored in any collection, and thus a generator does not provide an lvalue.

Chapter 4

Building New Relations

The upcoming chapters will share three running examples; the figures that contain the code can be concatenated to produce a valid example program for the Omega Library (the file `library_example.c`, distributed with the postscript file for this manual, contains this code). We will see how to create the following very simple set, slightly more complicated set, and relation:

$$
\begin{aligned}
S1 &:= \{\ [t] \mid 1 \le t \le \mathtt{n}\ \};\\
S2 &:= \{\ [x] \mid (0 \le x \le 100 \wedge \exists y \text{ s.t. } (2\mathtt{n} \le y \le x \wedge y \text{ is odd})) \vee x = 17\ \};\\
R &:= \{\ [i,j] \rightarrow [i',j'] \mid 1 \le i \le i' \le \mathtt{n} \wedge \neg(F(i) = F(i')) \wedge 1 \le j, j' \le \mathtt{m}\ \};
\end{aligned}
$$

In this chapter, we'll describe how to construct relations and sets from scratch. We'll describe how to declare relations; create and use the variables used within Presburger formulas; and how to build Presburger formulas that describes membership in the set or relation.

4.1 Creating relations

What we've referred to up until now as "sets" and "relations" are implemented as a single class, `Relation`. Each relation is marked as being either a set or a relation. Most functions on relations don't care whether they operate on sets or relations, but some require either a set or a relation, and they will check.
We provide the following constructors for `Relation`s:

`Relation()`
: This is a default constructor for the class. Relations created this way are known as null relations and must be assigned a value before they can be used. This constructor is provided to allow the creation of arrays of relations.

`Relation(int n_input, int n_output);`
: This constructor creates a relation with `n_input` input variables and `n_output` output variables. It can't be used in any functions yet, because it doesn't have a Presburger formula to describe which tuples it contains.

`Relation(Non_Coercible<int> nci);`
: This constructor creates a set. In normal use, you should pass an `int` to this function to get a set with that many variables in its tuple. (The class `Non_Coercible<int>` is used as a C++ trick so that you couldn't, for example, pass an `int` to a function expecting a `Relation`, and have the compiler construct a `Relation` from it on the fly.)

`Relation(const Relation &r, Conjunct *c)`
: This is used to create a relation by copying a conjunction of constraints c, from some other relation r. Conjuncts are created when a relation is simplified into disjunctive normal form and will be described in Chapter 5.

The input, output, or set variables can be given names:

`void Relation::name_input_var(int nth, String s)`
Set the name of the `nth` input variable to `s`.

`void Relation::name_output_var(int nth, String s)`
Set the name of the `nth` output variable to `s`.

`void Relation::name_set_var(int nth, String s)`
Set the name of the `nth` set variable to `s`.

If you don't name the variables yourself, each variable will get a name that depends on its position in the input or output tuple. If two variables in the same relation are given the same name, the print functions will add "primes" to one of them to distinguish them.
The following are some simple functions that extract information about a set or relation:

`bool Relation::is_set()`
Return true if the relation is a set, false if it is a relation.

`int Relation::n_inp()`
Find out how many variables are in a relation's input tuple.

`int Relation::n_out()`
Find out how many variables are in a relation's output tuple.

`int Relation::n_set()`
Find out how many variables are in a set's tuple.

The first six lines of Figure 4.1 show how the `Relation` constructors and some of the functions above are used to create the relations in our examples.

4.2 Building formulas

Presburger formulas are represented as a tree of formula nodes. `Formula` is the base class from which all classes of formulas nodes are derived. Depending on their class, formula nodes can have zero or more children. Children can be either other formula nodes or "atomic" constraints (single equality, inequality, or stride constraints). `Formula` is an abstract base class; a plain `Formula` node can never be used in a tree.
The various subclasses of formulas are:

`F_And`
Represents the logical conjunction of its children nodes. An `F_And` node with no children represents "True". In our current implementation, atomic constraints can only be added as children of `F_And` nodes.

`F_Or`
Represents the logical disjunction of its children nodes. An `F_Or` node with no children represents "False".

`F_Not`
Represents the logical negation of its single child node.

`F_Declaration`
This subclass of `Formula` is the abstract base class for all subclasses of `Formula` that can contain variable declarations.

`F_Forall`
`F_Forall` nodes have associated with them one or more variables. These variables are universally quantified in the formula represented by the `F_Forall` node's single child node.

```
#include <omega.h>

main()
    {
    Relation S1(1), S2(1), R(2,2);
    S1.name_set_var(1, "t");
    S2.name_set_var(1, "x");

    assert(!R.is_set());
    assert(S1.is_set());
    assert(S2.is_set());

    Free_Var_Decl n("n");
    Free_Var_Decl m("m");
    Free_Var_Decl f("F", 1);

    Variable_ID S1s_n = S1.get_local(&n);
    Variable_ID S2s_n = S2.get_local(&n);

    Variable_ID Rs_n = R.get_local(&n);
    Variable_ID Rs_m = R.get_local(&m);
    Variable_ID Rs_f_in  = R.get_local(&f, Input_Tuple);
    Variable_ID Rs_f_out = R.get_local(&f, Output_Tuple);

    Variable_ID i = R.input_var(1);
    Variable_ID j = R.input_var(2);
    Variable_ID i2 = R.output_var(1);
    Variable_ID j2 = R.output_var(2);

    Variable_ID t = S1.set_var(1);
    Variable_ID x = S2.set_var(1);
```

Figure 4.1: Example, Part 1: Declaring Relations and Variables

F_Exists

F_Exists nodes have associated with them one or more variables. These variables are existentially quantified in the formula represented by the F_Exists node's single child node.

Children can be added to any subclass of formula using the following member functions. They all return pointers to the newly created child node.

```
F_And * Formula::add_and()

F_Or * Formula::add_or()

F_Not * Formula::add_not()

F_Forall * Formula::add_forall()

F_Exists * Formula::add_exists()
```

The `F_And * Formula::and_with()` member function is also useful. Its effect is equivalent to replacing the formula that it is called on with an `F_And` node with the old formula as one of its children and another `F_And` node as its other child. This second `F_And` node is returned as the result.

Analogous versions of these member functions exist for the Relation class. A `Relation` can have only one child formula node.

In many cases, it is necessary to create a relation that contains all pairs of tuples (its formula is "True"), or that contains no pairs of tuples (its formula is "False".) It is possible to use the above constructors and member functions to create such relations. But since these relations are so common, we provide a shorthand way of creating them. The static member functions `Relation Relation::True` and `Relation Relation::False` return relations or sets with "True" and "False" respectively as their formulas. If one integer argument is provided then a set is returned this arity. If two integer arguments are provided then a relation is returned with these input and output arities respectively.

4.3 Referring to variables

An object of class `Var_Decl` represents all uses of a variable in a particular relation. `Var_Decl`s should never be created explicitly by application programs; they are created implicitly by member functions of various library classes. We will usually use `Variable_ID`s, which are pointers to `Var_Decl`s, to refer to such objects. To create constraints on a particular variable, it is necessary to determine the `Variable_ID` of that variable in the relation to which we want to add the constraints.
We now explain how to obtain `Variable_ID`s for various types of variables:

Input, output, and set variables

The `Relation` member functions `input_var(int n)`, `output_var(int n)`, and `set_var(int n)` produce `Variable_ID`s for the `n`th variable in the appropriate tuple. It is illegal request an input or output variable of a set, or to request a set variable from a relation.

Quantified Variables

The member function `Variable_ID F_Declaration::declare` creates a new `Var_Decl` and add it to the list of variables associated with the `F_Declaration` node that it is called on. It returns the `Variable_ID` for the newly created variable in the current relation. If a String is specified as an argument to the member function then that String is recorded as the name of that variable, otherwise the variable remains unnamed.

Scalar Global Variables

The variables we have seen so far have all been local to a single relation. Global variables, on the other hand, may be shared between relations. A global variable is created by creating an object of a class derived from `Global_Var_Decl`. The class `Free_Var_Decl` is an example of such a class, and is used for the global variables in the code in our examples (e.g., the creation of the global variables `n`

and `m` in Figure 4.1)). New subclasses of `Global_Var_Decl` can be created to keep track of any additional information that applications may want to record about global variables. The member function `Variable_ID Relation::get_local(const Global_Var_Decl *)` is used to return the `Variable_ID` of a scalar global variable in a particular relation. In Figure 4.1, the `Variable_ID Rs_n` is used to identify the global variable `n` in the relation `R`.

Global Variables Representing Uninterpreted Function Symbol
: As was the case for scalar global variables, global variables representing uninterpreted function symbol are created by creating objects of a class derived from `Global_Var_Decl`. Once again, class `Free_Var_Decl` is an example of such a class, but this time we need to use a `Free_Var_Decl` constructor that allows us to specify an arity, as shown by the creation of `f` in Figure 4.1. The enumeration `Argument_Tuple`, which contains the values `Input_Tuple`, `Output_Tuple`, and `Set_Tuple`, is used to specify the tuple to which an uninterpreted function symbol is to be applied. Since an uninterpreted function symbol can be applied to both the input and output tuple of a relation, two `Var_Decls` per relation may be necessary: one for the function applied to the input tuple and the other for the function applied to the output tuple. The member function `Relation::get_local(const Global_Var_Decl *, Argument_Tuple)`. is used to return the `Variable_ID` of one of these `Var_Decls` depending on which Argument_Tuple is requested. For example, we use this function to get the `Variable_ID`'s for `Rs_f_in` and `Rs_f_out` for the global variable `f` in Figure 4.1.

There are a few things you can find out about a variable given its `Variable_ID`:

`String Var_Decl::base_name`
: The name of the variable without primes.

`Var_Kind Var_Decl::kind()`
: Returns the kind of the variable; one of {`Input_Var`, `Output_Var`, `Set_Var`, `Global_Var`, `Forall_Var`, `Exists_Var`, `Wildcard_Var`} (a `Wildcard_Var` is equivalent to an existentially quantified variable, but appears only in simplified relations).

`int Var_Decl::get_position()`
: If the variable is an input, output, or set variable, returns its position in the tuple.

`Global_Var_ID Var_Decl::get_global_var()`
: If a variable corresponds to a global variable, returns its `Global_Variable_ID`.

`Argument_Tuple Var_Decl::function_of()`
: If a variable corresponds to a global variable, returns the argument to which it is being applied.

Note that there is no way to find out which relation a `Variable_ID` is associated with, and our implementation is free to either share `Variable_ID`s between relations or not (i.e., the result of `S1.set_var(1) == S2.set_var(1)` is not defined).

Figures 4.3 and 4.3 show many uses of formula building functions, as well as the creation of many equality and inequality constraints (see below). The creation of the variable `y` in the middle of Figure 4.3 shows the use of `F_Declaration::declare(Const_String)`.

4.4 Building atomic constraints

Atomic constraints come in three forms: equalities of the form $\sum a_i x_i + a_0 = 0$, inequalities of the form $\sum a_i x_i + a_0 \geq 0$, and stride constraints ($\sum a_i x_i + a_0$ is divisible by $step$).

Atomic constraints are manipulated using the `EQ_Handle`, `GEQ_Handle`, and `Stride_Handle` classes. These are derived from the class Constraint_Handle. They are referred to as "handles", because atomic constraints do not exist as independent objects. Instead, they are components of another class that is known only to the implementation; each `Constraint_Handle` contains information about how to access a particular constraint. The following functions can be used to add an atomic constraint to an `F_And`:

```
F_And *S1_root = S1.add_and();

GEQ_Handle tmin = S1_root->add_GEQ();   // t-1 >= 0
tmin.update_coef(t, 10);
tmin.update_coef(t, -9);       // t now has coef. 1
tmin.update_const(-1);
GEQ_Handle tmax = S1_root->add_GEQ();   // n-t >= 0
tmax.update_coef(S1s_n,1);
tmax.update_coef(t, -1);


F_Or  *S2_root = S2.add_or();
F_And *part1 = S2_root->add_and();

GEQ_Handle xmin = part1->add_GEQ();
xmin.update_coef(x,1);
GEQ_Handle xmax = part1->add_GEQ();
xmax.update_coef(x,-1);
xmax.update_const(100);

F_Exists *exists_y = part1->add_exists();
Variable_ID y = exists_y->declare("y");

F_And *y_stuff = exists_y->add_and();
GEQ_Handle ymin = y_stuff->add_GEQ();
ymin.update_coef(y,1);
ymin.update_coef(S2s_n,-2);
GEQ_Handle ymax = y_stuff->add_GEQ();
ymax.update_coef(x,1);
ymax.update_coef(y,-1);
Stride_Handle y_even = y_stuff->add_stride(2);
y_even.update_coef(y,1);
y_even.update_const(1);

F_And *part2 = S2_root->add_and();

EQ_Handle xvalue = part2->add_EQ();
xvalue.update_coef(x,1);
xvalue.update_const(-17);
```

Figure 4.2: Example, Part 2: Adding Constraints to $S1$ and $S2$

```
F_And *R_root = R.add_and();

GEQ_Handle imin = R_root->add_GEQ();
imin.update_coef(i,1);
imin.update_const(-1);
GEQ_Handle imax = R_root->add_GEQ();
imax.update_coef(i2,1);
imax.update_coef(i,-1);
GEQ_Handle i2max = R_root->add_GEQ();
i2max.update_coef(Rs_n,1);
i2max.update_coef(i2,-1);

EQ_Handle f_eq = R_root->add_not()->add_and()->add_EQ();
f_eq.update_coef(Rs_f_in,-1);
f_eq.update_coef(Rs_f_out,1);  // F(In) - F(Out) = 0

GEQ_Handle jmin = R_root->add_GEQ();
jmin.update_coef(j,1);
jmin.update_const(-1);
GEQ_Handle jmax = R_root->add_GEQ();
jmax.update_coef(Rs_m,1);
jmax.update_coef(j,-1);

GEQ_Handle j2min = R_root->add_GEQ();
j2min.update_coef(j2,1);
j2min.update_const(-1);
GEQ_Handle j2max = R_root->add_GEQ();
j2max.update_coef(Rs_m,1);
j2max.update_coef(j2,-1);
```

Figure 4.3: Example, Part 3: Adding Constraints to R

`EQ_Handle F_And::add_EQ()`
Creates an equality constraint as a child of the `F_And` node and returns a `EQ_Handle` for this constraint. All variables implicitly have a coefficient of zero in this constraint.

`GEQ_Handle F_And::add_GEQ()`
Creates an inequality constraint as a child of the `F_And` node and returns a `GEQ_Handle` for this constraint. All variables have an implicit coefficient of zero in this constraint.

`EQ_Handle F_And::add_stride(int step)`
The effect of this function is equivalent to creating an `F_Exists` node with a new variable alpha as a child of the `F_And` node and creating an equality constraint as a child of the `F_Exists` node. The coefficient of alpha in this constraint is `step` and the coefficients of all other variables is implicitly zero. This can be used to add constraints like "`y` is odd" or "`x+2y+17` is divisible by 3".

`void F_And::add_unknown()`
Adds an unknown constraint as a child of the `F_And` node, thus making the formula an upper bound (see Section 5.6). You don't need to use this function unless you are creating an inexact relation.

Sometimes you may have a relation (perhaps passed in to a function) that you want to modify by adding some constraints to it, but you don't have any pointers into the formula. In these situations, you can use the functions Relation::and_with_GEQ and Relation::and_with_EQ. The affect of calling these functions without arguments, on a relation R, with formula f, is equivalent to creating a constraint e of the appropriate type, changing R's formula to be an `F_And` node with f and e as children and returns a `Constraint_Handle` of the appropriate type for e. All variables have an implicit coefficient of zero in e. There's another version of this function that allows you to copy constraints between relations. If you call either function with a `Constraint_Handle` c as an argument, the affect is the same except that the coefficients of e are set to be the same as the coefficients of c. Notice that the latter form allows you to convert EQ's to GEQ's and vice-versa; the coefficients are copied, and the resulting constraint just has the semantics of whatever kind of constraint you asked for. It's a little confusing, but it saves you from copying each coefficient individually. The coefficients of variables and the constant terms in constraints can be updated by using the following member functions: functions:

`void Constraint_Handle::update_coef(Variable_ID, int delta)`
Add `delta` to the coefficient of the variable corresponding to `Variable_ID`.

`void Constraint_Handle::update_const(int delta)`
Add `delta` to the constant term of the constraint.

`void Constraint_Handle::multiply(int multiplier)`
Multiply each coefficient and the constant term by `multiplier`.

Warning: Remember that these functions add `delta` to the current value of a coefficient. There is no way to simply set a coefficient. This provides us with certain freedoms in our implementation, but it does make coding a bit trickier.

Setting numbers of EQs and GEQs

The library allows you to set the number of EQs and GEQs that are available in each problem. As this increases, the library can handle larger and larger problems, but memory use increases as well. If you need to solve large problems, you may have to increase these values. If you need to have many relations existing at the same time, you may need to decrease these values.

The variables to set are `maxGEQs` and `maxEQs`. They default to 70 and 35 respectively. If you set these variables, they **must** be set before you use the Omega Library. If you create any relations, and then change the value of these variables, you will get crashes.

4.5 Finalization

Many of our data structures can be *finalized* when they are complete. This is done via the various `finalize()` member functions. Adding new constraints to a structure that has been finalized is illegal. Explicit finalization allows the implementation to perform certain simplifications of constraints, which can improve efficiency in some cases. For example, we could provide an optimization that simply discards any constraints that are added to an `F_Or` that contains a tautology (this optimization is not currently in the library, but it is easy to explain). We can only be sure that one of the children is a tautology after it has been finalized: otherwise we would have to consider the possibility of the programmer adding something like $\wedge 1 = 2$ to this child, in which case it is no longer a tautology. Since finalization does not affect the relation, we have not shown its use in our examples. However, finalization is important to achieving maximum efficiency. If you are concerned with efficiency, you should finalize each part of a relation as soon as you have finished building it.

Chapter 5

Querying Existing Relations

This chapter outlines some ways to examine, simplify and query relations that you have built.

5.1 Relations during and after queries

Relations may be simplified to respond to a query, so you may see some changes in a relation if you print it before and after a query. For example, if you build a relation, print it with `Relation::print()`, then ask if it is empty, and then print it again, the formula in the second print will be different from but equivalent to the first one. `Relation::print_with_subs()` would show no difference between those two calls, since it copies the relation and simplifies it in order to print it.

So, once you begin to query a relation, the Presburger formula may change unexpectedly. Since the structure of the formula changes, there are restrictions on the operations you can perform. Specifically, most of the relation building operations are no longer available (in other words, it is implicitly finalized). You can no longer add constraints to `F_And` nodes, and you can no longer add new `Formula` nodes to any part of the formula. The only way to modify the Presburger formula after beginning to query it is to use `Relation::and_with_EQ` or `Relation::and_with_GEQ`.

If you wish to force a simplification of a relation for some reason, use the function `Relation::simplify()`. This is a hint to the library that it might be a good idea to simplify the relation immediately. You might want to do this on a relation that has a complicated formula, before you combine it with others using the functions in Chapter 6. Sometimes, this can speed up execution or reduce memory requirements.

5.2 Printing

The simplest way to examine a relation is to print it to the screen or to a file. The following member functions of `Relation` print relations in a number of different ways.

```
void Relation::print()
```

```
void Relation::print(FILE *output_file)
```
A basic function to print the relation for humans to read.

```
void Relation::print_with_subs(FILE *output_file, bool printSym=false)
```

```
String Relation::print_with_subs_to_string(bool printSym=false)
```
These functions attempt to print the relation in an easy-to-understand format. At each input variable and output variable, they try to print the variable as an affine function of the variables to the left. The variable printSym controls whether the set of symbolic variables used in the relation are printed.

```
void Relation::prefix_print(FILE *output_file)
```
This is a print function used primarily to debug programs that use the library. It is designed to make clear the structure of the formula tree and show the details of the variables used in the formula.

`void Relation::prefix_print()`
This is the same as the other version of prefix_print, but it prints to stdout.

`String Relation::print_formula_to_string()`
This allows you to extract a printed representation of the relation's formula, without the input and output variables being printed.

5.3 Simplification and satisfiability

A basic function of the library is checking whether a relation is empty (its Presburger formula is unsatisfiable), or whether a relation contains all possible tuples (its Presburger formula is a tautology).

In some cases, due to inexact formulas, it is not possible to answer these questions definitively; we need to describe them in terms of upper and lower bounds on the relation. The upper bound is computed by assuming all UNKNOWN constraints evaluate to true. The lower bound is computed by assuming all UNKNOWN constraints evaluate to false. (Functions to compute these bounds are described in section 6.2.)

The following member functions to check these conditions.

- `bool Relation::is_upper_bound_satisfiable()`
  Return true if the relation's Upper_Bound is satisfiable; that is, treat UNKNOWN constraints as true, then check satisfiability.
- `bool Relation::is_lower_bound_satisfiable()`
  Return true if the relation's Lower_Bound is satisfiable; that is, treat UNKNOWN constraints as false, then check satisfiability.
- `bool Relation::is_satisfiable()`
  Included for compatibility with older releases. Asserts that the results of `is_upper_bound_satisfiable` and `is_lower_bound_satisfiable` are equal, then returns the result.
- `bool Relation::is_obvious_tautology()`
  Return true if the relation's formula evaluated to a single conjunction with no constraints. Some tautologies will not fit this description.
- `bool Relation::is_definite_tautology()`
  Returns true if the relation's formula is a tautology.

The following functions allow you to inquire about inexactness in the relation:

- `bool Relation::is_exact()`
  Returns true if the relation's formula does not contain UNKNOWN.
- `bool Relation::is_inexact()`
  Returns true if the relation's formula contains UNKNOWN.
- `bool Relation::is_unknown()`
  Returns true if the relation's formula is the single constraint UNKNOWN.

Figure 5.3 shows some uses of these functions on the sets and relation we built in the previous chapter.

5.4 Querying variables

The function `Relation::global_decls()` returns a pointer to the collection of `Variable_ID`s that contains all the uses of global variables in the relation. This collection may include some global variables that have been eliminated during simplification.

The function `Relation::query_difference(Variable_ID v1, Variable_ID v2, int &lowerBound, int &upperBound, bool &guaranteed)` can be used to determine bounds on the possible value of `v1 -`

```
S1.print_with_subs(stdout);
assert(S1.is_upper_bound_satisfiable());
assert(!S1.is_tautology());
S1.print_with_subs(stdout);  // same as above print
printf("\n");

S2.print();
assert(S2.is_upper_bound_satisfiable());
assert(!S2.is_tautology());
S2.print();                 // different from above
printf("\n");

assert(R.is_upper_bound_satisfiable());
assert(!R.is_tautology());
R.print_with_subs(stdout);

int lb, ub;
bool coupled;
R.query_difference(i2, i, lb, ub, coupled);
assert(lb == 1);        // i < i2: i2 - i1 > 0
assert(ub == posInfinity);

for(DNF_Iterator di(R.query_DNF()); di; di++)
    {
    printf("In next conjunct,\n");
    for(EQ_Iterator ei = (*di)->EQs(); ei; ei++)
        {
        printf("  In next equality constraint,\n");
        for(Constr_Vars_Iter cvi(*ei); cvi; cvi++)
            printf("    Variable \%s has coefficient \%d\n",
                   (*cvi).var->char_name(),
                   (*cvi).coef);
        }
    for(GEQ_Iterator gi = (*di)->GEQs(); gi; gi++)
        {
        printf("  In next inequality constraint,\n");
        for(Constr_Vars_Iter cvi(*gi); cvi; cvi++)
            printf("    Variable \%s has coefficient \%d\n",
                   (*cvi).var->char_name(),
                   (*cvi).coef);
        }
    printf("\n");
    }
```

Figure 5.1: Example, Part 4: Querying Relations

v2. Note that these bounds are not necessarily tight. After this call, guaranteed will be true if these bounds are guaranteed to be tight. If the difference is not bounded below, lowerBound will be negInfinity. If the difference is not bounded above, upperBound will be posInfinity. These constants are defined in "oc.h", which is included by "omega.h".

5.5 Iterating through a DNF

Much of the power of the Omega library lies in the ability to walk through the simplified conjunctions (and their constraints) that define a relation. In this section, we'll describe method to do that.

To perform more detailed queries on a relation, we must tell the Omega Library the desired form of the results, and the amount of effort (time) that should be expended in trying to eliminate redundancies. Currently, the only form of query that is supported is disjunctive normal form, though in the future we may include disjoint disjunctive normal form, or a form that contains only inequality constraints.

To request that a relation be simplified to DNF, use the function query_DNF. This returns a pointer to a DNF (we discuss how to use a DNF below). Several levels of simplification are available. Increased effort levels result in better elimination of redundant information in the formula. You can set this level for both removing redundant constraints within a conjunction, or for removing entire conjuncts that are redundant with some other part of the formula. These levels are given as arguments to query_DNF. If no arguments are given, the levels default to (0,0), the lowest level of effort. The current effort levels are given below:

	Conjuncts	Constraints
0	Nothing extra	Nothing extra
1	Perform simple check for redundant conjunts	Remove constraints made redundant by any two others
2	Exact test to see if any conjunct is subset of any other	Exact test to see if any constraint is redundant
4		Also perform expensive tests to simplify form of constraints

The best way to traverse the formula is by using a series of iterator classes. The overall method to traversing a simplified formula is:

```
for each conjunct in the DNF
    for each equality constraint in the conjunct
        for each variable in the constraint
            perform some action
    for each GEQ constraint in the conjunct
        for each variable in the constraint
            perform some action
```

Most often, you will use the DNF * returned from query_DNF to initialize a DNF_Iterator, which allows you to traverse a list of Conjuncts, which represent the individual conjunctions in the DNF. Note that if you change the relation destructively after obtaining a simplified DNF from it, neither the result of a previous query_DNF call nor any iterators constructed from it will remain valid.

Examining constraints in a Conjunct

You can iterate through the EQs or the GEQs in a Conjunct, or both. To browse the EQs, either give the Conjunct as an argument to the EQ_Iterator constructor, or assign the result of the Conjunct::EQs() function to an existing EQ_Iterator. When you dereference an EQ_Iterator, it returns an EQ_Handle. The analogous techniques apply to GEQ_Iterators, to iterate through the GEQs, or to Constraint_Iterators, to iterate through both the EQs and the GEQs.

We have already seen the use of Constraint_Handles (and its subtypes EQ_Handle and GEQ_Handle) to build constraints. In this context, a Constraint_Handle (and its derived types) is used only to examine constraints; you cannot modify the constraints returned from any constraint iterator. The following operations are used to query a constraint.

- int Constraint_Handle::get_coef(Variable_ID v)

- `int Constraint_Handle::get_const()`
- `bool Constraint_Handle::has_wildcards()`

Here is an example of examining the constraints in a `Relation`. This loop finds all the GEQs involving `v` in all `Conjunct C` from `Relation R1`, and "ands" them with `Relation R2`'s formula. The only caveat to doing this is that `R2` must have at least as many variables in its tuples as `R1` does.

```
for(DNF_iterator di(R1.query_DNF()); di; di++)
    for(EQ_Iterator ei = (*di)->EQs(); ei; ei++)
        if((*ei).get_coef(v) != 0)
            R2.and_with_GEQ(*ei);
```

(Note: this may not be as useful as it looks, if the conjunct from `R1` contains existentially quantified variables (which, since `R1` is simplified, will actually be of the tyep `Wildcard_Var`). In that case, since each constraint is copied individually, the correspondence between two uses of the same quantified varaible will be lost.)

You could also loop through all the variables in each conjunct and check their coefficients in each constraint:

You can also loop through all the variables in the conjunct and check their coefficients in each constraint;

```
for(DNF_iterator di(R1.query_DNF()); di; di++)
    for(EQ_Iterator ei = (*di)->EQs(); ei; ei++)
        for(Variable_Iterator vi = (*di)->variables(); vi; vi++)
```

Since this is a common thing to do, there's a special iterator class that allows you to walk through all the variables that appear in a particular constraint, and get their coefficients. It's called Constr_Vars_Iter. There's also a version of it that will show you only the existentially quantified variables. (An easy way to check if a constraint involves any existentially quantified variables is to get that kind of iterator and check if it is live.)

For a Constr_Vars_Iter, the return value is a `Variable_Info` structure. It contains a Variable_ID and an integer for its coefficient in the constraint. You can also use the special functions `Variable_ID curr_var()` and `int curr_coef()` to get that information.

The **for** loop in Figure 5.3 shows an example of the use of these query functions that is redundant with the existing printing functions, but serves to illustrate the query operations.

5.6 Inexact relations

The Omega Library provides a way to work with inexact relations. We say that relation is inexact if one or more of its conjuncts contain UNKNOWN constraints. We call any conjunct that contains unknown constraints an inexact conjunct, and the conjunct that contains only unknown constraints an unknown conjunct. To inquire about this property of the conjunct you can use the `is_exact()`, `is_inexact()` and `is_unknown()` member functions of the Conjunct class.

An exact part of the relation containing an unknown conjunct provides a lower bound on the relation. An exact part of the relation containing inexact conjunct(s) provides an upper bound on the relation. Note, that an inexact conjunct can be always approximated by the unknown conjunct so a simplified relation can not contain both inexact and unknown conjuncts. Thus, a relation can be either exact, a lower bound, or an upper bound. We refer to this property of the relation as its accuracy status. Accuracy status is described by the enumerated type `Rel_Accuracy_Status`. To inquire about the accuracy status of the relation you can use the `get_accuracy_status()` member functions of the Relation class.

Inexact relations will be produced when the result of a relational operation is outside the class of relations we can represent. Recall that the Omega Library cannot handle relations that contain the application of a function to something other than the input, output, or set tuple. So if it is asked for the range of $\{[i] \rightarrow [i'] \mid 1 \leq i < i' \leq 10 \wedge F(i) > 0\}$, it produces an approximate answer.

You also can create the inexact relations explicitly by one of the following ways:

- adding an unknown constraint to the formula (see section 4.4)
- unioning or intersecting with unknown relation (see section 6.3). Namely, given that r is an exact relation and U is an unknown relation, r is a lower bound on r `union` U and is an upper bound on r `intersection` U.

You can use the functions `Upper_Bound` and `Lower_Bound` to produce exact relations from from the relation; they are described in section 6.2.

Chapter 6

Creating New Relations From Old

This section contains descriptions of the high-level operations on relations. Most of these operations take sets and/or relations as arguments and produce sets and or relations as results. In the following examples, x, x_1, x_2, y, y_1, y_2 and z are arbitrary tuples and f_1 and f_2 are arbitrary presburger formulas.

6.1 Important warning

The most important thing to remember when using these operations is that they destroy their arguments. If a relation Y is passed as an argument to a relational operation such as X = Union(Y, Z) then the value of Y (and Z) are set to Null relations after the operation. These is because it if often efficient to "steal" data structures associated with the arguments in building the result. If a relation to be passed as an argument to a relational operation needs to be used after that operation, the relation should be copied explicitly: either by creating another relation to pass in, or by using the `Relation::copy(Relation &)` function in the function call (as in X = `Intersection(X, copy(Y))`.)

Figure 6.1 shows the use of some of the relation operations with the sets and relation created in earlier figures.

6.2 Upper and Lower bounds

The following functions are used to produce upper and lower bounds on inexact relations. When applied to an exact relation, the return value is the original relation.

`Relation Upper_Bound(Relation &r)`
: Works for both sets and relations. Returns s such that $r \subseteq s$ and s is exact. Works by interpreting all UNKNOWN constraints as true.

`Relation Lower_Bound(Relation &r)`
: Works for both sets and relations. Returns s such that $s \subseteq r$ and s is exact. Works by interpreting all UNKNOWN constraints as false.

6.3 Binary relational operations

`Relation Union(Relation &`r_1`, Relation &`r_2`)`
: Works for both relations and sets. The arguments must have the same arity. If $r_1 = \{ x_1 \rightarrow y_1 \mid f_1(x_1, y_1) \}$ and $r_2 = \{ x_2 \rightarrow y_2 \mid f_2(x_2, y_2) \}$ then the result is $\{ x \rightarrow y \mid f_1(x, y) \vee f_2(x, y) \}$.

`Relation Intersection(Relation &`r_1`, Relation &`r_2`)`
: Works for both relations and sets. The arguments must have the same arity. If $r_1 = \{ x_1 \rightarrow y_1 \mid f_1(x_1, y_1) \}$ and $r_2 = \{ x_2 \rightarrow y_2 \mid f_2(x_2, y_2) \}$ then the result is $\{ x \rightarrow y \mid f_1(x, y) \wedge f_2(x, y) \}$.

```
Relation S1_or_S2 = Union(copy(S1), copy(S2));

// NOTE! THE FOLLOWING KILLS S1 AND S2
Relation S1_and_S2 = Intersection(S1, S2);

S1_or_S2.is_upper_bound_satisfiable();
S1_and_S2.is_upper_bound_satisfiable();

S1_or_S2.print();
printf("\n");

S1_and_S2.print();
printf("\n");

Relation R_R = Composition(copy(R), R);
R_R.query_difference(i2, i, lb, ub, coupled);
assert(lb == 2);
assert(ub == posInfinity);
}
```

Figure 6.1: Example, Part 5: Working with Relations

`Relation Composition(Relation &r1, Relation &r2)`
Works for relations only. The output arity of r_2 must be same as the input arity of r_1. If $r_1 = \{ x_1 \rightarrow y_1 \mid f_1(x_1, y_1) \}$ and $r_2 = \{ x_2 \rightarrow y_2 \mid f_2(x_2, y_2) \}$ then the result is $\{ x \rightarrow y \mid \exists z \text{ s.t. } f_1(z, y) \wedge f_2(x, z) \}$.

`Relation Join(Relation &r1, Relation &r2)`
Equivalent to `Composition(r2, r1)`.

`Relation Restrict_Domain(Relation &r1, Relation &r2)`
First argument must be a relation and second argument must be a set. The input arity of r_1 must be the same as the arity of r_2. If $r_1 = \{ x_1 \rightarrow y_1 \mid f_1(x_1, y_1) \}$ and $r_2 = \{ x_2 \mid f_2(x_2) \}$ then the result is $\{ x \rightarrow y \mid f_1(x, y) \wedge f_2(x) \}$.

`Relation Restrict_Range(Relation &r1, Relation &r2)`
First argument must be a relation and second argument must be a set. The output arity of r_1 must be the same as the arity of r_2. If $r_1 = \{ x_1 \rightarrow y_1 \mid f_1(x_1, y_1) \}$ and $r_2 = \{ x_2 \mid f_2(x_2) \}$ then the result is $\{ x \rightarrow y \mid f_1(x, y) \wedge f_2(y) \}$.

`Relation Difference(Relation &r1, Relation &r2)`
Works for both relations and sets. The arguments must have the same arity. If $r_1 = \{ x_1 \rightarrow y_1 \mid f_1(x_1, y_1) \}$ and $r_2 = \{ x_2 \rightarrow y_2 \mid f_2(x_2, y_2) \}$ then the result is $\{ x \rightarrow y \mid f_1(x, y) \wedge \neg f_2(x, y) \}$.

`Relation Cross_Product(Relation &r1, Relation &r2)`
Works for sets only. If $r_1 = \{ x_1 \mid f_1(x_1) \}$ and $r_2 = \{ x_2 \mid f_2(x_2) \}$ then the result is $\{ x \rightarrow y \mid f_1(x) \wedge f_2(y) \}$.

`Relation Gist(Relation &r1, Relation &r2, int effort=0)`
Works for both relations and sets. The arguments must have the same arity. If $r_1 = \{ x_1 \rightarrow y_1 \mid f_1(x_1, y_1) \}$ and $r_2 = \{ x_2 \rightarrow y_2 \mid f_2(x_2, y_2) \}$ then the result is $\{ x \rightarrow y \mid f(x, y) \}$ where f is a presburger formula such that $\forall x, y \; f(x, y) \wedge f_2(x, y) \Leftrightarrow f_1(x, y)$ Note: there are many f's that satisfy this property, so the results of this operation are not completely specified. We did this deliberately as our method for computing f may change in future releases. The effort parameter specifies how hard we try to make f tight, 0 represents least effort and 2 represents most effort.

6.4 Unary relational operations

`Relation Transitive_Closure(Relation &r, Relation & IterationSpace=Relation::Null())`

Works for relations only. The input and output arity of r and arity of *IterationSpace* should be the same. If r is a dependence relation, *IterationSpace* is a set describing iteration space of r. Parameter *IterationSpace* can be omitted. In this case the techniques requiring information about it are not employed.

Computing transitive closure exactly for all relations is undecidable, since $\{[x,y] \rightarrow [x+1, y+n]\}^+ = \{[x,y] \rightarrow [x+z, y+z*n] : 1 \leq z\}$ encodes multiplication; Presburger plus multiplication is undecidable. In cases where we cannot compute the exact transitive closure, we compute a lower bound. Details can be found elsewhere [KPRS95].

`Relation Domain(Relation &r)`

Works for relations only. If $r = \{ x \rightarrow y \mid f(x,y) \}$ then the result is $\{ x \mid \exists y \text{ s.t. } f(x,y) \}$

`Relation Range(Relation &r)`

Works for relations only. If $r = \{ x \rightarrow y \mid f(x,y) \}$ then the result is $\{ y \mid \exists x \text{ s.t. } f(x,y) \}$

`Relation Inverse(Relation &r)`

Works for relations only. If $r = \{ x \rightarrow y \mid f(x,y) \}$ then the result is $\{ y \rightarrow x \mid f(x,y) \}$

`Relation Complement(Relation &r)`

Works for both relations and sets. If $r = \{ x \rightarrow y \mid f(x,y) \}$ then the result is $\{ x \rightarrow y \mid \neg f(x,y) \}$

`Relation Project(Relation &r, Global_Var_ID v)`

Works with both relations and sets. If $r = \{ x_1 \rightarrow y_1 \mid f'(x_1, y_1) \}$ then the result is
$\{ x_1 \rightarrow y_1 \mid \exists z \text{ s.t. } f(x_1, y_1) \}$, where f is the same as f′ except that all occurrences of variable v are replaced by variable z.

`Relation Project(Relation &r, int pos, Var_Type vtype)`

Works with both relations and sets. If $r = \{ [a_1, .., a_m] \rightarrow [b_1, .., b_n] \mid f_1(a_1, .., a_m, b_1, .., b_n) \}$ then the result is
$\{ [a_1, .., a_m] \rightarrow [b_1, .., b_n] \mid \exists z \text{ s.t. } f_1(a'_1, .., a'_m, b'_1, .., b'_n) \}$, where $\forall i\ a'_i = a_i \wedge b'_i = b_i$ except that $a'_{pos} = z$ if vtype is Input_Var and $b'_{pos} = z$ if vtype = Output_Var.

`Relation Project_Sym(Relation &r)`

Works with both relations and sets. If $r = \{ x_1 \rightarrow y_1 \mid f'(x_1, y_1) \}$ then the result is $\{ x_1 \rightarrow y_1 \mid \exists a_1, .., a_n \text{ s.t. } f(x_1, y_1) \}$, where f is the same as f′ except that all occurrences of each global variable g_j are replaced by variable a_j.

`Relation Project_On_Sym(Relation &r)`

Works with both relations and sets. If $r = \{ x \rightarrow y \mid f(x,y) \}$ then the result is $\{ [\,] \mid \exists x, y \text{ s.t. } f(x, y) \}$.

`Relation Extend_Domain(Relation &r)`

Works with relations only. If $r = [a_1, .., a_m] \rightarrow [b_1, .., b_n] \mid f(a_1, .., a_m, b_1, .., b_n) \}$ then the result is $\{ [a_1, .., a_m, a_{m+1}] \rightarrow [b_1, .., b_n] \mid f(a_1, .., a_m, b_1, .., b_n) \}$,

`Relation Extend_Domain(Relation &r, int n)`

Equivalent of applying the simple Extend_Domain function n times.

`Relation Extend_Range(Relation &r)`

Works with relations only. If $r = \{ [a_1, .., a_m] \rightarrow [b_1, .., b_n] \mid f(a_1, .., a_m, b_1, .., b_n) \}$ then the result is $\{ [a_1, .., a_m] \rightarrow [b_1, .., b_n, b_{n+1}] \mid f(a_1, .., a_m, b_1, .., b_n) \}$,

`Relation Extend_Range(Relation &r, int n)`

Equivalent of applying the simple Extend_Range function n times.

`Relation Extend_Set(Relation &r)`
Works with sets only. If $r = \{ [a_1, .., a_m] \mid f(a_1, .., a_m) \}$ then the result is $\{ [a_1, .., a_m, a_{m+1}] \mid f(a_1, .., a_m) \}$,

`Relation Extend_Set(Relation &r, int n)`
Equivalent of applying the simple Extend_Set function n times.

`Relation Deltas(Relation &r)`
Works for relations only. The input arity must be the same as the output arity. If $r = \{ x_1 \rightarrow y_1 \mid f(x_1, y_1) \}$ then the result is $\{ z \mid \exists x, y \text{ s.t. } f(x, y) \wedge z = y - x \}$

`Relation Deltas(Relation &r, int p)`
Works with relations only. If $r = \{ [a_1, .., a_m] \rightarrow [b_1, .., b_n] \mid f_1(a_1, .., a_m, b_1, .., b_n) \}$ then it must be the case that $p \leq m \wedge p \leq n$ and the result is $\{ [c_1, .., c_p] \mid f_1(a_1, .., a_m, b_1, .., b_n) \wedge \forall j \ 1 \leq j \leq p, \ c_j = b_j - a_j \}$

`Relation Approximate(Relation &r)`
Works for both relations and sets. If $r = \{ x_1 \rightarrow y_1 \mid f'(x_1, y_1) \}$ then the result is $\{ x \rightarrow y \mid f(x, y) \}$, where f is the same as f′ except that all quantified variables are henceforth designated as being able to have non-integer rational values. The effect of this designation is that all of these variables will be able to eliminated exactly (via Fourier variable elimination) when the formula is simplified.

`Relation Approximate(Relation &r, int strides_allowed)`
Works for both relations and sets. If strides_allowed is False then the result is the same as the simple Approximate function. If strides_allowed is True then the result is the same as the simple Approximate function except that if a quantified variable is only involved in one constraint then it is not designated as being able to have non-integer rational values. The effect of this function is that the only variables that can't be eliminated exactly, are those that correspond to simple stride constraints.

`Relation EQs_to_GEQs(Relation &r, bool excludeStrides=false)`
Works for both relations and sets. If $r = \{ x_1 \rightarrow y_1 \mid f(x_1, y_1) \}$ then the result is $\{ x \rightarrow y \mid f'(x, y) \}$, where f′ is the same as f except that all equality constraints are converted into a matching pair of inequalities. If excludeStrides is True then this convertion process is not applied to those equality constraints that correspond to stride constraints (i.e. those constraints involving a single quantified variable).

`Relation Sample_Solution(Relation &r)`
For a relation R, returns a relation `S` $\subseteq$ `R` where each variable in S has exactly one value. If R is inexact, the result may be inexact.

`Relation Symbolic_Solution(Relation &r)`
For a relation R, returns a relation `S` $\subseteq$ `R` where each input, output, or set variable in S has exactly one value, plus constraints on the symbolic variables.

6.5 Advanced operations (Hulls, Farkas lemma, ...)

The following operations are a little more sophisticated than Presburger arithemitic. A good place to find more information about the mathematical basis of these operations is Schrijver's *Theory of Linear and Integer Programming* [Sch86]. First, some nice, mathematical definitions. Note that in each of the following, there is set value for t or fixed set of x_i's. Rather, if you can find a value of t and set of x_i's that would make a point be part of the result, then it is part of the result.

Linear Hull of X = $\{\sum_{i_1}^{t} lambda_i x_i \mid x_i \in X\}$

Affine Hull of X = $\{\sum_{i_1}^{t} lambda_i x_i \mid x_i \in X \wedge 1 = \sum_{i=1}^{t} lambda_i\}$

Conic Hull of X = $\{\sum_{i_1}^{t} lambda_i x_i \mid x_i \in X \wedge \forall_{i=1}^{t} \lambda_i \geq 0\}$

Convex Hull of X = $\{\sum_{i_1}^{t} lambda_i x_i \mid x_i \in X \wedge \forall_{i=1}^{t} \lambda_i \geq 0 \wedge 1 = \sum_{i=1}^{t} lambda_i\}$

More intuitive definitions:

Convex Hull of X is the tightest set of inequality constraints who's intersection contains all of X

Affine Hull of X is the tightest set of equality constraints who's intersection contains all of X (i.e., all of the equality constraints from the convex hull)

Conic Hull of X is the tightest set of inequality constraints with a zero constrant term who's intersection that contains all of X. The means that the conic hull is a cone, starting at the origin, that includes all of X.

Linear Hull of X is the tightest set of equality constraints with a zero constrant term who's intersection contains all of X. The linear hull is a hyperplane passing through the origin and including all of X.

These operations all work by applying appropriate versions of Farkas's lemma twice. *Warning:* These operations can all be computationally expensive, and might result in either generating a *very* large number of constraints, or *very* large coefficients (either of which will currently raise an exception). We are beinging to experiment with trying to use these operations in a way that won't blow out of control, but our work on this is only just beginning. Of the operations above, convex hull is the most difficult and linear hull is the easiest to compute.

We provide several other operations which don't have as clean a mathematical definition but are computationally more tractiable. Many of these also allow a context to be provided: a content is a known, convex, upper bound on the region.

Decoupled Convex Hull - This version of the convex hull operation only generates a constraint involving both variables x and y if their is a one constraint involving both x and y in the argument. The solution is always contains the exact convex hull and other than as required by these restrictions, is as tight as possible.

For example, the convex hull of $\{[1:10, 1:10]\} \cup \{[11:20, 11:20]\}$ is $\{[x, y] : 1 \leq x, y \leq 20 \wedge -9 \leq x - y \leq 9\}$. But the decoupled convex hull would simply be $\{[1:20, 1:20]\}$.

FastTightHull - This function tries to use the convex hull computation above. However, if that computation appears that it might be too complex, it uses a more approximate calculation, such as Decoupled Convex Hull or Linear Hull. It is possible that it won't correctly estimate the difficulty of a computation, and either approximate something that is, in fact easy, or tackle something that is too difficult to compute (generating a fatal error).

QuickHull - This function takes a tuple of relations, each of which is a single conjunct relation. A constraint is in the result of QuickHull only if it appears in one of the relations and is directly implied by a single constraint in each of the other relations (which must be a parallel constraint).

Hull - Combines Quick Hull, methods that check to see if any constraint appearing in one conjunct in fact bounds all the conjuncts, and FastTightHull. This function is intended to give the best, overall balance of accuracy and efficiency.

6.6 Low level relational operations

This section describes two low level relational operations that are used to implement many of the above operations. The use of these low level functions is discouraged in cases where some combination of the higher level operations can be used instead.

These operations take as arguments mappings which specify for each input and output variable in the input relation(s), the input, output or existentially quantified variable in the resultant relation that they correspond to. For example $map(Input_Var, 1) = (Output_Var, 3)$ specifies all occurrences of the 1^{st} input variable will be replaced by the 3^{rd} output variable in the resultant relation.

`void MapRel1(Relation &r, Mapping &map, Combine_Type ctype, int n_inp, int n_out)`
Works with both relations and sets (see discussion below for sets).

`Relation MapAndCombineRel2(Relation &r1, Relation &r2, Mapping &map1, Mapping &map2, Combine_Type Op, int n_inp, int n_out)`
Works with both relations and sets (see discussion below for sets).

MapRel1 and MapAndCombineRel2 can also take as arguments and produce as results sets. To do so, the specified mappings must refer to Set_Vars rather than Input_Vars and Output_Vars. It is important not to mix Set_Vars with Input_Vars or Output_Vars in the same object.

Mappings are declared using a constructor that takes two integer arguments that represent the input and output arities of the input relation. Mappings are specified using the member function `void set_map(Var_Type in_type, int i, Var_Type out_type, int j)`. For example the example above would be specified by `map.set_map(Input_Var,1,Output_Var,3)`.

6.7 Relational functions that return boolean values

`bool Must_Be_Subset(Relation &r1, Relation &r2)`
Works for both relations and sets. The arguments must have the same arity. Let $s_1 = \text{Upper_Bound}(r_1)$ and $s_2 = \text{Lower_Bound}(r_2)$. If $s_1 = \{ x_1 \rightarrow y_1 \mid f_1(x_1, y_1) \}$ and $s_2 = \{ x_2 \rightarrow y_2 \mid f_2(x_2, y_2) \}$ then the result is $\forall x, y \; f_1(x, y) \Rightarrow f_2(x, y)$.

`bool Might_Be_Subset(Relation &r1, Relation &r2)`
Works for both relations and sets. The arguments must have the same arity. Let $s_1 = \text{Lower_Bound}(r_1)$ and $s_2 = \text{Upper_Bound}(r_2)$. If $s_1 = \{ x_1 \rightarrow y_1 \mid f_1(x_1, y_1) \}$ and $s_2 = \{ x_2 \rightarrow y_2 \mid f_2(x_2, y_2) \}$ then the result is $\forall x, y \; f_1(x, y) \Rightarrow f_2(x, y)$.

`bool Is_Obvious_Subset(Relation &r1, Relation &r2)`
Works for both relations and sets. The arguments must have the same arity. If $r_1 = \{ x_1 \rightarrow y_1 \mid f_1(x_1, y_1) \}$ and $r_2 = \{ x_2 \rightarrow y_2 \mid f_2(x_2, y_2) \}$ then the result is B, where $B \rightarrow (\forall x, y \; f_1(x, y) \Rightarrow f_2(x, y))$. This function will often return true in cases where r_1 is a subset of r_2, but it is not guaranteed to. The situations in which Is_Obvious_Subset agrees with exact subset are not specified, as they may change in future releases.

6.8 Generating code from relations

The library provides code generation functions that can produce loop nests for a set of statements, each with its own iteration space. The algorithm allows the union of these iteration spaces to be non-convex, and performs optimizations to reduce the level of overhead caused by iterating over a non-convex region. Since the overhead-reduction process increases code size, the user may specify how much overhead is to be removed. The algorithm is described elsewhere [KPR95].

To use the following functions, include the file `code_gen/code_gen.h` in your program.

These two functions return a string containing C pseudo-code:

- `String MMGenerateCode(Tuple<Relation> & transformations, Tuple<Relation> &original_IS, Relation known, int effort=0);`
- `String MMGenerateCode(Tuple<Relation> & transformations, Tuple<Relation> & original_IS, Tuple<naming_info *> & name_func_tuple, Relation known, int effort=0);`

The interface to the function uses several `Tuple`s, with the first position in each tuple for the first statement, the second position in each for the second statement, and so on. For each statement, you must provide:

- A set, the original iteration space for each statement
- A relation, a transformation for each statement to the new iteration space
- Optionally, a an object derived from `class naming_info` to provide a name for each statement

The transformation is an affine, 1-1 mapping from points in the statement's original iteration space to the new iteration space. The output arity of the transformation relations is the dimensionality of the resulting iteration space, and all transformations must have the same output arity. The input arity of each transformation must be the same as the number of set variables in the corresponding iteration space. If no transformation is desired, specify the identity relation for the transformation.

If symbolic constants are used in the iteration spaces, you can provide any information you have about their values in the set `known`. It must have the same number of set variables as the dimensions of the resulting iteration space. This information can allow us to generate simpler code.

The algorithm can use different levels of effort to remove control overhead, at the cost of code duplication. The `effort` parameter is an integer that specifies that all avoidable control overhead should be removed from `effort + 1` inner loops. By default, overhead is removed from the innermost loop (effort of 0). For a five dimensional iteration space, an effort level of 1 says that there should be no overheads nested inside four loops; an effort level of 2 means no overheads at nesting level 3 or greater; and so on. An effort of -1 means omit all overhead removal.

To specify names for the statements, you should create a class derived from `class naming_info`, and pass into the code generation routines a `Tuple` of pointers to that class. The class needs to provide the following functions:

`naming_info()`
: A default constructor.

`virtual naming_info& operator=(const naming_info &n2)`
: The assignment operator.

`virtual String name(Relation *current_map)`
: The function to provide a `String` representation of a statement, given the current arguments. See below for a full description.

`virtual String debug_name()`
: A `String` to represent the statement in debugging output.

`virtual char * debug_char_name()`
: A `char*` to represent the statement in debugging output. (This is necessary because of the lifetime of temporaries in C++; in a statement like `printf("%s",(char *) n->name())`, the cast to `char*` is considered a use of the temporary `String`, which is then deleted.)

`virtual String declaration()`
: Returns a `String` representation of the C declarations for the statement, if any are required. This is currently unused.

The `Relation *current_map` argument to the `naming_info::name` function is a mapping from the index variables of the generated loops to the corresponding original index variables. The new index variables have a different iteration space than any of the original statements, called the "transformed iteration space"; thus the `current_map` relation maps an iteration in the transformed iteration space to an iteration in the statement's original iteration space. Calling the function `Relation::print_outputs_with_subs_to_string()` on `current_map` will return a comma-delimited list of expressions for calculating the original iteration.

6.9 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 function `Reachable_Nodes` performs this function; to use it, include `omega/reach.h`. It takes a single argument of type `reachable_info`, defined as follows:

```
class reachable_information {
public:
Tuple<String> node_names;
Tuple<int> node_arity;
Dynamic_Array1<Relation> start_nodes;
Dynamic_Array2<Relation> transitions;
};
```

For a graph of n nodes, the tuples must have n elements, and the `Dynamic_Array`s must have $n + 1$ elements; the zeroth element in the `Dynamic_Array`s is unused.

The current implementation is very straightforward and can be very slow. It computes the transitive closure of the transitions, then applies the resulting relations to the start states.

6.10 Avoiding copy overhead

When the relations have large formulas, copying and assigning relations can be costly. In some instances, such as when you return a relation from a function, or when you copy a locally-declared relation into a list or array, the relation being copied may be thrown away after the operation.

The library has a mechanism to make these copies more efficient. It does reference counting of relations in some limited cases, so that no copy is actually made; each relation shares the same "body". In order to get this behavior, call the function `Relation::finalize()` on your relation. This essentially marks the relation as "read-only" as far as the library is concerned. That means that there are no constraints being built, and no more formula nodes can be added anywhere in its formula tree. You'll still be able to perform high-level operations on it, and you'll still be able to query it and simplify it – the library detects this and does an actual copy if any operation will change the relation.

6.11 Compressing relations

In past releases, `Relations` could be compressed to take up less memory. Because of improvements in the library's handling of memory, we no longer use this feature. The interface still exists to avoid breaking existing code, but the functions don't do anything.

If you find that the memory improvements are not sufficient for you, you can try re-enabling the compression features with the compiler option `-DINCLUDE_COMPRESSION`. Since we don't use this anymore, it is possible that code in the current release is not compatible with this option.

6.12 Reclaiming memory used by `Relations`

You may also want to reclaim the memory that a relation uses – for example, if you have an array of relations, and no longer need them. You can assign the result of `Relation::Null()` to each relation whose memory you want to reclaim.

Chapter 7

The Uniform Library

This chapter documents version 1.1.0 of the Uniform library, located in the `omega/uniform` directory. To use it in your programs, include the file `uniform/uniform.h`, and link against `libuniform.a`.

7.0.1 Introduction

The Uniform Library is a source to source parallelizing transformation system. It is a complete implementation of the transformation system described in Wayne Kelly's Ph.D. dissertation: "Optimization within a Unified Transformation Framework". It is designed to replace existing ad hoc transformation techniques by a sound underlying foundation on which future work can be built.

One of the key goals of this system is to produce the same transformed code for a given program, regardless of the form in which that program is presented. That is, the system should produce the same code even if some of the original loops had been interchanged or distributed. This is achieved by considering only the inherent computation and data dependences and ignoring everything else about the form in which that computation happened to be presented by the programmer.

It uses an abstraction called space mappings to describe how computation should be distributed across processors and an abstraction called time mappings to describe how the computations should be ordered on each processor. This provides a uniform way in which to represent all such transformations - hence the name of the library.

It contains components to select the best set of space mappings, to select the best set of time mappings given a set of space mappings and to generate parallel code that corresponds to these choices.

Much of the power of the system comes from the use of Tuple relations and sets provided by the Omega Library to represent and manipulate space mappings, time mappings, iteration spaces and data dependences.

7.0.2 Usage

Input:

- The system takes as input a program written in a language based on a sequential execution model.
- The system is designed to be somewhat language independent - rather than parsing a program in any given language, it asks questions about the statements in the program by way of a set of functions defined in a file called "uniform-interf.h". The system can therefore be ported to new compilers simply by providing a new implementation for those functions. An implementation has been provided for use with Petit, and is contained in the file "uniform-interf.c" in the petit src directory.
- There are currently many restrictions on the set of programs which can be transformed. It is hoped that many of these will be relaxed in the future:

  1. Programs must have static control flow. This means that at compile time it must be possible to give a simple closed form description of the set of computations that will be performed at

runtime (i.e., the set of iterations that will be executed at run time must be able to be described exactly using the Omega library's Tuple relations). This generally implies that the only looping structures are FOR loops, and that all loop bounds and conditionals are affine functions of outer level loop variables and symbolic constants.

2. Only single procedures can be transformed and any procedure calls are assumed to have no side effects.

Output:

- The system produces as output a parallel SPMD C program with calls to functions from the p4 portable runtime library for share memory systems (as modified by Monica Lam's SUIF group at Stanford University). Versions of this runtime library are available for the Stanford DASH machine, the SGI Power Challenge and the KSR.
- The transformed program is currently written to a file called "aux.out".

Invoking The System:

- To use the library directly, a program simply needs to call the function "uniform(...)". The single argument to this function is a string containing the options to be passed on the transformation system (see OPTIONS below).
- To use the transformation system from within Petit, simply use the "-W" option on the command line, followed immediately by any options that you want to have passed on to the transformation system.

  For example: "petit -x -Wr10c -r foo.t" will turn on array expansion and reduction operation recognition in Petit and pass the string "r10c" to the uniform transformation system. That in turn will specify that the computation to communication ratio for the target machine is 10:1 and that potentially expensive transitive closure calculations should be performed.

Options:

- `-c` Potentially expensive transitive closure calculations will be performed. This will improve the quality of the solution in some cases.
- `-d` Only select space mappings that can be specified as data distributions.
- `-m` Allow some time and space mappings to be specified manually.
- `-s` Use the naive search algorithm (useful in combination with trace option).
- `-t` Output trace information to the file "uniform.trace".
- `-px` Use x as the number of physical processors when estimating costs.
- The default value is 10.
- `-nx` Use x as the number of iterations per loop when estimating costs. The default value is 100.
- `-rx` Use x as the computation to communication ration of the target machine when estimating costs. The default value is 5.

Bibliography

[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.

[Dow72] P. Downey. Undeciability of presburger arithmetic with a single monadic predicate letter. Technical Report 18-72, Center for Research in Computing Technology, Havard Univ., 1972.

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

[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.

[PW93] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Lecture Notes in Computer Science 768: Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Portland, OR, August 1993. Springer-Verlag.

[PW94] William Pugh and David Wonnacott. Nonlinear array dependence analysis. Technical Report CS-TR-3372, Dept. of Computer Science, University of Maryland, College Park, November 1994.

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