

# The IDP system

Johan Wittocx      Maarten Mariën      Stef De Pooter

September 10, 2010

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Tutorial	3
1.1.1	Map Colouring	3
1.1.2	N-Queens	6
1.1.3	Defining Family Relations	8
1.1.4	A Last Example	9
<b>2</b>	<b>Input Language</b>	<b>11</b>
2.1	Problem Specifications and Instances	12
2.2	Comments and Whitespace	13
2.3	Vocabulary Declaration	13
2.3.1	Basic Declarations	13
2.3.2	The Input Vocabulary Part	14
2.3.3	The Output Vocabulary Part	14
2.3.4	The General Vocabulary Part	15
2.3.5	Advanced Declarations	15
2.4	Problem Description	17
2.4.1	Basic Syntax	17
2.4.2	Arithmetic	22
2.4.3	The Type of a Variable	23
2.4.4	Ordered Types	26
2.4.5	Extended Existential Quantifiers	27
2.4.6	Aggregates	28
2.4.7	Partial functions	30
2.4.8	Vocabulary Declarations	31
2.4.9	Implicit Constraints	31
2.5	Optimal Solutions	32
2.5.1	Minimizing/Maximizing a Term	32
2.5.2	Subset Minimality/Maximality	33
2.6	Problem Instance	33
2.6.1	Input Structure	33
2.6.2	Partial Input Structure	35
2.7	Structuring the Specification	36
2.7.1	Symbol Specifications	36

2.8	Example . . . . .	37
2.9	Obscure Features of IDP . . . . .	39
2.9.1	Headers for Mathematicians . . . . .	39
2.9.2	Extra semicolons . . . . .	40
<b>3</b>	<b>Invoking the IDP System</b>	<b>41</b>
3.1	The IDP interface . . . . .	41
3.2	System Architecture . . . . .	41
3.3	GIDL . . . . .	42
3.3.1	Input Files . . . . .	42
3.3.2	GIDL Options . . . . .	42

# Chapter 1

## Introduction

The IDP system is a system for model expansion in the context of FO(ID), an extension of first-order logic with inductive definitions. It is suitable for solving a wide variety of problems, such as scheduling and verification problems. Technically, it can solve all NP problems.

The style of problem solving using IDP is called *declarative*: one specifies formally *what* the solution to the problem should look like, rather than specifying—programming—*how* a solution can be obtained. Once the specification is complete, solutions can simply be found by letting the IDP system search for them.

Therefore, the main thing to learn when learning IDP, is the *meaning* of different statements one can write in the IDP language, as well as a *methodology* of writing formal specifications that capture the intended meaning.

This manual is conceived as follows:

- in Chapter 1, we introduce the system by example. The main features of the IDP language are demonstrated by a small set of simple examples;
- in Chapter 2, we provide a detailed description of the whole IDP language;
- in Chapter 3, we describe the architecture of the IDP system and how to use it.

### 1.1 Tutorial

In this section, we introduce the basic usage of the IDP system by means of a few examples. For an in-depth coverage of all features, the reader is referred to the next chapter.

#### 1.1.1 Map Colouring

Consider the problem of colouring a map with four different colours. Neighbouring countries should have a different colour. The following example shows

how this problem can be solved with IDP. In the example, everything between ‘/\*’ and ‘\*/’ is a comment, as is everything from ‘//’ till the end of the line. Comments are ignored by the system.

Example 1.1: Map Colouring

```
/*
Map colouring example
*/

Given:
  type Country
  type Colour
  Neighbour(Country, Country)

Find:
  Colouring(Country) : Colour

Satisfying:
  ! c1 c2 : Neighbour(c1, c2)
    => Colouring(c1) ~= Colouring(c2).

Data: // an example of a map
  Country = { Belgium; The_Netherlands; France;
             Germany; Luxembourg; Denmark }
  Colour = { yellow; green; red; orange }
  Neighbour = { Denmark, Germany;
               The_Netherlands, Germany; The_Netherlands, Belgium;
               Belgium, Luxembourg; Belgium, France;
               Belgium, Germany; Germany, France;
               Germany, Luxembourg; France, Luxembourg }
```

### Declarations

Observe that the specification consists of four different blocks of code. The first two (indicated by the headers ‘Given:’ and ‘Find:’) contain a declaration of the specific symbols that are used in the example. Each symbol has to be declared before it can be used.

The symbols declared below ‘Given:’ are the ones that describe the input to the problem. In this example, the input is a map and a set of colours. These are described by two *types* and one *predicate*. Intuitively, a type denotes a set of objects, while a predicate denotes a relation between different objects. I.e.:

```
Given:
  type Country // denotes the set of countries
               // on the map
```

```

type Colour // denotes the set of colours
            // that can be used

Neighbour(Country, Country) // denotes the
// 'neighbour' relation between two countries.

```

In general, a type declaration is of the form ‘type MyType’, where MyType is the name of the type. A name of a type should start with an uppercase letter. A predicate declaration is of the form ‘MyPred(MyType<sub>1</sub>, ..., MyType<sub>n</sub>)’ where the name MyPred of the predicate should start with an uppercase letter. The types that occur in the declaration should have been declared before. A predicate without any arguments can be declared by ‘MyPred’ or by ‘MyPred()’.

The symbols declared below ‘Find:’ are the ones that describe the solution. In this case, only one symbol is declared, namely a function that maps each country to a colour. In general, a function declaration is of the form ‘MyFunc(MyType<sub>1</sub>, ..., MyType<sub>n</sub>) : MyType<sub>0</sub>’. A function without arguments is called a *constant* and can be declared by ‘MyConst : MyType’.

### Constraints

The part of the input below the ‘Satisfying:’ header contains the constraints that should be satisfied by each solution. In the example, there is only one constraint:

```

! c1 c2 : Neighbour(c1, c2)
  => Colouring(c1) ~= Colouring(c2).

```

Here, ‘!’ is to be read as *for each ... it holds that*, ‘=>’ as *if ... then* and ‘~=’ as *is not equal to*. Hence the whole constraint expresses

*For each country  $c_1$  and  $c_2$ , it holds that if  $c_1$  is a neighbour of  $c_2$  then the colour of  $c_1$  is not equal to the colour of  $c_2$ .*

Every colouring that satisfies this constraint is indeed a solution to the map colouring problem.

The following is a list of some of the symbols that can be used for expressing constraints:

symbol	declarative reading
!	for each ... it holds that
?	there exist ... such that
~	not ...
&	... and ...
	... or ...
=>	if ... then ...
<=>	... if and only if ...

Variables, such as  $c_1$  and  $c_2$  in the constraint above, start with a lowercase letter.

Each constraint ends with a ‘.’.

## Instance

In the last part of the example, below ‘Data:’, the types are enumerated, as well as all other symbols that are declared in the ‘Given’ part. E.g., ‘Colour = { yellow; green; red; orange }’ lists all colours and ‘Neighbour = {Denmark, Germany; The\_Netherlands,Germany; The\_Netherlands,Belgium; ...}’ all pairs of countries that are neighbours. Note that pairs are separated by semicolons, while the individual elements in a pair are separated by commas. A type must be enumerated before a predicate having that type as one of its arguments can be enumerated. E.g., Country must be enumerated before Neighbour.

## Invoking IDP

The IDP system can be downloaded from <http://dtai.cs.kuleuven.be/krr/software>. To try out the map colouring example, put the specification (i.e. the ‘Given’, ‘Find’ and ‘Satisfying’ parts) in a file `mapcol.idp` and the instance (i.e. the ‘Data’ part) in a file `mapcolinst.idp`. Start the IDP interface by opening the jar file (`IDPx.y.jar`) with java, or by running

```
$ java -jar IDPx.y.jar
```

Point to the two files you have just created, and push Run. The output should be a solution like

```
Colouring = {Belgium->red; Denmark->green; France->yellow;
Germany->orange; Luxembourg->green; The_Netherlands->green; }
```

### 1.1.2 N-Queens

The  $n$ -Queens problem consists of placing  $n$  queens on a  $n \times n$  chess board such that no queen attacks another. The following is an IDP specification of this problem:

Example 1.2:  $n$ -Queens Problem

```
/* *****
n-Queens problem
***** */

Given:
  type int Row
  type int Col

Find:
  Queen(Row, Col) // denotes the position
                  // of the queens

Declare:
  Diag(Row, Col, Row, Col) // Diag(r1, c1, r2, c2)
```

```

        // denotes that square (r1,c1) and (c2,r2)
        // are on the same diagonal

Satisfying:
    // There is exactly one queen on each row
    ! r : ?1 c : Queen(r,c).

    // There is exactly one queen on each column
    ! c : ?1 r : Queen(r,c).

    // Define what it means that two squares are
    // on the same diagonal
    { Diag(r1,c1,r2,c2) <- r1 < r2
      & abs(r1-r2) = abs(c1-c2). }

    // If (r1,c1) and (r2,c2) are on the same
    // diagonal, they cannot both contain a queen.
    ! r1 c1 r2 c2 : Diag(r1,c1,r2,c2) =>
      ~(Queen(r1,c1) & Queen(r2,c2)).

Data:
    /* instance for 8 queens */
    Row = {1..8}
    Col = {1..8}

```

## Integer Types

In the above example, the declarations of the types `Row` and `Col` contain the extra keyword `'int'`. This declares the type to be an *integer type*. Enumerations of integer types can only contain integer objects. Integer types can be used in arithmetic expressions. E.g., if `Row` and `Col` would not have been declared as integer types, then `'abs(x1-x2) = abs(y1-y2)'` would yield an error message.

## Declare Block

Every symbol that is not part of the input to the problem, nor relevant to be shown in the solution can be declared in a block below the header `'Declare:'`.

## Exactly $n$

In the constraint `'! r : ?1 c : Queen(r,c)'`, the symbol `'?1'` means *there exists exactly one ... such that*. As such, the constraint can be read as:

For each row  $r$ , there exists exactly one column  $c$  such that there is a queen on square  $(r,c)$ .

In general, one can write ‘?n’, meaning *there exist exactly n ... such that*. Here n is a positive integer.

## Definitions

Besides constraints, the *n*-queens example contains a *definition* of the *Diag* predicate:

```
{ Diag(x1,y1,x2,y2) <- abs(x1-x2) = abs(y1-y2). }
```

This definition can be read as:

*Diag* holds for (x1,y1,x2,y2) if and only if the absolute value of x1-x2 is equal to the absolute value of y1-y2.

In general, a definition is a set of rules, enclosed between ‘{’ and ‘}’. Each rule is of the form ‘MyPred(...) <- ...’ and ends with a ‘.’. In the next example, more complicated definitions will be shown.

## Shorthands for enumerating types

The enumeration of the rows, ‘Row = {1..8}’, is a shorthand for writing ‘Row = {1;2;3;4;5;6;7;8}’.

### 1.1.3 Defining Family Relations

This example merely shows the use of definitions. Not only simple ones, but also recursive definitions. I.e., a definition where a concept is defined in terms of itself.

Example 1.3: Family Relations

```
Given:
  type Person
  Parent(Person,Person) // Parent(x,y) means
                        // x is a parent of y
Find:
  Sibling(Person,Person)
  Ancestor(Person,Person)
Satisfying:
  // x is a sibling of y if they have a
  // common parent p
  { Sibling(x,y) <- ? p : Parent(p,x) & Parent(p,y). }

  {
    // x is an ancestor of y if it is a
    // parent of y ...
    Ancestor(x,y) <- Parent(x,y).
```

```

// ... or x is an ancestor of an ancestor p of y
Ancestor(x,y) <- ? p : Ancestor(x,p)
                        & Ancestor(p,y).
}
Data:
...
```

As can be seen, the definition of `Ancestor` is recursive. The second rule in that definition defines ancestor in terms of itself.

Observe the difference between the definition of `Ancestor` as in the example above and the following constraint:

```

Ancestor(x,y) <=> (Parent(x,y) |
? p : (Ancestor(x,p) & Ancestor(p,y))).
```

Given three persons A,B,C and a `Parent` relation {A,B; B,C}, the only model of the definition is that A is an ancestor of B and C and B is an ancestor of C. On the other hand, the constraint does not exclude, e.g., that A is an ancestor of itself. The model of the definition is the minimal model of the constraint.

### 1.1.4 A Last Example

In the *Hamiltonian circuit problem*, a number of cities and roads between these cities are given. The task is to find a circular route, passing exactly once through each city. The following is an IDP specification of this problem. Here, '`?<2`' means *there exist less than 2*. This notation generalizes to '`?<n`', where `n` is a natural number.

Example 1.4: Hamiltonian Circuit

```

/*****
Hamiltonian Circuit
*****/

Given:
  type City
  Road(City, City)

Find:
  In(City, City) // Denotes the roads used
                  // by the route

Declare:
  Start : City // a city where the route starts
              // and ends
  Visit(City) // denotes the visited cities
```

**Satisfying:**

```
// Only travel over roads
! c1 c2 : In(c1,c2) => Road(c1,c2).
// Visit each city at most once
! c_in : ?<2 c_out : In(c_out,c_in).
// The route does not split
! c_out : ?<2 c_in : In(c_out,c_in).

// Define what it means to visit a city
{ Visit(c) <- In(Start,c).
  Visit(c) <- Visit(c_out) & In(c_out,c). }

// Every city must be visited
! c : Visit(c).
```

**Data:**

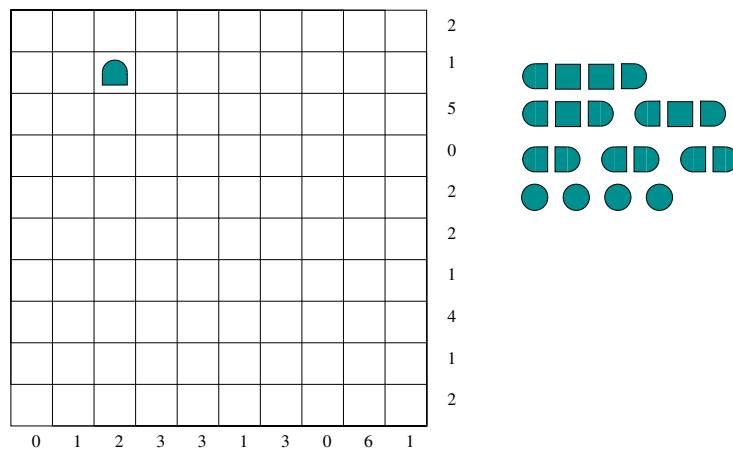
```
City = {A..D}
Road = {A,B; A,C; B,A; B,C; C,D; D,B; D,A}
```

## Chapter 2

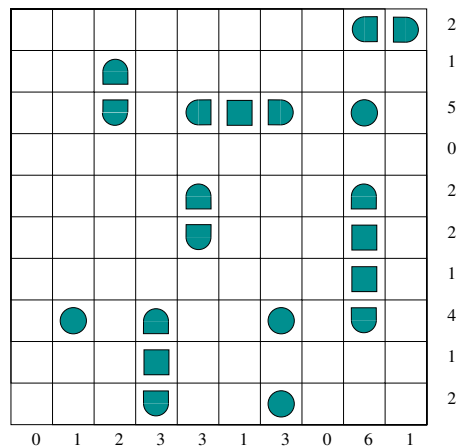
# Input Language

In this chapter, the input language for the IDP system is described, as well as the semantics of the features that are not in standard FO(ID). We keep the explanation of the semantics on an informal level as much as possible.

We use the *battleship puzzle* as a running example to illustrate the syntax of the IDP language. Such a puzzle consists of a  $10 \times 10$  grid, a number for each row and column of the grid, and a list of ships, as shown below.



The ships must be placed in the grid and should not touch each other, not even diagonally. Moreover, each row or column must contain precisely the number of ship parts indicated on that row or column. As a hint, some parts are already in the grid. The following is a solution to the particular puzzle shown above.



## 2.1 Problem Specifications and Instances

In the battleship puzzle, the *input* to the problem consists of two lists of 10 numbers and a list of ships. For any such input, the puzzle may or may not have a solution, or many solutions. Thus we have a distinction between the *problem specification* (the rules of the puzzle, that hold for any input) and the specific *problem instance* (the input to the puzzle).

In IDP, we make the same distinction. An IDP problem instance starts with the keyword ‘**Data:**’. This keyword is then followed by an interpretation for the input symbols (see Section 2.6). Such an instance can come in the same file as an IDP problem specification at the end, or as a separate file.

The IDP problem specification specifies, firstly, which symbols must be interpreted in any valid IDP instance to go along with the specification. This section is started by the keyword ‘**Given:**’. Next, after the keyword ‘**Find:**’ one can specify which symbols form the solution to the problem. Finally, the keyword ‘**Satisfying:**’ marks the beginning of the actual specification (see Section 2.4).

Thus an IDP specification (with instance included) has the following structure:

```

Given:
    // The symbols of which an interpretation
    // will be given under "Data".
Find:
    // The symbols that form the solution.
Satisfying:
    // The actual specification.
Data:
    // A problem instance
    // (an interpretation of the "Given" symbols).

```

Additionally, one can declare more symbols that will be used in the specification after the keyword ‘`Declare:`’. The declaration of symbols in a ‘`Given`’, ‘`Find`’ or ‘`Declare`’ block is explained in Section 2.3.1.

## 2.2 Comments and Whitespace

Anywhere in an IDP specification, comments can be placed, which are ignored by the IDP system. IDP uses C-style comments:

- Anything between ‘`//`’ and the end of the line is a comment.
- Anything between ‘`/*`’ and ‘`*/`’ is a comment.

Also arbitrary whitespaces (spaces, tabs, and returns) are ignored, except the return that marks the end of a comment started with ‘`//`’, and only insofar as they do not break up keywords (e.g. ‘`/ /`’ instead of ‘`//`’ is invalid).

## 2.3 Vocabulary Declaration

Before a certain type, predicate or function can be used in the description of the problem, it must be declared. There are four places in an IDP specification where new vocabulary can be declared:

- in the *input vocabulary part*,
- in the *output vocabulary part*,
- in the *general vocabulary part*,
- inside the description of the problem.

The declarations have the same syntax in each of these parts. We first present the syntax for basic declarations and then discuss the differences between the three first parts. How to declare symbols inside the problem description is explained in section 2.4. Finally, we give an overview of more involved declarations.

### 2.3.1 Basic Declarations

A vocabulary consists of types, predicate symbols and function symbols.

A type represents a class of objects. E.g., the ships, the lengths of the ships, the rows and the columns are types in the battleship puzzle. A basic *type* declaration consists of the keyword ‘`type`’, followed by the name of the type.

Relation symbols denote relations between objects represented by the types. E.g. we can declare a relation *HasShip* between rows, columns and ships, with the intended meaning that a certain row  $r$ , column  $c$  and ship  $s$  are in the relation if and only if the square on the position  $(r, c)$  contains a part of ship  $s$ . A predicate symbol declaration consists of the name of the symbol, followed by

a ‘(’, a comma separated list of types, and a ‘)’’. All types that are in the list must have been declared before. When a type  $T$  is declared, a predicate  $T(T)$  is automatically declared implicitly.

A function symbol denotes a function between types. E.g. a function which maps each ship to its length. A function symbol declaration consists of the name of the symbol, followed followed by a ‘(’, a comma separated list of types, a ‘)’, a colon ‘:’, and a name of a type. The latter type is called the *return type* of the function. The types in the list are called the *input types*. All these types must have been declared before the function declaration.

Names of types, predicates and functions start with an uppercase letter.

The following is an example of a part of the declarations for the battleship puzzle.

```
/* type declarations */
type Ship      // Also declares the
               // predicate Ship(Ship)

type Length
type Row
type Col
type Direction // Horizontal or vertical

/* Predicate declarations */
HasShip(Row, Col, Ship)

/* Function declarations */
ShipLength(Ship) : Length // maps a ship to
                          // its length
ShipDir(Ship) : Direction // maps a ship to
                          // its direction
```

### 2.3.2 The Input Vocabulary Part

As mentioned before, declarations can be placed in different parts of the IDP specification. The *input vocabulary part* starts with the header ‘Given:’. All symbols that are declared in this part are the symbols whose interpretation is known before the problem is solved. In our running example, the ships, the lengths of the ships, the numbers for each row and column should be declared in the input vocabulary part.

### 2.3.3 The Output Vocabulary Part

The *output vocabulary part* contains the symbols whose interpretation constitutes the solution to the problem. E.g., the *HasShip* predicate symbol and the *ShipDir* function belong to the output vocabulary. The output vocabulary part starts with the header ‘Find:’. The output vocabulary cannot contain declarations of types.

### 2.3.4 The General Vocabulary Part

All declarations that do not belong to the input or output vocabulary can be declared in the *general vocabulary part*, which starts with the header ‘**Declare:**’. Typically, the symbols that are declared here are auxiliary symbols: they are used to simplify the description of the problem, but are not relevant to describe the solution. An example of such a symbol could be the predicate  $HasShip(Row, Col)$ , with the intended meaning that  $(r, c)$  belongs to the relation if there is a part of *some* ship at square  $(r, c)$ .

The following example shows the declarations for our example.

```
/******  
    Battleship puzzle  
******/  
  
Given:  
    type Ship  
    type Length  
    type Row  
    type Col  
    type Direction  
  
    ShipLength(Ship) : Length  
  
Find:  
    HasShip(Row, Col, Ship)  
    ShipDir(Ship) : Direction  
  
Declare:  
    HasShip(Row, Col)
```

Observe that it is allowed to have multiple predicates with the same name, provided the number of arguments differs.

### 2.3.5 Advanced Declarations

#### Predicate Symbols

A predicate  $P$  that has no arguments, i.e., an *atom*, can be declared both by ‘ $P$ ’ and by ‘ $P()$ ’.

#### Function Symbols

A function with no input arguments is called a *constant*. A constant  $C$  with output type  $T$  can be declared both by ‘ $C:T$ ’ and by ‘ $C():T$ ’.

Functions are always *total*, which means that they have an output value for each of their input values. E.g. the function  $ShipLength$  should assign a length to every ship. If one wants a function symbol that does not have

this property, it should be explicitly declared as a *partial* function. This is achieved by placing the keyword ‘**partial**’ before the declaration of the function. Additionally, one can specify a restricted domain of input values on which the function must be total. E.g., for submarines (the ships consisting of only one part), the direction (vertical or horizontal) is irrelevant. Thus, the function *ShipDir* could be declared partial. But each ship with length more than one must have a direction, i.e., for these ships *ShipDir* should be total. We illustrate this in the following example:

```

Declare:
  type Ship
  type Length
  type Direction
  ShipLength(Ship) : Length
  partial F(Ship) : Ship // A partial function
  partial ShipDir(s : Ship) : Direction
    domain ShipLength(s) > 1

```

The last declaration contains a variable followed by a colon and a type name, instead of only a type name in the input type position of the function. The declaration is followed by the keyword ‘**domain**’ and a formula, containing the variable. The precise meaning of such a declaration, as well as the syntax of formulas is explained in section 2.4.

### Types

Additionally to basic type declarations, the following are possible.

- A object can only be used in arithmetical expressions (see section 2.4.2) if its type is an *integer type*. Integer types are declared by adding the keyword ‘**int**’ to the type declarations. It has to be placed immediately after the ‘**type**’ keyword. In our example, the types *Row*, *Col* and *Length* could be declared as integer types.
- A type *A* can be a direct subtype of another type *B*, called its *parent type*. This is declared by adding the keyword ‘**isa**’, followed by the name of type *B*. E.g., we could declare a type *Submarine*, which is a direct subtype of *Ship*. An *ancestor type* of a subtype is its parent type or an ancestor type of its parent type. The objects belonging to a subtype must be a subset of the objects belonging to the parents type. For the semantics of subtypes, see 2.4.3 and 2.4.9.
- All objects of a type can be enumerated when the type is declared. Doing so has the effect of simultaneously declaring a constant for each object. This constant has the same name as the corresponding object. The syntax of an enumeration for a type is illustrated in the example below for the type *Direction*. It corresponds to the syntax for enumerating domains in

the problem instance description (see section 2.6). Also, the objects must obey the rules for the names of constants.

To avoid typing the keyword ‘`type`’ over and over, several types can be declared all at once, as is shown in the example below.

```
type { // avoid retyping ‘type’
  Ship
  Submarine isa Ship
  int Row
  int Col
  int Length
  Direction = { Hor; Ver }
  /* Declares the Direction objects Hor and Ver, and
     ALSO declares the constants Hor:Direction and
     Ver:Direction, with the obvious interpretations.
  */
}
```

## Variable Declarations

In a general vocabulary part, also variables can be declared to have a certain type. This has the effect that whenever this variable is used below the declaration, it has this type. For non-declared variables, the type is inferred from the problem description, as is explained in the next section. A variable declaration consists of the name of the variable, followed by a colon and the name of a type. The name of a variable should start with a lowercase letter.

## 2.4 Problem Description

### 2.4.1 Basic Syntax

The problem description starts with the header ‘`Satisfying:`’, and may further contain first-order logic sentences and inductive definitions. The sentences express constraints that should be satisfied by the solution. E.g., the battle-ship problem description should contain a sentence expressing that ships do not touch each other. On the other hand, definitions are used to define concepts in terms of other concepts.

### Terms

Before explaining the syntax for sentences, we need to introduce the concept of a term and a formula. We also give the syntax for terms and formulas in IDP.

A *term* is inductively defined as follows:

- a variable is a term;

- a constant is a term;
- if  $F$  is a function symbol with  $n$  input arguments and  $t_1, \dots, t_n$  are terms, then  $F(t_1, \dots, t_n)$  is a term.

In IDP, variables start with a lowercase letter and may contain letters, digits and underscores. When writing a term in IDP, the constant and function symbols occurring in that term should be declared before.

The *type of a term* is defined as its return type (see section 2.3.1) in the case of constants and functions. The type of a variable is derived from its occurrences in formulas (see section 2.4.3). If a term occurs in an input position of a function, then the type of the term and the type of the input position must have a common ancestor type. The following example may clarify this.

```

Declare:
  type {
    Ship
    Submarine isa Ship
    ShortShip isa Ship
    Length
  }
  MyShip : Ship
  MySub : Submarine
  MyLen : Length
  ShipLength(Ship) : Length
  ShortLength(ShortShip) : Length

/* some terms */
MyShip           // has type Ship
ShipLength(MyShip) // has type Length

ShipLength(MyLen) // wrong term: MyLen is not of
                  // type Ship

ShipLength(MySub) // ok, MySub has type Submarine
                  // and Ship is a common
                  // ancestor of Submarine and Ship

ShortLength(MySub) // also ok, MySub has type
                  // Submarine and Ship is a
                  // common ancestor of
                  // Submarine and ShortShip

ShortLength(MyShip) // also ok

```

## Formulas and Sentences

A *formula* is inductively defined by:

- if  $P$  is a predicate symbol with arity  $n$  and  $t_1, \dots, t_n$  are terms, then  $P(t_1, \dots, t_n)$  is a formula;
- if  $t_1$  and  $t_2$  are terms, then  $t_1 = t_2$  is a formula;
- if  $\varphi$  and  $\psi$  are formulas and  $x$  is a variable, then the following are formulas:  
 $\neg\varphi, \varphi \wedge \psi, \varphi \vee \psi, \varphi \Rightarrow \psi, \varphi \Leftarrow \psi, \varphi \equiv \psi, \forall x \varphi$ , and  $\exists x \varphi$ .

The following order of binding is used:  $\neg$  binds tightest, next  $\wedge$  and  $\vee$ , then  $\Rightarrow$  and  $\equiv$ , and finally  $\forall$  and  $\exists$ . Desambiguation can be done using brackets ‘(’ and ‘)’. E.g. the formula  $\forall x P(x) \wedge \neg Q(x) \Rightarrow R(x)$  is equivalent to the formula  $\forall x ((P(x) \wedge (\neg Q(x))) \Rightarrow R(x))$ .

As for terms, if term  $t$  occurs in predicate  $P$ , then the type of  $t$  and the type of the input position of  $P$  where it occurs must have a common ancestor type. For formulas of the form  $t_1 = t_2$ ,  $t_1$  and  $t_2$  must have a common ancestor type or they must both have an integer type.

```
Declare:
  type Ship
  type Submarine isa Ship
  type int Length
  type int Row
  type int Col

  HasShip(Row, Col, Ship)
  ShipLength(Ship) : Length
  MyShip : Ship
  MySub : Submarine
  MyRow : Row
  MyCol : Col

/* some formulas */

  HasShip(MyRow, MyCol, MyShip)

  HasShip(MySub, MyCol, MySub) // wrong formula: MySub
    // in first position is not of type Row

  MyShip = MySub // ok
  MyShip = MyRow // wrong, Ship and Row have no
    // common ancestor
  HasShip(MyCol, MyCol, MyShip) // ok, although
    // Row and Col have no common ancestor, they
    // both have an integer type
```

The *scope* of a quantification  $\forall x$  or  $\exists x$ , is the quantified formula. E.g., in  $\forall x \psi$ , the scope of  $\forall x$  is the formula  $\psi$ . An occurrence of a variable  $x$  that is not inside the scope of a quantification  $\forall x$  or  $\exists x$  is called *free*. A *sentence* is a formula containing no free occurrences of variables. If an IDP problem specification contains formulas that are not sentences, the system will return an error message, specifying which variables occur free. Each sentence in IDP should end with a dot ‘.’.

The IDP syntax of the different symbols in formulas are given in the table below. Also the informal meaning of the symbols is given.

Logic	IDP	Declarative reading
$\wedge$	&	and
$\vee$		or
$\neg$	~	not
$\Rightarrow$	=>	implies
$\Leftarrow$	<=	is implied by
$\equiv$	<=>	is equivalent to
$\forall$	!	for each
$\exists$	?	there exists
$=$	=	equals
$\neq$	~=	does not equal

A universally quantified formula  $\forall x P(x)$  becomes ‘! x : P(x)’ in IDP syntax, and similarly for existentially quantified formulas. As a shorthand for the formula ‘! x : ! y : ! z : Q(x,y,z).’, one can write ‘! x y z : Q(x,y,z)’.

In IDP, every variable has a type. The informal meaning of a sentence of the form  $\forall x \psi$ , respectively  $\exists x \psi$ , where  $x$  has type  $T$  is then ‘for each object  $x$  of type  $T$ ,  $\psi$  must be true’, respectively ‘there exists at least one object  $x$  of type  $T$  such that  $\psi$  is true’. If a variable is declared (see section 2.3.5) then its type is the type specified in that declaration. Otherwise, the type is derived from the occurrences of the variable in the quantified formula. How this is done is explained in section 2.4.3.

To illustrate the syntax and meaning of sentences, a part of the battleship problem description is shown below.

```
Satisfying:

// Each ship's first part is located
// somewhere on the grid...
! s : ? r c : Init(s,r,c).

// ... and can only be at one location.
! s r1 r2 c1 c2 : (Init(s,r1,c1) & Init(s,r2,c2))
                  => (r1 = r2 & c1 = c2).

// If some square contains water,
// it does not contain a part of a ship.
```

```

! r c : Water(r,c) => ~HasShip(r,c).

// Ships cannot touch each other
! r1 c1 r2 c2 s1 s2: Neighbour(r1,c1,r2,c2)
  & HasShip(r1,c1,s1) & HasShip(r2,c2,s2)
=> s1 = s2.

```

## Definitions

A definition defines a concept, i.e. a predicate, in terms of other predicates. Formally, a definition is a set of rules of the form

$$\forall x_1, \dots, x_n P(t_1, \dots, t_m) \leftarrow \varphi$$

where  $P$  is a predicate symbol,  $t_1, \dots, t_m$  are terms that may contain the variables  $x_1, \dots, x_n$  and  $\varphi$  a formula that may contain these variables.  $P(t_1, \dots, t_m)$  is called the *head* of the rule and  $\psi$  the *body*.

A definition in IDP syntax consists of a set of rules, enclosed by ‘{’ and ‘}’. Each rule ends with a ‘.’. The definitional implication  $\leftarrow$  is written ‘<-’. The quantifications before the head may be omitted in IDP, i.e., all free variables of a rule are implicitly universally quantified. If the body of a rule is empty, the rule symbol ‘<-’ can be omitted. The following are definitions that may occur in the battleship specification.

```

/* Define HasShip/2 in terms of HasShip/3 */
{ // start of the definition

  // Square r,c contains a part of a ship if there
  // exists a ship s such that r,c contains a part
  // of that ship s.
  HasShip(r,c) <- ? s : HasShip(r,c,s).

} // end of the definition

/* Define HasShip/3 in terms of the position of
the initial part of a ship, its length and
direction */
{

  // square r,c contains a part of ship s if it
  // contains its initial part
  HasShip(r,c,s) <- Init(s,r,c).

  // If the direction of ship s is vertical and
  // the position of its initial part is square

```

```

// (r1,c), then every square (r,c), where r is
// between r1 and r1 + length of s, contains a
// part of ship s
HasShip(r,c,s) <- Init(s,r1,c)
                  & Direction(s) = Ver
                  & r > r1 & r < Length(s) + r1.

// Same for ships with a horizontal direction
HasShip(r,c,s) <- Init(s,r,c1)
                  & Direction(s) = Hor
                  & c > c1 & c < Length(s) + c1.
}

```

Also recursive definitions are allowed in IDP. The following defines a *reachability* relation recursively.

```

Given:
  type City
  Connected(City, City)
Find:
  Reachable(City, City)
Satisfying:
{
  // c2 is reachable from c1 if they are
  // connected ...
  Reachable(c1,c2) <- Connected(c1,c2).
  // ... or there exists a city c3 that is
  // reachable from c1 and connected to c2
  Reachable(c1,c2) <- Connected(c1,c3)
                      & Reachable(c3,c2).
}

```

### 2.4.2 Arithmetic

Besides terms as defined above, *arithmetic terms* can be used in IDP. They can occur in each position where a normal term with an integer type can occur. The following are arithmetic terms:

- A term with an integer type is an arithmetic term;
- Every integer  $0, -1, 1, -2, 2, \dots$  is an arithmetic term;
- If  $t_1$  and  $t_2$  are arithmetic terms, then  $-t_1, t_1 + t_2, t_1 - t_2, t_1 * t_2, t_1 / t_2, t_1 \bmod t_2$  and  $abs(t_1)$  are arithmetic terms. These arithmetic operators are treated as partial functions (see 2.4.7).

The table below lists the IDP syntax and meaning of the symbols in arithmetic terms.

math	IDP	
+	+	addition
-	-	subtraction
*	*	multiplication
/	/	integer division
mod	%	remainder
abs	abs	absolute value

To compare two arithmetic terms, the following can be used.

math	IDP
=	=
≠	~ =
<	<
≤	=<
>	>
≥	>=

Note that ‘<=’ denotes reverse implication, and not ‘less than’! ‘Less than’ is denoted by ‘<’ instead.

### 2.4.3 The Type of a Variable

There are three ways to assign a type  $t$  to a variable  $v$ :

- Declare the variable  $v$  to have type  $t$ , as explained in section 2.3.5. This has the side-effect that every variable with name  $v$  is of type  $t$ .
- Explicitly mention the type of  $v$  between ‘[’ and ‘]’ when  $v$  is quantified. Then  $v$  gets type  $t$  in the scope of the quantifier. E.g.,

```
// explicitly assign type Ship to s,
// Row to r and Col to c
! s [Ship] : ? r [Row] c [Col] :
    HasShip(r,c,s).

{ ! r [Row] c [Col] : HasShip(r,c) <-
  ? s [Ship] : HasShip(r,c,s).
}
```

To specify that a variable should range over all integers, one can write ‘[int]’.

- The standard way: do not mention the type of  $v$  but let the system automatically derive it. The rest of this section explains how this is done.

## Automatic derivation of types for variables

We distinguish between *typed* and *untyped* occurrences. The following are typed occurrences of a variable  $x$ :

- an occurrence as argument of a predicate:  $P(\dots, x, \dots)$ ;
- an occurrence as argument of a function:  $F(\dots, x, \dots) = \dots$ ;
- an occurrence as return value of a function:  $F(\dots) = x$  or  $F(\dots) \neq x$ ;
- an occurrence as argument or return value of an arithmetic function. The type of such a position is an implicit type, whose interpretation ranges over all integers.

All others positions are untyped.

Basically, if a variable occurs in a typed position, it gets the type of that position. This is illustrated in the following example.

```
Given:
  type Ship
  type int Length
  type int Row
  type int Col
  ShipLength(Ship) : Length
Declare:
  HasShip(Row, Col, Ship)
Satisfying:
  ! s : ? r c : HasShip(r,c,s). // s has type
    // Ship, r type Row and c type Col

  ! s : ? l : ShipLength(s) = l. // s has type
    // Ship, and l type Length

  ! x : ? y : x = y + 1. // both x and y range
    // over all integers
```

If a declared variable with type  $T_1$  occurs in a typed position of type  $T_2$ , then  $T_1$  and  $T_2$  should have a common ancestor type.

The more complicated cases arise when a variable does not occur in any typed position, or it occurs in two typed positions with a different type. The system is designed to give a reasonable type to such variables. However, the choices made by the system are ad hoc and are probably not the ones the user intended. When in doubt, one can explicitly assign types to variables, or run GIDL with option `--data` to check the derived types.

First consider the case where a variable occurs in typed positions with different types. The IDP system will then give a warning. If all the typed positions where the variable occurs have a common ancestor type  $T$ , then the variable is

assigned this type  $T$ . If they do not have a common ancestor, but are all of an integer type, then the variable is assumed to range over all integers.

```

Given:
  type Ship
  type ShortShip isa Ship
  type Submarine isa Ship
  type int Row
  type int Col
Declare:
  CanDive(Submarine)
  IsShort(ShortShip)
  HasShip(Row,Col,Ship)
Satisfying:
  ! s : CanDive(s) => IsShort(s). // s occurs in a
    // position of type Submarine and type
    // ShortShip, so it is assigned type Ship.

  ! s : ? r : HasShip(r,r,s). // r ranges over all
    // (positive and negative) integers.

```

Now consider the case where a variable does not occur in a typed position. If it occurs in a position where it is forced to be of an integer type, then it ranges over all integers. Else, the IDP system reports an error.

```

! x : ? y : x = y. // error: cannot derive
  // the types of variables x and y

! x : 1 < x & x < 5 => x ~ 10 // Ok. Since x
  // is compared to integers, it must have an
  // integer type. Hence, x is assumed to range
  // over all integers.

```

Because of the architecture of the system, problems can only be solved if all variables with an integer type range over a finite number of integers. Sometimes, however, if a variable ranges over all integers, only a finite interval of them is relevant. E.g., the constraint ' $! x : 1 < x \ \& \ x < 5 \Rightarrow x = 10$ ' is certainly satisfied for all  $x$ 's outside the interval  $[2..4]$ , so only this interval is relevant to look at. Therefore, this constraint does not yield an error message. It is beyond the scope of this manual to describe when the system can find a finite relevant interval and when it cannot. When it cannot, one of the following error messages is returned:

```

ERROR: Could not derive an upper bound for integer
       variable 'x'.
ERROR: Could not derive a lower bound for integer
       variable 'x'.
ERROR: Could not derive bounds for integer

```

```
variable 'x'.
```

To avoid these errors, one can add bounds:

- If  $x$  is universally quantified, i.e., it occurs as ' $! x : \dots$ ', then write ' $! x : \text{phi}(x) \Rightarrow \dots$ ' instead, where  $\text{phi}$  is a formula that can obviously only be true for a finite interval of  $x$ 's. E.g.,  $\text{phi}$  is the formula ' $1 < x \ \& \ x < 5$ '.
- If  $x$  is existentially quantified, then write ' $? x : \text{phi}(x) \ \& \ \dots$ ' instead of ' $? x : \dots$ ', where  $\text{phi}$  is as above.

## 2.4.4 Ordered Types

### Ordering of Non-Integer Types

Besides terms with an integer type, also terms with the same non-integer type can be compared using '<', '>', '=<' and '>='. These symbols are then interpreted as the alphabetical order on the objects of that type. To refer to the least, respectively greatest, element of that order, the built-in term 'MIN', respectively 'MAX' can be used. The built-in predicate 'SUCC', taking two arguments, is interpreted by *the successor of the first argument is the second argument*. The type of 'MIN' and 'MAX' are derived similar to the derivation of the type of a variable.

```
Given:
  type Animal = { Horse; Cow; Goat; Donkey }
Satisfying:
  Horse > Donkey.           // true
  Horse < Cow.              // false
  MIN = Cow.               // true
  ! x : Animal(x) => MAX >= x. // true
  SUCC(Donkey, Goat).      // true
  SUCC(Donkey, Horse).     // false
```

The use of 'MIN', 'MAX' and 'SUCC' is strongly discouraged, especially when using subtypes.

Note that if a type is not declared integer, it is ordered alphabetically, even if its objects are integers.

### Chains of (in)equalities

As in mathematics, one can write chains of (in)equalities in IDP. They can be used as shorthands for conjunctions of (in)equalities. E.g.:

```
Satisfying:
  ! x y : (1 =< x < y =< 5) => ...
  // is a shorthand for
  ! x y : (1 =< x) & (x < y) & (y =< 5) => ...
```

## 2.4.5 Extended Existential Quantifiers

As already mentioned, the existential quantifier ‘?’ can be used to express “*there exists a ... such that ...*”. Adding a number  $n$  immediately behind the ‘?’ changes the meaning to “*there exist precisely  $n$  ... such that ...*”. On the other hand, adding  $< n$  changes the meaning to “*there exist strictly less than  $n$  ... such that ...*”.

```
Given:
  type {
    Ship
    int Row
    int Col
  }

Find:
  Init(Ship, Row, Col) // a ship's initial segment
  HasShip(Ship, Row, Col)

Satisfying:
  // A ship's initial segment is at exactly
  // one position.
  ! s : ?1 r c : Init(s, r, c).
  // A position cannot contain segments of more than
  // one ship
  ! r c : ?<2 s : HasShip(s, r, c).
```

Observe the following subtle difference between writing

$$! s : ?1 r c : \text{Init}(s, r, c)$$

and

$$! s : ?1 r : ?1 c : \text{Init}(s, r, c).$$

The former is to be read as

For each ship  $s$  there is exactly one *pair*  $(r, c)$  such that the initial segment of  $s$  is at  $(r, c)$ .

The latter has the meaning

For each ship  $s$  there is exactly one row  $r$  such that there is exactly one column  $c$  such that the initial segment of  $s$  is at  $(r, c)$ .

I.e., the latter does not exclude that a ship has more than one initial fragment, as long as for each row besides  $r$ , there are none ore strictly more than one column that contains the initial segment of  $s$ . Hence for any number  $n$ , ‘ $?n x y :$ ’ is not a shorthand for ‘ $?n x : ?n y :$ ’. The same observation holds for ‘ $?<n$ ’.

## 2.4.6 Aggregates

Aggregates are functions that take a set as argument, instead of a simple variable. IDP supports some aggregates that map a set to an integer. As such, they can be seen as integer terms.

There are two kinds of sets in IDP.

- An expression of the form ‘[  $\phi_1$  ;  $\phi_2$  ; ... ;  $\phi_n$  ]’, where each  $\phi_i$  is a formula denotes the set of all the  $\phi_i$  that are true.
- An expression of the form ‘{  $x_1$   $x_2$  ...  $x_n$  :  $\phi$  }’, where the  $x_i$  are variables and  $\phi$  is a formula denotes the set of all tuples  $(a_1, a_2, \dots, a_n)$  of objects such that  $\phi$  is true if each occurrence of an  $x_i$  in  $\phi$  is replaced by  $a_i$ .

The current system has support for five aggregate functions:

**Cardinality:** The cardinality of a set is the number of elements in that set. The IDP syntax for the cardinality of a set  $S$  is ‘`card S`’ or ‘`# S`’.

**Sum:** Let  $S$  be a set of the second form, i.e., of the form ‘{  $x_1$   $x_2$  ...  $x_n$  :  $\phi$  }’, where  $x_1$  is a variable with an integer type. Then ‘`sum S`’ denotes the number

$$\sum_{(a_1, a_2, \dots, a_n) \in S} a_1,$$

i.e., it is the sum of the first arguments of all tuples in  $S$ .

**Product:** Similarly as for sum, one can write ‘`prod S`’ to denote

$$\prod_{(a_1, a_2, \dots, a_n) \in S} a_1,$$

i.e., the product of the first arguments of all tuples in  $S$ .

**Maximum:** One can write ‘`max S`’ to denote the maximum value of the first argument of a tuple in  $S$ , i.e.,

$$\max(\{a_1 \mid (a_1, a_2, \dots, a_n) \in S\}).$$

**Minimum:** To get the minimum value, write ‘`min S`’.

Below, we show several examples of aggregates.

Example 2.1: Cardinality

```
Given:
type Ship
type int Row
type int Col
type int Num
```

```

// The number of ship segments on
// each row and column
RowNum(Row) : Num
ColNum(Col) : Num

Find:
// The squares that contain a segment
HasShip(Row,Col)

Satisfying:

// The number assigned to the rows and columns must
// be equal to the number of segments on that
// row or column.
! r : RowNum(r) = #{ c : HasShip(r,c) }.
! c : ColNum(c) = #{ r : HasShip(r,c) }.

```

#### Example 2.2: Cardinality

```

Declare:
type Person
HasFever(Person)
LowHeartRate(Person)
LowBloodPressure(Person)
SevereSepsis(Person)

Satisfying:
// a person has severe sepsis if he has
// at least 2 of the symptoms
{ SevereSepsis(x) <-
  2 =< #[ HasFever(x); LowHeartRate(x);
        LowBloodPressure(x) ].
}

```

#### Example 2.3: Sum

```

Given:
type Course
type int Points
SP(Course) : Points

Find:
Selected(Course)

Satisfying:

```

```
// The sum of the number of points of all selected
// courses must lie between 60 and 70.
60 =< sum{ p c : Selected(c) & SP(c) = p} =< 70.
```

### 2.4.7 Partial functions

A normal function is total: it assigns an output value to each of its input values. On the other hand, *partial* functions do not necessarily have this property. In IDP, partial function  $F$  can arise in different situations. Either  $F$  is explicitly declared as partial function, or it is declared total, but its input types or output type are subtypes or integer types.

The semantics of a partial function  $F$  is given by transforming constraints and rules where  $F$  occurs as follows:

- in a *positive* context,  $P(\dots, F(x), \dots)$  is transformed to  $\forall y (F(x) = y \Rightarrow P(\dots, y, \dots))$ ;
- in a *negative* context,  $P(\dots, F(x), \dots)$  is transformed to  $\exists y (F(y) = x \wedge P(\dots, y, \dots))$ .

Here,  $P(\dots, F(x), \dots)$  occurs in a positive context if it occurs in a sentence and in the scope of an even number of negations, or if it occurs in a body of a rule and in the scope of an odd number of negations. All other occurrences are in a negative context.

```
Declare:
  type A
  type B isa A
  type int C = {1..10}
  partial F(A) : A
  G(B) : A
  P(A)
  Q(A)
  R(C)

Satisfying:

  ! x : P(F(x)). // meaning:
                 // ! x y : F(x) = y => P(y).

  { P(x) <- Q(F(x)). // meaning:
    // P(x) <- ? y : F(x) = y & Q(y).
  }

  ? x : P(x) & P(G(x)). // x is of type A.
                       // Hence G is partial in this sentence.
```

```

! x : C(x) => R(x+x) // '+' is partial, because
// x+x is not of type C for x > 5.
// Therefore, this sentence means
// ! x : C(x) => (! y : x+x=y => R(y)).

```

If a function  $F$  is explicitly declared partial, one can specify a domain for which it is total in the form of a formula  $\varphi$ . Then, for all input arguments for which  $\varphi$  is true, and only for these,  $F$  has an output. To declare the domain of a partial function, the keyword ‘domain’ can be used, as is shown in the following example.

```

Given:
type Ship
type Submarine
type Dir = { Hor; Ver }

partial ShipDir(s : Ship) : Dir
  domain ~Submarine(s)
// i.e. for every ship s that is not a submarine,
// ShipDir maps s to a direction.

```

## 2.4.8 Vocabulary Declarations

As mentioned above, symbols can not only be declared in vocabulary parts, but also in the problem description part. To do so, use the keyword ‘declare’, followed by the declaration and a dot. E.g.,

```

declare Neighbour(Row, Col, Row, Col).

```

To declare multiple symbols all at once, the following syntax can be used.

```

declare {
  HasShip(Row, Col)
  Neighbour(Row, Col, Row, Col)
}

```

Declaring symbols inside the problem description part is equivalent to declaring them in the general vocabulary part.

## 2.4.9 Implicit Constraints

Besides the constraints given by the user, the IDP system implicitly adds the following constraints:

- For each predicate declaration ‘ $P(T_1, \dots, T_n)$ ’ the constraint

```

! v_1 ... v_n : P(v_1, ..., v_n)
=> T_1(v_1) & ... & T_n(v_n).

```

This ensures that P is certainly false outside its domain.

- Similarly, for each (partial) function declaration ‘F(T<sub>1</sub>,...,T<sub>n</sub>) : T’ the constraint

```
! v_1 ... v_n : F(v_1, ..., v_n)=v
=> T_1(v_1) & ... & T_n(v_n) & T(v).
```

## 2.5 Optimal Solutions

Sometimes, one is interested in an *optimal* solution of a problem, rather than any solution. E.g., consider the problem of assigning tasks to several persons. Each of them has certain preferences for the tasks. An optimal solution should assign the tasks such that the preferences are taken into account as much as possible.

The IDP system allows two kinds of optimization statements. One where the value of a term with an integer type is minimized or maximized and another one where a subset-minimal or subset-maximal solution is searched. In each input, only one optimization statement is allowed.

### 2.5.1 Minimizing/Maximizing a Term

To specify that the IDP system should return a solution where a term *t* with integer type has a minimal value, add the header ‘Minimize:’, followed by that term. To maximize the value, add the header ‘Maximize:’.

```
Given:
type Person
type Task
type int Pref = {1..5}
type int Num

PrefTask(Person,Task) : Pref // the higher
// PrefTask(p,t), the more person p dislikes
// task t.

NrPersons : Num // the number of persons
NrTasks : Num // the number of tasks

Find:
Assign(Task) : Person

Satisfying:

// dont assign too many tasks to the same person
! p: #{ t: Assign(t) = p}
```

```
        =< (NrTasks/NrPersons + 1).

Minimize:

    // minimize the number of violated preferences
    sum{ pf p t: Assign(t) = p & PrefTask(p,t) = pf}
```

### 2.5.2 Subset Minimality/Maximality

To state that the system should return solutions where a set  $S$  is minimized or maximized according to the subset ordering  $\subseteq$ , use the header ‘Subsetminimize:’, respectively ‘Subsetmaximize:’, followed by  $S$ .

## 2.6 Problem Instance

### 2.6.1 Input Structure

A particular input to a problem can be given by giving an interpretation to all types and all predicate and function symbols that are declared given. Such an interpretation consists of

- for each type an enumeration of all objects of that type;
- for each predicate an enumeration of all tuples in the relation represented by that predicate;
- for each function an enumeration of the output for each input.

The enumeration starts with the header ‘Data:’.

#### Type Enumeration

The syntax for a type enumeration is

```
MyType = { E1_1; E1_2; ... ; E1_n }
```

where `MyType` is the name of the enumerated type and `E1_1; E1_2; ... ; E1_n` are the names of the objects of that type. Names of objects can be (positive and negative) integers, or start with an upper- or lowercase letter<sup>1</sup>. A non-integer type can have integers as elements, but an integer type can only have integers as elements.

<sup>1</sup>Note that if a type is enumerated in the given part, the syntax for the element names is more strict because they are also the names of constants.

### Predicate Enumeration

The syntax for enumerating all tuples for which a predicate `MyPred` with  $n$  arguments is true is as follows.

```
MyPred = { El_1_1, ..., El_1_n;  
          ... ;  
          El_m_1, ..., El_m_n  
        }
```

A predicate can only be enumerated after all types of its arguments are enumerated.

It is also possible to write parentheses around tuples.

```
MyPred = { (El_1_1, ..., El_1_n);  
          ... ;  
          (El_m_1, ..., El_m_n)  
        }
```

This notation makes it possible to state that a proposition (a predicate with no arguments) is true, by using an empty tuple.

```
True = { () }  
False = { }
```

### Function Enumeration

The syntax for enumerating a function `MyFunc` with  $n$  arguments is

```
MyFunc = { El_1_1, ..., El_1_n -> El_1;  
          ... ;  
          El_m_1, ..., El_m_n -> El_m  
        }
```

As for predicates, a function can only be enumerated if all types of its input arguments and the type of its output argument is enumerated before. To give the interpretation of a constant, one can simply write `MyConst = El` instead of `MyConst = { -> El }`.

### Shorthands

Shorthands like `MyType = {1..10; 15..20}` or `MyType = { a..e; A..E }` may be used for enumerating types or predicates with only one argument.

### Example

The following is a representation of the input for the battleship puzzle

```

Data:
// type enumerations
Ship = { Submarine1; Submarine2;
         Submarine3; Submarine4;
         Destroyer1; Destroyer2;
         Destroyer3; Cruiser1;
         Cruiser2; Battleship
        }
Row = {1..10}
Col = {1..10}
Num = {0..10}
Length = {1..4}

// function enumerations
ShipLength = {
  Submarine1 -> 1; Submarine2 -> 1;
  Submarine3 -> 1; Submarine4 -> 1;
  Destroyer1 -> 2; Destroyer2 -> 2 ;
  Destroyer3 -> 2; Cruiser1 -> 3;
  Cruiser2 -> 3; Battleship -> 4
}

RowNum = { 1 -> 2; 2 -> 1; 3 -> 5;
          4 -> 0; 5 -> 2; 6 -> 2;
          7 -> 1; 8 -> 4; 9 -> 1;
          10 -> 2;
        }

ColNum = { 1 -> 0; 2 -> 1; 3 -> 2;
          4 -> 3; 5 -> 2; 6 -> 1;
          7 -> 3; 8 -> 0; 9 -> 5;
          10 -> 1;
        }

```

## 2.6.2 Partial Input Structure

Besides a full interpretation of the symbols declared as given, one can specify partial interpretations for the other symbols. A partial interpretation for a predicate consists of an enumeration of tuples that are certainly in the relation, and tuples that are certainly not in the relation. All other tuples are unknown.

The syntax is illustrated in the following example.

```

Partial: // Header for partial input structure

/* Encode the hint for the battleship puzzle */

```

```

HasShip =
  // first specify the tuples that are
  // certainly in the relation ...
  { 2,3; 3,3 }
  // ... then the ones that are
  // certainly not in the relation
  { 1,3 }

```

Similarly, partial interpretations for functions can be specified.

## 2.7 Structuring the Specification

The IDP system allows a lot of freedom in structuring a specification. There is no fixed order for the different headers and each header can be used several times in a specification. Also, the specification can be spread over multiple files (see section 3.3.1). Basically, the only constraint is that every symbol must be declared before it can be used.

### 2.7.1 Symbol Specifications

One symbol can be declared multiple times. This will not produce an error message, provided the declarations are exactly the same. As soon as a symbol is declared inside an input vocabulary part, it belongs to the input vocabulary, and similarly for the output vocabulary. Instead of redeclaring a symbol, it can be referred to by a shorthand. For predicate symbols, the shorthand consists of the name of the predicate, followed by ‘/’ and the number of its arguments. For a function symbol, it consists of the name of the symbol, followed by ‘/’, the number of its input types, and a colon.

Using the above, a part of the declarations of the battleship puzzle could be as follows.

```

/*****
  Battleship puzzle
*****/

Declare:
  type Ship
  type Length
  type Row
  type Col
  type Direction
  ShipLength(Ship) : Length
  HasShip(Row, Col, Ship)
  ShipDir(Ship) : Direction
  Init(Ship, Row, Col)

```

```

Given:
  ShipLength/1:

Find:
  Init/3
  ShipDir/1:

```

Observe that for a type, it does not matter whether it is declared in the general or in the input vocabulary part.

## 2.8 Example

As an overview, the whole battleship specification is shown below.

```

/*****
  Battleship Puzzle
  *****/

/** The symbols and constraints **/

Declare:
  type {
    Ship
    int Row
    int Col
    int Num
    int Length
    Dir = { Hor; Ver }
  }

  RowNum(Row) : Num
  ColNum(Col) : Num

  ShipLen(Ship) : Length
  partial ShipDir(s:Ship) : Dir domain ShipLen(s) > 1

  HasShip(Row,Col)
  HasShip(Row,Col,Ship)
  Init(Row,Col,Ship)

  Neighbour(Row,Col,Row,Col)

Satisfying:

  // Each ship's initial segment is
  // at precisely one square

```

```

! s : ?1 r c : Init(r,c,s).

// Define where the ship's segments are in terms of
// its initial segment, length and direction
{ HasShip(r,c,s) <- Init(r,c,s).
  HasShip(r,c,s) <- Init(r,ci,s) &
    ShipDir(s) = Hor &
    ci < c < ci + ShipLen(s).
  HasShip(r,c,s) <- Init(ri,c,s) &
    ShipDir(s) = Ver &
    ri < r < ri + ShipLen(s).
}

// Define when a square contains a ship
{ HasShip(r,c) <- HasShip(r,c,s). }

// The row and column number
// should be respected
! r : RowNum(r) = #{ c : HasShip(r,c) }.
! c : ColNum(c) = #{ r : HasShip(r,c) }.

// Ships cannot touch each other
{ Neighbour(r1,c1,r2,c2) <- abs(r1 - r2) =< 1 &
  abs(c1-c2) =< 1. }

! r1 c1 r2 c2 s1 s2: ( Neighbour(r1,c1,r2,c2) &
  HasShip(r1,c1,s1) & HasShip(r2,c2,s2))
=> s1 = s2.

// Break symmetries
! s1 s2 r1 c1 r2 c2 : ( ShipLen(s1) = ShipLen(s2) &
  s1 < s2 & Init(r1,c1,s1) & Init(r2,c2,s2))
=> (r1 < r2 | (r1 = r2 & c1 < c2) ).

/** Input and output symbols */

Given:
  ShipLen/1:
  RowNum/1:
  ColNum/1:

Find:
  Init/3
  ShipDir/1:

```

```

/** A problem instance */

Data:

Ship = { Submarine1; Submarine2;
         Submarine3; Submarine4;
         Destroyer1; Destroyer2;
         Destroyer3; Cruiser1;
         Cruiser2; Battleship
       }
Row = {1..10}
Col = {1..10}
Num = {0..10}
Length = {1..4}

ShipLen = {
  Submarine1 -> 1; Submarine2 -> 1;
  Submarine3 -> 1; Submarine4 -> 1;
  Destroyer1 -> 2; Destroyer2 -> 2;
  Destroyer3 -> 2; Cruiser1 -> 3;
  Cruiser2 -> 3; Battleship -> 4
}

RowNum = { 1 -> 2; 2 -> 1; 3 -> 5;
          4 -> 0; 5 -> 2; 6 -> 2;
          7 -> 1; 8 -> 4; 9 -> 1;
          10 -> 2;
        }
ColNum = { 1 -> 0; 2 -> 1; 3 -> 2;
          4 -> 3; 5 -> 3; 6 -> 1;
          7 -> 3; 8 -> 0; 9 -> 6;
          10 -> 1;
        }

Partial:
HasShip = { 2,3; 3,3 } { 1,3 }

```

## 2.9 Obscure Features of IDP

Some obscure features of the system . . . .

### 2.9.1 Headers for Mathematicians

If you like to call a problem description ‘theory’, a problem instance ‘structure’, declarations ‘vocabulary’, etc., then the following alternative headers can be

used instead of the normal ones.

Normal	Alternative
Given:	%% inputvoc
Find:	%% outputvoc
Declare:	%% vocabulary
Satisfying:	%% theory
Data:	%% structure
Minimize:	%% minimize
Partial:	%% partial

### 2.9.2 Extra semicolons

In the problem instance part, it is allowed to write an extra semicolon after the last tuple of an enumeration of a type, predicate or function. E.g., it is allowed to write ‘ $P = \{A,B; C,D; A,C; \}$ ’ instead of ‘ $P = \{A,B; C,D; A,C \}$ ’. This feature can be of use when writing tools that automatically generate problem instances.

## Chapter 3

# Invoking the IDP System

In this chapter, the architecture of the IDP system and the different options are explained. For basic usage, only the first section is relevant.

### 3.1 The IDP interface

There is both a graphical and a command-line interface to the IDP system. To run the graphical version, open the jar file (`IDPx.y.jar`) using a java runtime as program (usually right-click the file, select "open with" and choose "java runtime") or, from command-line run

```
$ java -jar IDPx.y.jar
```

The command-line version has more options. Put the problem specification in `myspec.idp` and the data in `myinst.idp` and pass them to IDP as follows:

```
$ java -jar IDPx.y.jar -t=myspec.idp -d=myinst.idp
```

To learn about additional options, run

```
$ java -jar IDPx.y.jar --help
```

### 3.2 System Architecture

Before explaining the different options to control the behaviour of the system, we explain its architecture.

The IDP system does not directly solve the specification as given by the user, but first compiles it into an equivalent specification in the more simple *ECNF* format. This compilation is done by the *grounder* of the system, called *GIDL*. The *ECNF* specification can be orders of magnitude greater than the original IDP input. Next, a propositional *solver* is used to find solutions for the *ECNF* specification. The current<sup>1</sup> solver is called *MINISAT(ID)*. The solutions

---

<sup>1</sup>Previously, *MIDL* was the solver for IDP

it produces are not human readable. Finally, a translation tool translates the solutions of the solver into human readable form. It uses translation information that is produced by the grounder.

The IDP interface basically runs the three components one by one. I.e., it executes the following

```
$ gidl myspec.idp myinst.idp -o myecnf --transfile mytranslinfo
$ minisatid -nN myecnf mysolutions
$ translation_tool mysolutions mytranslinfo
```

The only option for MINISAT(ID) is the number of solutions it should search. How to control the behaviour of GIDL is described in the next section.

## 3.3 GIDL

GIDL admits various ways to structure the input into different files. Also, there are a lot of options to control its output.

### 3.3.1 Input Files

GIDL accepts an arbitrary number of input files and reads them in the order they are provided. If a file does not start with a header, it is read as if it started with a `Data` header. If no (implicit) `Data` header is encountered after reading all input files, GIDL expects input from stdin.

### 3.3.2 GIDL Options

#### Grounding options

- a Forces GIDL to not use aggregates to encode functions. This option is useful to create ground files for solvers without support for aggregates. Remark that if the input theory itself contains aggregates, the ground file will contain aggregates, even when this option is specified.
- noexp This option forces GIDL to ground all definitions, instead of calculating the models of the definitions which do not depend on open, non-given symbols.

#### Translation options

- t Do not add any translation information.
- transfile **my\_file** Write the translation information to the file **my\_file**.
- noarbit Do not add the list of arbitrary atoms. Specifying this option can lead to faster grounding, as in this case, GIDL does not keep track of which atoms occur in the ECNF output. All atoms that would be added to the list of arbitraries, will be interpreted by false.

## Approximation options

With the *approximation options*, the user can specify parameters for the approximation algorithm [8]. Choosing appropriate parameters can have a huge impact on the size of the grounding and the running time of GIDL. The default values are not guaranteed to produce good results in all cases.

- appruns= $N$**  For  $0 \leq N$ , this options sets ( $N \times$  the number of subformulas in the input theory ) as the maximum number of times the current bounds can be refined. This only affects the approximation phase. I.e., if  $N = 0$ , GIDL still does the bottom-up calculation. See below how to avoid both approximation and bottom-up calculation.
- maxapp= $N$**  This option specifies the maximum size of the bounds that are computed during the approximation phase and bottom-up calculation. The size of a bound is the number of internal nodes in the binary decision diagram used for the internal representation of the bound. By default,  $N = 4$ .
- alltotal** This options makes GIDL assume that all definitions in the input theory are total. In some cases, this can improve the applicability of the bounds, computed during the approximation phase. If it is not specified, GIDL assumes all definitions are non-total. In a next version of GIDL, a component might be added that automatically detects in some cases the totality of definitions.
- nosimp** During the approximation phase, several simplifications are applied to keep the bounds small. Some of these are dependent on the input structure for the problem. If the option **--nosimp** is specified, only simplifications that are independent of the input structure are tried.
- noapp** If this option is specified, there will be no approximation and no bottom-up calculation. The options setting the number of runs and size of the bounds have no effect in this case.

## Debug options

- data:** Show the internal representation of the theory and stop. This option is useful to see which types are assigned to the variables in the theory.
- debug:** Run the debugger. (Does not work yet)

## Semantics

- non-cautious:** This option changes the semantics of partial functions. In a positive context,  $P(\dots, F(x), \dots)$  is then transformed to  $\exists y (F(y) = y \wedge P(\dots, y, \dots))$  instead of  $\forall y (F(x) = y \Rightarrow P(\dots, y, \dots))$ . Vice versa in a negative context.

### Type derivation

A bunch of options to change the automatic derivation of types for variables. Try `gid1 --help` to see the explanation.

### Other options

**-W** Do not show warnings.

**-v** Print the version number and stop.

**-o my\_file** Write the grounding to `my_file`, instead of to stdout.

**--verbose** When this option is specified, several statistics are printed on stderr. Besides statistics about the size of the groundfile, the following are currently included.

**Reading specification:** Time needed to parse, check and transform the input.

**Creating tables:** Time needed to calculate the interpretations for some of the defined predicates.

**Approximation:** Time spent on the approximation phase.

**Grounding:** Time spent on actual grounding.

**Fixpoint reached:** If no, the approximation is stopped because the maximum number of refinements is reached. If yes, the most precise bounds, given the maximum size, is computed.

**Number of deletes:** The number of not executed refinements because of the maximum size of the bounds. Remark that when this is equal to 0 and a fixpoint is reached, more precise bounds cannot be obtained by the current implementation of the approximation algorithm. Hence, specifying more runs or a greater maximum size for the bounds will not lead to more precise bounds in this case.

**Extra simplifications:** Number of times a bound was simplified by reasoning about equality, arithmetic or the given domain.

**--help** Show a summary of the options and stop.

# Bibliography

- [1] Marc Denecker. Extending classical logic with inductive definitions. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *CL*, volume 1861 of *LNCS*, pages 703–717. Springer, 2000.
- [2] Herbert B. Enderton. *A Mathematical Introduction To Logic*. Academic Press, 1972.
- [3] Maarten Mariën, Johan Wittocx, and Marc Denecker. The IDP framework for declarative problem solving. In *Search and Logic: Answer Set Programming and SAT*, pages 19–34, 2006.
- [4] Maarten Mariën, Johan Wittocx, and Marc Denecker. Integrating inductive definitions in SAT. In Nachum Dershowitz and Andrei Voronkov, editors, *LPAR*, volume 4790 of *LNCS*, pages 378–392. Springer, 2007.
- [5] Maarten Mariën, Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In Hans Kleine Büning and Xishun Zhao, editors, *SAT*, volume 4996 of *LNCS*, pages 211–224. Springer, 2008.
- [6] David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 430–435. AAAI Press / The MIT Press, 2005.
- [7] David G. Mitchell, Eugenia Ternovska, Faraz Hach, and Raheleh Mohebali. Model expansion as a framework for modelling and solving search problems. Technical Report TR 2006-24, Simon Fraser University, Canada, 2006.
- [8] Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding with bounds. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 572–577. AAAI Press, 2008.
- [9] Johan Wittocx, Maarten Mariën, and Marc Denecker. The IDP system: a model expansion system for an extension of classical logic. In Marc Denecker, editor, *LaSh*, pages 153–165, 2008.

- [10] Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding FO and FO(ID) with bounds. *Journal of Artificial Intelligence Research*, 38:223–269, 2010.