Report:
Correctness Criteria for the Animation of Z Specifications via a Logic Programming Language
(Draft 2 September 2007)

M.M. West,
University of Huddersfield, Queensway, HD1 4DH, UK
email: M.M.West@hud.ac.uk,
WWW home page: `http://scom.hud.ac.uk/scommmw`

Abstract

This report presents techniques for the animation of Z utilising the Gödel logic programming language. The techniques are presented briefly and illustrated by a simple example. Correctness criteria for Z animations have been developed by other authors and these are applied to a logic programming language (the original was applied to a functional language). Formal arguments are presented which show that the animation obey the criteria for correctness. The report is based on the Ph D thesis [18], which it clarifies and updates.

1 Introduction

This report presents techniques for the animation of Z utilising the Gödel logic programming language. The techniques are presented briefly and illustrated by a simple example. Published criteria for correctness of an animation are compared and contrasted with the method of Abstract Interpretation (AI). In AI a concrete semantics is related to an approximate one that explicitly exhibits an underlying structure present in the richer concrete structure. In our case, the concrete semantics is Z associated with ZF set theory. The approximate semantics of the execution are the outputs of Z.

The criteria are applied to a logic programming language (the original was applied to a functional language). Formal arguments are presented which show that the structure simulation rules obey the criteria for correctness. The report is based on the Ph D thesis [18], which it clarifies and updates.

2 Z and Its Animation

The initial work in the development of the Z notation was at Oxford University in the early 1980's. Its designers' intention was for the major part of the notation to be 'conventional' first order logic and set theory. However the notation has a modular form: the data types, constraints on data types and the means of updating data types are grouped into *schema* which are composed using *schema calculus*. The Z Notation is widely used in Industry [2]. It has been used in the development of a Smartcard-based electronic cash system [15].

In a recent talk by Michael Jackson [10], he remarked that although it can be proved that a computer system is correct with respect to its formal specification, it is not possible to prove that the formal specification correctly models real world requirements. For this reason *animation* is proposed to aid validation of the specification. Animation involves execution of the specification and is a form of testing. Tests used for animation can seldom be exhaustive. However they can produce counter examples to a supposed relationship and can demonstrate whether some domain specific properties are present or absent. However for this to be useful, it is necessary that a specification be correctly represented by its animation. *Abstract Approximation* was suggested by Breuer and Bowen [3] to provide a formal framework and some proof rules for the correct animation of Z. In abstract approximation, the interpretation of Z syntactical objects in both the execution language (in our case the LP) and in Z are compared. For correctness, the interpretation in the LP domain must always underestimate the interpretation in the Z domain. This approach is unusual in that it is more common for a *program* to refine a specification rather than the converse. The reason for this more unusual approach can be expressed informally: an animation of a Z specification must not contain any more information than the original Z specification, as it may mislead. "Abstract Approximation" is similar to, but not the same as the established technique of "Abstract Interpretation" and these are compared later.

The declarative nature of Z means that the most natural choices for programming languages for animation of Z are also declarative. A small sample of examples is [6, 3, 16, 19, 17], involving both logic programming and functional languages. A further example of an animator is the *Alloy Analyser* [9] which simulates execution of *Alloy* - a Z like language[1]. Logic programming languages involve relations which require the input of one or more of its parameters and will return as output combinations of alternative value(s) of its other parameters. A function, in principle, returns a single value, however the value might be a tuple. Functions can return a partial answer, e.g. part of a list, thus allowing an unbounded list to represent an infinite set. Logic programming does not allow this, but does allow backtracking to provide the values one at a time. We believe that a logic programming language is preferable because queries of the "what if" variety can then be posed: given predicate `Pred(x,y,z,w)` the value(s) of `w` can be established where `x, y, z` are ground or (in principle) the

[1] See also http://sdg/lcs.mit.edu/alloy

value(s) of `x`, where `y, z, w` are ground. Other advantages include the use of meta-interpreters and techniques of inductive logic to investigate and explain the conflict between expected outputs and actual outputs. (For an example, see [12] for an Air Traffic Control Application.) The Z Notation is based on a *typed* version of ZF set theory. The 'sets as types' feature means that formulae in ZF containing expressions such as $\forall x \bullet \mathcal{A}(x)$ where $\mathcal{A}(x)$ is a wff appears (in Z) as $\forall x : \tau \bullet \mathcal{A}(x)$ where τ is some set valued object. Ordered pairs, such as (a, b), are usually derived from ZF as in $(a, b) = \{\{a\}, \{a, b\}\}$. However this would cause problems with incompatibility if a, b were of different types and for this reason ordered pairs are defined axiomatically. In a similar manner the natural numbers $\mathbb{N}$ are defined by the Peano axioms rather than as a definition arising from the infinity axiom. There follows a presentation of a fragment of the Z syntax [14][2]. An example which illustrates and augments is that of a small file system.

A Z specification commences with a sequence of paragraphs defining the basic types, global constants, schemas etc. The principle is of 'definition before use'. Basic types or 'given sets' are defined:

$$Paragraph \quad ::= \quad [Ident, \ldots, Ident]$$

Note that the integers are, by default, assumed as a basic type.

An example of a small file system involves a single given set of file identifiers *FileId*:

$$[FileId]$$

Axiomatic descriptions introduce global variables, with optional constraints:

$$\begin{array}{|l} Declarations \\ \hline Predicate;\ \ldots;\ Predicate \quad (optional) \end{array}$$

In the file system there are a maximum number of files *MaxFiles*, a non-zero integer:

$$\begin{array}{|l} MaxFiles : \mathbb{N} \\ \hline MaxFiles > 0 \end{array}$$

Further sets are defined by *Free Types*, which can be recursive (trees etc.) The simplest kind of free type can be used to define an enumerated set for example:

$$Dir ::= File_1 \mid File_2 \mid File_2$$

implying that $File_1$, $File_2$, $File_2$ exhaust the set *Dir* and are all distinct.
Schemas: the horizontal form is defined as follows:

$$Paragraph ::= Schema\_Name \mathrel{\widehat{=}} [\, Declaration \mid Predicate;\ \ldots;\ Predicate \,]$$

[2]For simplicity the optional parts will be indicated textually.

and the vertical form is:

Schema_Name

Declaration

Predicate; ...; *Predicate*

Note that schema names can also be referenced, and the referenced name *decorated*; the significance of this will be explained later:

$$Schema\_Ref ::= Schema\_NameDecoration$$

Declarations consist of basic declarations (including schema references) and sequences of basic declarations:

$$Basic\_Decl ::= Ident, \ldots : Expression \mid Schema\_Ref$$
$$Declaration ::= Basic\_Decl; \ldots; Basic\_Decl$$

Examples are $Files : \mathbb{F}\, FileId$ which indicate that the state variable *Files* is a finite subset of the set *FileId* , and $Count : 0 \,..\, MaxFiles$, which indicates that Count can take values between zero and *MaxFiles*. These form the declarations for the schema *FileSys*:

FileSys

$Files : \mathbb{F}\, FileId$

$Count : 0 \,..\, MaxFiles$

$\#Files = Count$

with predicate $\#Files = Count$ providing the relationship between state variables. If a predicate is missing, the default *true* is assumed.

Predicates and *Expressions* will be more fully defined later in the context of the rest of the example specification.

The schema denoting any change in state variables is $FileSys'$, where the 'prime' denotes a decoration:

$FileSys'$

$Files' : \mathbb{F}\, FileId$

$Count' : 0 \,..\, MaxFiles$

$\#Files' = Count'$

The 'primed schema name' has the effect that the variables of $FileSys'$ are primed and denote the values after some operation. Note that their types and predicate constraint are the same as for the unprimed version. Thus we can represent the operation of adding a file.

AddFID

$FileSys, FileSys'$
$NewFile? : FileId$

$Count < MaxFiles$
$NewFile? \notin Files$
$Files' = Files \cup \{NewFile?\}$
$Count' = Count + 1$

Predicates and *Expressions* are described next in a semi-formal way:
Predicates The simplest forms of predicates are equality ($=$), set membership ($\in$) and subset ($\subseteq$). In addition:

$$Predicate ::= Predicate \lor Predicate \mid Predicate \land Predicate$$
$$\mid \text{“}\forall Declaration \mid Predicate\text{”} \mid \text{“}\exists Declaration \mid Predicate\text{”}$$

Expressionsat their simplest can consist of an identifier. For simplicity we shall designate identifiers as follows: the set of basic and free types is denoted by the set *GIVEN*, the set of schema names is *NAME*, and the set of variable names (within a schema) is *VAR*. The sets named in *GIVEN* form the basic data types from which the typed sets are generated.

1. The integers, integer values and integer expressions involving arithmetic operations,
2. Set displays (such as $0 \,..\, MaxFiles$)
3. Power sets: if T_i is a typed set, $\mathbb{P}\ (T_i)$, whose type variables are subsets of T_i; (such as $\mathbb{F} : FileId$, the set of finite subsets of *FileId*),
4. Cartesian Product: T_i is a typed set then further typed sets can be defined recursively as $T_1 \times T_2 \ldots \times T_n$, whose type variables are tuples. Relations, functions etc are modelled by their graphs so for example if T_1, T_2 are typed sets then the set of binary relations between T_1, T_2 is

   $$T_1 \leftrightarrow T_2 == \mathbb{P}(T_1 \times T_2)$$

5. Sets formed from set operations such as union, intersection, distributed union etc.

Because we need to consider output of a schema - which is a set of bindings - we also extend the Z syntax so that they include *bindings*, for consider a schema $Sch \mathrel{\widehat=} [d \mid p]$ whose declaration d involves n variables named $x_i,\ i = 1 \,..\, n$ with their types. A *binding* provides values of x_i which satisfy p:

$$binding \quad ::= \quad < x_1 \Rrightarrow a_1, \ldots, x_n \Rrightarrow a_n >$$

3 Approach to Animation using a Logic Program

Previous work in the translation from Z to the logic programming language Prolog is [19] where the Assembler from Hayes [8] provided a case study. We called the method we developed 'structure simulation' and this was used in a real world application when Z and Prolog were used to help solve safety problems with Pelican equipment. Structure Simulation was later extended, using the Gödel Programming Language rather than Prolog. The Unix File System from Hayes was animated [17] and later an extended version of the Assembler [18].
Animation Rules: We advocate the Gödel programming language for animation in preference to the original animator, Prolog. Gödel is defined as a theory in first order logic - an implementation must be sound with respect to this semantics and calls to negative literals are ground. Gödel is strongly typed so that the sorts in Gödel can model the types of Z, and also a set data type is supported. Gödel has a flexible computation rule with user-defined control declarations. Further, the language is modular and modules can be exported (to modules) and can themselves import (other modules). The language is now briefly described.

A **predicate definition** consists of a declaration, specifying the type(s) of its arguments, and a set of statements of the form `Head <- Body.` where `<-` in Gödel means "if" and in contrast to Prolog, upper case is used for constants and lower case for variables. `Head` is an atom with the defining predicate and `Body` is a formula in first order logic and may be absent. Body can include first order constructs such as universal and existential quantification. Logical and is `&` and or is `or`. **Sets** in Gödel are implemented in the 'sets' module. Set membership is `In` and subset (`Subset`), set union `+` and intersection `*` . Sets terms can also be intensional (see [4] for the semantics). An example the definition of $\bigcup A$ where A is a set of sets and $\bigcup A$ gives the union of sets in A:

```
PREDICATE  DUnion:  Set(Set(a)) * Set(a).
% x is a set of sets and y is the distributed union of x
DUnion(x, y)  <- y = {z :  SOME [w] (w In x & z In w) }.
[Lib] <- DUnion({{1, 3}, {2, 3}, {5, 1}, {}}, x).
x = {1,2,3,5} ?
```

Sets are implemented in a manner based on finite set theory - the axioms of set theory are equivalent to the ZF axioms except for the Infinity Axiom. The sets module is used as a basis for the `Lib` module, a library of code which implements the Mathematical Toolkit.

Relations, functions are modelled using a type constructor, `OP` which allows the definition of `OrdPair` and hence of a *relation* between two sets.

```
CONSTRUCTOR OP/2.
FUNCTION    OrdPair : a * b ->  OP(a,b).

%%%%% declaration of partial function from  set a to set  b %%%%%
PREDICATE   PF  : Set(OP(a,b)) * Set(a)  * Set(b).
```

```
%%PF checks the set pf for functionality from s1 to s2
 PF(pf, s1, s2) <- ALL [x,y] (OrdPair(x,y) In pf
                  ->  (x In s1) & (y In s2)  &
               ALL [u] (OrdPair(x, u) In pf -> u = y)).

% query and answer: partial function is checked
[Lib] <- PF({OrdPair(1,2), OrdPair(3,2)}, {1,2,3}, {1,2,3}).
 Yes
```

Other features will be described as and when they are needed. The following provides an overview of the translation rules using the small file system.
Given Sets: The given sets of the specification are declared as one of the base types (in Gödel) and some constants of the base types introduced to model the environment (for example `F1, F2, F3`). Other base types include schema and variable names.
Variable bindings: A further BASE type is `BindVar`, used to facilitate the binding formation of schemas. `Bind1` achieves this for FileId. `BindVar` can be instantiated `Bind1(Files, {F1, F2})` or uninstantiated. `Bind1(Files, files)` where `files` is a program variable.
Schema bindings and schema decorations: An example of a schema is `FileSys` where schema and variable names are decorated via functions so $\Delta FileSys$, $FileSys'$, $Files'$ are denoted by `Del(FileSys)`, `DSch(FileSys)` and `DSet(Files)`.

```
% schema for state FileSys - head contains a schema binding
% the ordering in the list reflects the order provided by the user.
SchemaType( [ Bind1(Files, files ), Bind2(Count, count )], FileSys)
  <- setFID = {x : IsFileId(x) } & %% schema predicate
    files Subset setFID & count In {y : 0 =< y =< 10} &
    Card(files, count).

%  AddFID - includes declarations of  FileSys, Del(FileSys)
SchemaType(binding, AddFID ) <- SchemaType(binding1,  Del(FileSys)) &
   binding1 = [Bind1(Files, files ), Bind2(Count, count ),
     Bind1(DSet(Files), files1 ), Bind2(DSet(Count), count1 )]  &
      Append(binding1, [ Bind3(IN(NewFile), newfile)] , binding) &
     count < 10 &      setFID = {x : IsFileId(x) } &
     newfile In  setFID &   ~ ( newfile In files) &
    files1 = files + {newfile} &      count1 =  count +   1 .
```

Answer sets: If we query schemas *AddFID* the result is an output *answer set* of schema bindings. The schema declarations for AddFID are such that all possible subsets of `setFID` are generated and the predicate checks their values so that all states are eventually generated and the possible inputs which are associated. This represents an extreme - and would not occur if (for example) more complex data types are used such as partial function.

```
% test of schema AddFID.
[Demo2] <- SchemaType(b, AddFID).
 % two (of many) possible schema bindings
b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F1}),
           Bind2(DSet(Count),1),Bind3(IN(NewFile),F1)] ? ;
```

```
b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F2}),
          Bind2(DSet(Count),1),Bind3(IN(NewFile),F2)] ? ;
...
```

UnDef (below) is a contrived example of a schema which outputs some results then fails with an error message.

UnDef

$$X, Y : \mathbb{N}$$

$$Y \in \{1, 2, 3\}$$
$$((X = 1) \lor (X = 3) \lor (\{X, 1, 2, 3\} = \{1, 2, 3, 4\}))$$

This should result in the set of bindings:

$$\{< X \Rrightarrow 1, Y \Rrightarrow 1 >, < X \Rrightarrow 2, Y \Rrightarrow 1 >, < X \Rrightarrow 4, Y \Rrightarrow 1 >,$$
$$< X \Rrightarrow 1, Y \Rrightarrow 2 >, \ldots < X \Rrightarrow 4, Y \Rrightarrow 3 >\}$$

When animated this results in:

```
[Demo2] <- SchemaType( [ Bind2(X, x ), Bind2(Y, y) ], UnDef).
 x = 1,  y = 1 ? ;
 x = 3,  y = 1 ? ;
 Floundered.  Unsolved goals are:
 Goal: {v_1,1,2,3}={1,2,3,4}
 Delayed on: v_1
```

The above is an example of an incomplete answer set - where the output depends on the way the query is evaluated (generally it echoes the code order).

Coverage - Case Studies The strategy for animation is to *test* whether a term satisfies the conditions for it to be a partial function (as in the example above) rather than to generate all possible values from two sets. This is to prevent the data generated growing exponentially with respect to the size of the given sets. With this proviso, the library currently covers all of the structures in the Toolkit - except for the formation of transitive closure of a relation. Usage of various structures in the case studies from Hayes [8] is described next.

Assembler: The Assembler includes relations, functions, sequences, composition of relations and and structures such as domain and range restriction and subtraction. As explained previously, we believe that the structure of the animation should reflect the schema structuring, so each schema is modelled separately, then composed with another to model (for example) schema conjunction. This contrasts with other work in this field as described in section 4. The 'basic' Assembler was presented in [19] where assembly contexts were modelled and treated as if they were declared in the signature of schema *Assembly*. The animation checked that sequences of assembly and machine instructions adhered to the constraints. This was extended in the thesis [18] and the *Implementation* schema was also modelled as a two phase assembler via schemas $Phase1, Phase2$. In Hayes *Implementation* is expanded out and it is proved that it is the same

Syntactical Element	Covered	Case Study	Comment
Relations and Functions	Yes	Assembler and Unix File System	Not Transitive Closure of Relation
Domain and range restriction and subtraction	Yes	Assembler and Unix File System	
Sequences	Yes	Assembler	
Bags	No		
Schema Disjunction, Conjunction	Yes	Assembler and Unix File System	
Binding formation $\theta schema$	Yes	Unix File System	
$\exists\, Schema$	Yes	Assembler	Requires Expansion
$\forall\, Schema$	No		
Generic Schemas	No		

Table 1: Coverage of Z Syntax

as *Assembly* where $Implementation \mathrel{\widehat=} Phase1 \land Phase2 \setminus (st, rt, core)$. Variables st, rt are tables and the sequence *core* can be thought of as acting as a 'place-holder' for the output machine sequence. The animation was able to demonstrate this, but only the values st, rt could be hidden (and derived); a value of *core* was necessary for the computation.

Unix File System: The structures used included functions as for the assembler. A schema *CHAN* represents a file and a current position in the file. A channel storage system *cstore* denotes a partial function between channel identifiers (the set *CID*) and *CHAN*: $cstore : CID \pfun CHAN$ and *cstore* is updated by the opening of a a new channel: $cstore' = cstore \oplus \{cid! \mapsto \theta CHAN\}$. A query will provide possible values of $cstore'$, given the value of *cstore* - so it is unlike the Assembler which mainly checks values. Table 1 indicates syntactical elements which are covered by the rules and the case studies from Hayes [8] (if any) which they occur in.

4 Comparison with other Work

This section compares our work with a small sample of related work. The SuZan project is described in [6], in which a subset of Z is animated in Prolog. The principal difference between our technique and that of the SuZan project is the use of predicates in the schema signature to generate data so that *Schema* is used as a means of a *generate and test* cycle. The signature type constructor predicates are coded in a manner which generate values (of a function for example) from instantiated given sets, and these values are subsequently tested for conformance with *Predicate*. However the data generated is liable to grow exponentially with respect to the size of the given sets, so the researchers have provided the execution process with forms of control (such as 'unfolding'). The method is unwieldy (as it does not allow extra-logical features such as Prolog

'cut') but it does allow a wide range of 'what if' queries. The use of 'pure' Prolog is to try to ensure that the animation correctly represents the Z.

The Jaza animator [16] uses Haskell for animation purposes - and covers most of the Toolkit. However before translation takes place, each expression involving schema calculus must be expanded out into a full schema. There are a variety of data structures which represent sets, some of which are similar to the Gödel representation. However it also includes set comprehensions which are *not* expanded out. Functions are also represented by a special kind of set which include the domain and range sets plus boolean flags. The author reports that possible combinatorial explosions have been overcome so far with devices such as coercion functions and a query mechanism. A table is provided which compares coverage and correctness of Z animators - including Jaza, however no systematic proof of correctness of Jaza animation is provided.

In contrast, the Miranda implementation provided by [3] is accompanied with a proof of correctness. However the implementation does not cover given sets. Also, any schema references, expressions etc. are expanded and absorbed into the schemas which reference them. Whereas in our approach they are treated in a modular fashion.

5 The Application of Abstract Approximation

This section presents a formal framework for establishing the correctness of the animations described in section 2. The correctness criteria chosen was *Abstract Approximation* and this is described in rest of this paper, together with a demonstration of correctness of 'structure simulation'.

Abstract approximation was suggested by Breuer and Bowen [3] to provide a formal framework and some proof rules for the correct animation of Z. In abstract approximation, the interpretation of Z syntactical objects in both the execution language (in our case the LP) and in Z are compared. The Z or 'concrete' interpretation is the interpretation we would expect if we had been evaluating the objects using set theoretic (ZF) considerations and the logic programming domain D_{LP} is the 'abstract domain'. The comparison is in an "extended Z domain" into which the two semantic domains are injected. The two evaluations are compared in 'equivalent' environments and the comparison

is in the Z domain. This is illustrated in Figure 1.

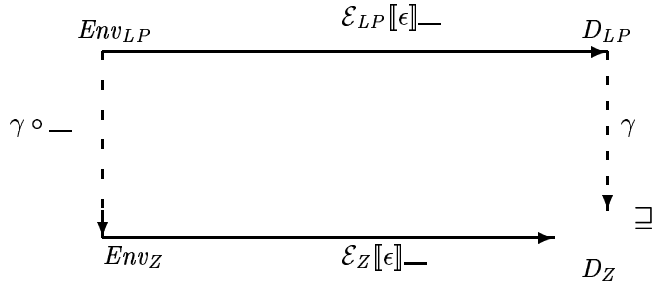

Figure 1: Approximation Diagram for LP and Z domains

A concretisation function γ relates the abstract with the concrete. The approximation expresses the underlying concept of 'safeness' in abstract approximation, that a computation in D_{LP} should never provide more information than the result obtained by the evaluation of an expression in Z. This is in order that no incorrect information is output. In order to accommodate non-terminating executions, such as integer overflow, both domains are extended by the inclusion of a '$\perp$' element for each type. The appropriate domain (or sub-domain) for each bottom element will, in the main, be understood by its context. Exceptions are indicated as they occur.

The method has similarities to *abstract interpretation* [5] which was initially used for static analysis of imperative programs. However in abstract interpretation, the sets of values are abstracted by set *descriptors* (e.g. 'odd', 'even' for integers) whereas in abstract approximation sets (in Z) are abstracted by sets in the animation - although they may be less well defined.

Recalling that the set of variable names (within a schema) are VAR, $Env_{LP} == VAR \twoheadrightarrow D_{LP}$, is the set of all possible LP environments associated with a specification and each syntactic expression in Z is evaluated in an LP environment $\rho_{LP} \in Env_{LP}$. Similarly $\rho_Z \in Env_Z$ where $Env_Z == VAR \twoheadrightarrow D_Z$. The two environments are related by the concretisation function $\gamma : D_{LP} \twoheadrightarrow D_Z$, so that $\rho_Z = \gamma \circ \rho_{LP}$. The ordering derives from both the possibility of non-termination of the execution and variable values remaining undefined. It is in respect of all types of domain elements:

$$a \sqsubseteq b \Leftrightarrow (a = \perp \ or \ a = b).$$

The ordering relation works co-ordinatewise on tuples and ordering on sets corresponds to two standard 'powerdomain' orderings. (See also [7] for a complete treatment.) The first is where sets contain incomplete elements and can be

expressed formally:

$$D_1 \sqsubseteq D_2 \Leftrightarrow (\forall\, d_1 : D_1 \bullet \exists\, d_2 : D_2 \bullet d_1 \sqsubseteq d_2) \wedge (\forall\, d_2 : D_2 \bullet \exists\, d_1 : D_1 \bullet d_2 \sqsupseteq d_1).$$

For example, $\{1,2,3,\bot,4\} \sqsubseteq \{1,2,3,4,5\}$ and so it is *not* the same as subset ordering. The second is where sets are 'incomplete'. The notation is to 'tag' them, for example $\{1,2,3,4\}_{\cup\bot}$. The ordering for 'incomplete sets' is as follows:

$$(D_1)_{\cup\bot} \sqsubseteq D_2 \Leftrightarrow (D_1)_{\cup\bot} \sqsubseteq (D_2)_{\cup\bot} \Leftrightarrow (\forall\, d_1 : D_1 \bullet \exists\, d_2 : D_2 \bullet d_1 \sqsubseteq d_2).$$

We 'refine' an incomplete set if we complete it or (in addition) we add some more elements: For example, $\{1,2,3,4\}_{\cup\bot} \sqsubseteq \{1,2,3,\bot,4,5\}$. The ordering for incomplete sets is a 'pre-order' for if s,t are incomplete then $s \sqsubseteq t$ and $t \sqsubseteq s$ does not imply that they are equal. An example can be obtained by comparing $\{1,2,3,4\}_{\cup\bot}$ and $\{1,2,3,\bot,4\}_{\cup\bot}$. In [3], incomplete sets were implemented by Miranda in the output of 'lazy lists' to represent a partially defined set. The rule for correct approximation is presented next. The abstract (programming) interpretation of Z syntax is denoted: $\mathcal{E}_{LP}[\![\ldots]\!]\rho_{LP}$ and the Z interpretation is denoted $\mathcal{E}_Z[\![\ldots]\!]\rho_Z$. The rule for approximation represents the fact that if ϵ is a syntactic Z expression then the following condition must hold for a correct animation of Z in D_{LP}:

Approximation Rule 1 (AR1)

$$\gamma(\mathcal{E}_{LP}[\![\epsilon]\!]\rho_{LP}) \sqsubseteq \mathcal{E}_Z[\![\epsilon]\!](\gamma \circ \rho_{LP}).$$

The following conditions form the basis of a structural induction rule in which if it can be shown that **AR1** holds for syntactic variable $\epsilon = x$, then it also holds for syntactical expression $\epsilon = fx$. For example, f might be the syntactic operator '$\cup$' on variable tuple $\epsilon = (x_1, x_2)$. We denote by f_Z, f_{LP} the interpretation in the Z domain and LP domain respectively of the syntactic expression fx. Thus if fx is set union, then $f_Z x$ is the set theoretic evaluation of set union and $f_{LP} x$ is the induced operation in D_{LP} of set union. In order to show **AR1**, the following must hold for the operators f of Z on variables x:

Condition 1 In order to prove correctness it is necessary to show that the interpretation in D_{LP} is built recursively for each operator of Z, acting on each syntactic Z expression.

$$f_{LP}(\mathcal{E}_{LP}[\![x]\!]\rho_{LP}) = \mathcal{E}_{LP}[\![fx]\!]\rho_{LP}$$

Condition 2 A further condition is a property of Z, i.e. the manner in which expressions in the Z domain are evaluated.

$$f_Z(\mathcal{E}_Z[\![x]\!]\rho_Z) = \mathcal{E}_Z[\![fx]\!]\rho_Z.$$

However this condition is only true for complete sets and is not in general true for incomplete sets.

Condition 3 The third condition is the key one, which encapsulates the approximating mechanism:

$$\gamma(f_{LP}(\mathcal{E}_{LP}[\![x]\!]\rho_{LP})) \sqsubseteq f_Z(\gamma(\mathcal{E}_{LP}[\![x]\!]\rho_{LP})).$$

Conditions 1–3 provide the basis of a structural induction rule:
Structural Rule for Induction:
If **Conditions 1–3** hold for all $\rho_{LP} : VAR \twoheadrightarrow D_{LP}, x \ : \ \Sigma$, then **AR1** for $\epsilon = x$ implies **AR1** $\epsilon = fx$.
PROOF

$\gamma(\mathcal{E}_{LP}[\![x]\!]\rho_{LP}) \sqsubseteq \mathcal{E}_Z[\![x]\!](\gamma \circ \rho_{LP})$ [Base Case]
$f_Z(\gamma(\mathcal{E}_{LP}[\![x]\!]\rho_{LP})) \sqsubseteq f_Z(\mathcal{E}_Z[\![x]\!](\gamma \circ \rho_{LP}))$
[f_Z Monotone for environment with complete sets]
$\gamma(f_{LP}(\mathcal{E}_{LP}[\![x]\!]\rho_{LP})) \sqsubseteq f_Z(\mathcal{E}_Z[\![x]\!](\gamma \circ \rho_{LP}))$ [cond. 3, $\sqsubseteq$ transitive]
$\gamma(f_{LP}(\mathcal{E}_{LP}[\![x]\!]\rho_{LP})) \sqsubseteq \mathcal{E}_Z[\![fx]\!](\gamma \circ \rho_{LP})$ [cond. 2, $\rho_Z = \gamma \circ \rho_{LP}$]
$\gamma(\mathcal{E}_{LP}[\![fx]\!]\rho_{LP}) \sqsubseteq \mathcal{E}_Z[\![fx]\!](\gamma \circ \rho_{LP})$ [cond. 1]
□

The base types for induction include integers, instantiations of given sets, sets of integers and variables. However since **Condition 2** is only true for complete sets, then **AR1** can only be used for complete sets. In order to encompass incomplete sets, we need to extend ZF operations. A further induction rule is presented, as in [3]: **AR2** is implied by **AR1**, provided that we interpret f_Z for incomplete sets in such a way that is is monotonic in the refinement relation for incomplete sets.
Approximation Rule 2 (AR2)
Recall $\rho_{LP} : VAR \twoheadrightarrow D_{LP}$ is an environment in the execution domain D_{LP}, then $\gamma \circ \rho_{LP} : VAR \twoheadrightarrow D_Z$ is an environment in D_Z and write $\rho_Z = \gamma \circ \rho_{LP}$. Consider ρ'_Z environment in D_Z which refines ρ_Z, viz. $\rho_Z \sqsubseteq \rho'_Z$. Then **AR2** is:

$$\rho_Z \sqsubseteq \rho'_Z \Rightarrow \gamma(\mathcal{E}_{LP}[\![\epsilon]\!]\rho_{LP}) \sqsubseteq \mathcal{E}_Z[\![\epsilon]\!]\rho'_Z.$$

AR2 is stronger than **AR1**, for if we take $\rho_Z = \rho'_Z$, then **AR2** becomes **AR1**. Conversely, assuming the monotonicity of f_Z, then **AR1** implies **AR2**.
If **Conditions 1–3** hold for all $\rho_{LP} : VAR \twoheadrightarrow D_{LP}, x \ : \ \Sigma$, and f_Z is monotone, then **AR2** for $\epsilon = x$ implies **AR2** for $\epsilon = fx$.

6 Comparison of Abstract Interpretation and Abstract Approximation

The resemblances between abstract interpretation and abstract approximation can be seen in Figure 2, which contains the approximation diagrams for each

concept.

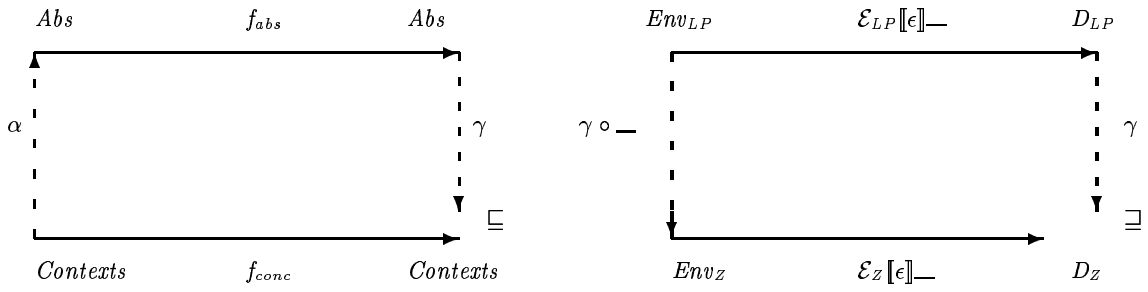

Figure 2: (i) Abstract Interpretation and (ii) Abstract Approximation

Reference [5] describes the use of abstract interpretation in static analysis of imperative programs and diagram **(i)** of Figure 2 pictures the abstract interpretation of program environments at a program node. The abstractions are of the contexts associated with a program node. The figure indicates the loss of information when concretising the result of an abstract interpretation. Diagram **(ii)** of Figure 2 represents loss of information when interpreting a piece of Z syntax in an abstract execution domain as compared with the concrete 'Z' domain.

Diagrams **(i)** and **(ii)** are similar in that they both represent an abstract and concrete interpretation of a piece of syntax. However for abstract interpretation, the abstraction is a *set descriptor*, whereas for abstract approximation integers, sets, tuples in the abstract correspond to integers, sets, tuples in the concrete. However the abstract object may not be as well defined as the associated concrete object. Abstract approximation also represents loss of information, for the reason that a program may terminate, or only provide a partial answer. There is a difference, too in the manner in which both represent lack of information. In abstract interpretation the *top* element corresponds to the *least* precise information, the set $\mathbb{Z}$. The *most* precise information is given by the empty set, and corresponds to $\perp$. The abstract interpretation is an *upper* approximation. As pointed out by [11], the ordering is opposite to the ordering of domain theory; the *top* element corresponds to total lack of information. Abstract approximation has the ordering of domain theory; the *bottom* element corresponds to total lack of information and it is a *lower* approximation.

Abstract Interpretation involves two comparisons. First of all the concrete context inputs are abstracted (via α) and an abstract interpretation of a language construct performed. The resulting abstraction is concreted (via γ) then compared with the concrete interpretation of the context and a loss of information is found. The second comparison involves commencing with the abstract contexts on input arcs, concretising, via γ then performing the concrete interpretation. The result is the same as if an abstract interpretation had been

directly performed: the second comparison results in no loss of information. On the other hand abstract approximation explicitly involves only *one* comparison: a syntactic object is interpreted in an environment in the abstract (execution) domain and the value concreted (via γ). This value is compared to that found by concreting an execution environment and interpreting the same syntactic object in the resulting concrete environment in a concrete (Z) domain. In that case the first result underestimates the second. For abstract approximation the notion of 'safeness' is that the abstract approximation will always provide correct information. However there may be little or no information provided if the program fails to terminate. The principle of safeness means that we do not want to output the *wrong* information, for it may mislead.

The remainder of this report is restricted to the application of the generics of abstract approximation to the logic programming domain. Although the examples supplied are in Gödel, the framework is intended to apply to any logic programming language with sound semantics and with sets and types.

7 Formalising Structure Simulation

This section formalises the translation of Z syntax to a logic program. The assumption is that the specification has been translated to a logic program and that the user queries this program, as in the case of the simple example and case studies of [19, 17]. Section 7.1 presents an overview of the animation approach in initialising and querying the specification. Section 7.2 describes the representation of the expressions and sets of Z in the LP domain and Section 7.3 defines the concretisation function γ in mapping between the abstract and concrete domains.

7.1 Overview

The user of the animator supplies some values for the constants and sets of the specification, followed by an initial imposed environment ρ^o. If the initial environment is consistent with the constraints of the specification, an answer set is output which provides a set of values for the other schema variables.

A schema may have other schemas as references or be defined in terms of schema expressions (schema conjunction for example). A top level call to a schema will provide the answer substitution set, which models the appropriate set of schema bindings, where a single binding is of the form:

$$< x_1 \Rrightarrow a_1, \ldots, x_n \Rrightarrow a_n >$$

or

$$[Bind_1(X_1, a_1), \ldots, Bind_n(X_n, a_n)]$$

in the LP. The characteristic predicate for the schema (in the LP) is from the *interpreted* syntax of the schema definition and includes syntactical objects such as

set expressions, predicates, declarations and references to other schemas. Suppose $Env_{LP} == VAR \twoheadrightarrow D_{LP}$ is the set of all possible environments associated with a specification, then $\rho^o_{LP} \in Env_{LP}$ for the characteristic predicates to succeed. The user is then supplied with the set of answer substitutions satisfying each schema, which will have been constrained by its environment $\rho^o_{LP} \in Env_{LP}$. At the end of the execution the environment ρ^o_{LP} will have been enhanced to one of a set of environments $\rho_{LP} \in Env_{LP}$ and each answer substitution will conform to ρ_{LP}.

Note that for terminating computations of expressions we are interested *either* in a value which terminates with success (and there may be many such values contributing to many schema bindings) *or* in a finite failure. In Nicholson et al. [13], the continuation semantics of Prolog is presented. However this is not our approach – we are not interested in the computational processes (for example backtracking) which accompanies such a search. We assume that some variables are initially instantiated, and some are initially undefined and are subsequently computed. (Whereas Nicholson et al. are concerned with the possible *changing* values of variables as the program executes.) Non-termination or floundering occurs when it is not possible to compute the undefined values.

This section describes how the integers, given sets and derived expressions of Z are abstracted by the *declarative semantics* of a logic programming language. The integers, $\mathbb{Z}$, and instantiated given sets G^k form the basis of domains D_{LP} and D_Z. Syntactic expressions of Z considered are schemas, predicates and expressions such as arithmetical and set expressions. The *output* is confined to schema bindings.

Thus evaluations of arithmetical, set and expressions *other than* these, take place *as part of a program execution to determine or check schema bindings.* A description is presented in the next subsection, of how the Z syntax is interpreted in the LP, and of how the outputs can be ordered.

7.2 The Logic Programming Domain

Recall that the given sets are denoted $GIVEN$, the set of schema names $NAME$, and the set of variable names (within a schema) are VAR. The variable and schema names will subsequently be interpreted as constants in D_{LP}, which means confining them to allowed constant names in the programming language. However, in most of what follows, $x_1, \ldots x_n$ etc. will be used to 'stand for' the variable names.

The proposed abstract domain, D_{LP}, includes representations of integer values, instantiated values, tuples, bindings and sets. n-Tuples are represented by functions of arity n and sets are represented both as *terms* and as answer sets. $GIVEN$ is captured by declaring $\{G^1, \ldots, G^N\}$ as bases of the program and for each base G^k is declared the constants $g^k_1, \ldots g^k_n$. In order to 'collect together' the constants to form a set, for each base a predicate is constructed denoting membership. For example predicate `IsFileId` is applied thus: `IsFileId(F1) IsFileId(F2)` etc. The *set* of file identifiers is formed by set comprehension.

In the execution domain D_{LP}, the refinement ordering is generated by:

$$\forall\, x : D_{LP} \bullet \bot \sqsubseteq x$$

and by recursion on the representation of sets and tuples. The partial element(s) $\bot$ of D_{LP} represents undefined or incomplete element(s) of the various types. The formalisation involves *set terms* as previously described. It also involves *sets of answer (substitutions)*, where the latter are only considered in the case of schema outputs.

7.2.1 Set Objects in the LP

During execution, one or more computations may fail to terminate and this has the following effect on set objects:

1. Set terms: recall that (finite) set terms are represented by

   $$\{a_1, a_2, \ldots, a_n\}, \text{ or } a_1 \circ (\ldots (a_n \circ \varnothing))))$$

   where each a_i is itself a term. The following set contains an incomplete element:

   $$(\bot \circ (a_1 \circ (\ldots (a_n \circ \varnothing))))$$

   and when a computation of a set term fails to terminate, in an attempt to evaluate an infinite set for example, we obtain:

   $$(a_1 \circ (\ldots (a_n \circ \bot)))$$

   In both cases the set evaluates to '$\bot$' since functions are strict. This bottom element is designated $Null_\bot$ to distinguish it as a set and the equivalent of $\varnothing_{\cup\bot}$ in D_Z. We have, for all set a:

   $$a \cup (Null_\bot) = (Null_\bot) \cup a = Null_\bot$$
   $$a \cap (Null_\bot) = (Null_\bot) \cap a = Null_\bot$$

2. Sets of answer substitutions: recall that for some schema Sch, a binding is denoted in the LP:

   $$\theta Sch = [Bind_1(X_1, a_1), \ldots, Bind_n(X_n, a_n)]$$
   $$\text{where } a_k \in D_{LP}, X_i \in VAR, Sch \in NAME.$$

   and that the answers to a query concerning the characteristic predicate of Sch provide a subset of

   $$\{Sch \bullet \theta Sch\}$$

   which depends on the values instantiated. However an answer set can output some results then fail with an error message. An example would be a schema with

```
(x = 1) \/ ({x, 1, 2, 3} = {1, 2, 3, 4})
```

in the predicate. Whether the answer set contains some or indeterminate answers depends on the way it is evaluated (generally it echoes the code order). Thus there is no way of knowing, from the output, the nature of the rest of the set. This set is an example of the incomplete output set of schema *UnDef* where, if $b_i, (i = 1 \ldots k)$ is a schema binding the incomplete set of answers can be denoted

$$\{b_1, \ldots b_k\}_{\cup \perp}$$

where no more answers are provided after the k^{th} which is followed by the output of an error message, as in the example.

Figure 3 contains the abstract (or LP) representation of Z expressions which is defined recursively via *terms* in the logic programming language.

$D_{LP} ::= m, m$ *an integer*
$\quad | \; g_i^k,$ where each g_i^k is base G^k
$\quad | \; T_n(a_1, \ldots, a_n)$ where $a_k \in D_{LP},$ a tuple
$\quad | \; \{a_1, \ldots, a_n\}$ where $a_k \in D_{LP},$ enumerated free type
$\quad | \; \{a_1, \ldots, a_n\}$ where $a_k \in D_{LP}, a_k \neq \perp,$ an complete set *term*
$\quad | \; \{a_1, \ldots, \perp, \ldots a_n\} (= Null_\perp)$ where $a_k \in D_{LP}$
$\quad | \; \{a_1, \ldots, a_n\}_{\cup \perp} (= Null_\perp)$ where $a_k \in D_{LP},$
$\quad | \; Bind_i(X_i, a_i)$ where $a_i \in D_{LP}, X_i \in VAR$
$\qquad$ a single variable binding
$\quad | \; [\; Bind_1(X_1, a_1), \ldots, Bind_n(X_n, a_n) \;],$ where $a_k \in D_{LP}, X_i \in VAR,$
$\qquad$ a schema type $--$ a single schema binding

Figure 3: The Interpretation of Expressions in the LP Domain

D_{LP} is supplemented by the following answer set for a schema, of complete and incomplete schema bindings:

$$D_{LP} ::= \{b_1 \ldots b_n\} \mid \{b_1 \ldots b_n\}_{\cup \perp}$$

where each b_i is of the form:

$$[\; Bind_1(X_1, a_1^i), \ldots, Bind_N(X_n, a_n^i) \;]$$

In addition, for each base type G^k there is a unary predicate IsG^k which is applied to each constant, g_i^k, of the base[3]. Bindings of variable names to values, $Bind_i(X_i, a_i)$ provide an environment, and also interpret schema types, as described.

[3]Where the meaning is apparent we shall in future remove super and subscript from instantiated elements and given sets and use g, G, respectively.

7.3 Concretisation Function γ

Expressions of D_{LP} are mapped to expressions of D_Z, and predicates of D_{LP} to predicates of D_Z. $\perp$ of D_{LP} maps to $\perp$ of D_Z. A concretisation function $\gamma : D_{LP} \rightarrow D_Z$ is constructed which maps to D_Z from an abstract domain D_{LP} recursively as follows: integers in D_{LP} map to integers in D_Z, instantiated values of given sets map to instantiated values in D_Z, set terms map to sets, and functions, $T_n(a_1, a_2, \ldots, a_n)$, map to n-tuples. If a_i is a term in the LP and $X_i \in VAR$, G a 'typical' base type representing a given set and g a 'typical' member. The mapping γ for *terms* is defined in Figure 4.

$$
\begin{array}{lcl}
\gamma(m) & = & m, m \text{ an integer} \\
\gamma(g) & = & g, \text{ constant of base type } G \text{ is} \\
 & & \text{mapped to instantiated element } g \\
\gamma(\{a_1, \ldots, a_n\}) & = & \{\gamma(a_1), \ldots, \gamma(a_n)\}, \\
\gamma(\perp \circ (a_1 \circ (\ldots (a_n \circ \varnothing)))) & = & \gamma(Null_\perp) = \varnothing_{\cup\perp} \\
\gamma(a_1 \circ (\ldots (a_n \circ \perp))) & = & \gamma(Null_\perp) = \varnothing_{\cup\perp} \\
\gamma(T_n(a_1, \ldots, a_n)) & = & T_n(\gamma(a_1), \ldots, \gamma(a_n)), \\
\gamma([Bind_1(X_1, a_1), \ldots, Bind_n(X_n, a_n)]) & = & \{X_1 \mapsto \gamma(a_1), \ldots, X_n \mapsto \gamma(a_n)\} \\
 & & \text{a single schema binding} \\
\gamma(\perp) & = & \perp
\end{array}
$$

Figure 4: γ: LP terms

We now define how γ maps the *answer substitutions* which model schema bindings. A complete (incomplete) set of schema bindings maps to a complete (incomplete) set of schema bindings. If binding b_i is an answer substitution for $i = 1 \ldots m$ then Figure 5 shows the mapping of a complete and an incomplete set of b_i. Where the set is incomplete we obtain the set up to element k, $0 \leq k < m$, then no more values.

$$
\begin{array}{lcl}
\gamma(\{b_1 \ldots b_m\}) & = & \{c_1 \ldots c_m\} \\
\gamma(\{b_1 \ldots b_k\}_{\cup\perp}) & = & \{c_1 \ldots c_k\}_{\cup\perp}
\end{array}
$$

where

$$
\begin{array}{l}
b_i = [\ Bind_1(X_1, a_1^i), \ldots, Bind_n(X_n, a_n^i)\ ], \\
c_i = \{X_1 \mapsto \gamma(a_1^i), \ldots, X_n \mapsto \gamma(a_n^i)\},
\end{array}
$$

for $i = 1 \ldots m$

Figure 5: γ: Answer Substitutions

8 Correctness: Proof Arguments

The structural induction process is intended to show that the answer set output from the LP for a given query *abstracts* or underestimates the answer set expected from the Z interpretation. We need to determine how a given piece of Z syntax will be interpreted in the LP and Z domains in a given environment. The basis for the induction is the integers and given sets of the specification.

8.1 Structural Induction: Strategy

The base types for the induction are (i) integers, (ii) sets of integers, (iii) given sets and their instantiated elements and (iv) variables, so the first task is to show how their interpretation in the LP underestimates the interpretation in Z. Induction is over each Z construct and includes

1. Numeric expressions
2. Set expressions (union and distributed union)
3. Predicate expressions: infix
4. Set comprehension and variable declarations
5. Predicates: quantified expressions (which depend on declarations)
6. Schemas and Schema Expressions

9 Base Types

(i) Integers
The assumption is that there are largest positive and negative integers available in the system, $MaxInt$, $MinInt$, which cannot be exceeded. Any attempt to do so may cause the computation to terminate. Thus for $m \in \mathbb{Z}$:

$$\mathcal{E}_{LP}[\![\mathrm{m}]\!]\rho_{LP} = m = \mathcal{E}_Z[\![m]\!]\rho_Z, \;\; -MinInt \leq m \leq MaxInt$$
$$\mathcal{E}_{LP}[\![\mathrm{m}]\!]\rho_{LP} = \bot, m < -MinInt \text{ or } m > MaxInt$$

($\bot$ may be implemented by the output of an error message, or alternatively to the character ∞. The latter is suggested by the IEEE floating point standard.) Thus since $\gamma(\bot) = \bot$:

$$\gamma(\mathcal{E}_{LP}[\![m]\!]\rho_{LP}) \sqsubseteq \mathcal{E}_Z[\![m]\!]\rho_Z, m \in \mathbb{Z}$$

(ii) Sets of integers s
Suppose s is a subset of $\{i : \mathbb{N} \mid -MinInt \leq i \leq MaxInt\}$ and assuming that the memory bounds are not exceeded, then the abstract interpretation is *exact*.

Where $MinInt, MaxInt$ are exceeded, (for example $s = \mathbb{Z}$) then s is interpreted as $Null_\perp$ in the LP, and therefore underestimates its interpretation in the Z domain.

$$\gamma(\mathcal{E}_{LP}[\![\{i : -MaxInt \leq i \leq MaxInt\}_{\cup\perp}]\!]\rho_{LP}) = \gamma(Null_\perp) = \varnothing_{\cup\perp} \sqsubseteq \mathcal{E}_Z[\![\mathbb{Z}]\!]\rho_Z,$$

(iii) Given sets and their instantiated elements
Suppose G, g is a given set and typical element. The are interpreted in the LP by base type G, associated constant g and predicate IsG. In each case the abstract interpretation is exact for:

$$\gamma(\mathcal{E}_{LP}[\![g]\!]\rho_{LP}) = g$$
$$\gamma(\mathcal{E}_{LP}[\![G]\!]\rho_{LP}) = \gamma(\{x : IsG(x)\}) = G$$

(iv) Variables
The value of a variable can be obtained as a 'lookup' in the environment, where Env interprets the LP environment as in Andrews [1]. Assuming that the variable has a defined value, the trivial interpretation in the LP is:

$$(x_i = a_i) \longleftarrow Env_{LP}$$

which can be denoted:

$$\mathcal{E}_{LP}[\![x_i]\!]\rho_{LP} = a_i \Leftrightarrow$$
$$(x_i = a_i)\ \&\ true \quad (a_i \neq \perp).$$

If the variable is undefined because of finite failure, the answer returned is false:

$$\mathcal{E}_{LP}[\![x_i]\!]\rho_{LP} = \perp \Leftrightarrow (a_i = \perp)\ \&\ false.$$

If the variable is associated with a schema binding, there are *no* values which satisfy the instantiated variables and the schema predicate, so no answer substitutions. For further discussion see Section 14.

If the variable is undefined because of non-termination or floundering, the answer returned is $\perp$:

$$\mathcal{E}_{LP}[\![x_i]\!]\rho_{LP} = \perp \Leftrightarrow (a_i = \perp)\ \&\ \perp.$$

In either case this is an exact approximation of the interpretation in D_Z, since $\rho_Z = \gamma \circ \rho_{LP}$.

10 Numerical and Set Expressions

Provided $MaxInt, MinInt$ are not exceeded (as in Section 9) the evaluation is via the Peano rules of arithmetic, as in the concrete domain, otherwise the expression evaluates to $\perp$. Thus if fx is a numerical expression, evaluating to m:

$$\mathcal{E}_{LP}[\![fx]\!]\rho_{LP} = m = \mathcal{E}_Z[\![fx]\!]\rho_Z, -MinInt \leq m \leq MaxInt$$
$$\mathcal{E}_{LP}[\![fx]\!]\rho_{LP} = \perp;\ \mathcal{E}_Z[\![fx]\!]\rho_Z = m, m > MaxInt \text{ or } m < -MinInt$$

Thus the abstract evaluation underestimates the concrete and:

$$\gamma(\mathcal{E}_{LP}[\![fx]\!]\rho_{LP}) \sqsubseteq \mathcal{E}_Z[\![fx]\!]\rho_Z,$$

We next apply the rules to set expressions, beginning with set union[4]. Set operators such as intersection and power set are a special case of set comprehensions and will be treated in Section 12.

10.1 Set Union

Consider the syntactic expression '$x_1 \cup x_2$' which is interpreted via an equivalent 'term' in the LP, denoted '$x_1 \cup_{LP} x_2$'. (Recall that in Gödel, 'union' is provided by a function '+'.)

Suppose sets are complete and with complete elements:

$$x_1 \mapsto a_1, x_2 \mapsto a_2 \in \rho_{LP}.$$

The expression $x_1 \cup_{LP} x_2$ is evaluated using the LP ground substitution $\{x_1/a_1, x_2/a_2\}$ so that $(x_1 \cup_{LP} x_2)\{x_1/a_1, x_2/a_2\}$ evaluates to $(a_1 \cup_{LP} a_2)$. We assume that $\cup_{LP}$ is set-theoretic and implements $\cup$ for finite sets in the same manner as $\cup$ for ZF. (See Appendices B, C of my thesis [18].)

Condition 1 becomes:

$$f_{LP}(\mathcal{E}_{LP}[\![(x_1, x_2)]\!]\rho_{LP}) = a_1 \cup_{LP} a_2 = \mathcal{E}_{LP}[\![x_1 \cup x_2]\!]\rho_{LP}.$$

which will hold for set operations for terminating computations. That is the interpretation of ZF operations on sets is built recursively.

Condition 2

If x_1, x_2 are complete sets, $\gamma(x_1, x_2)$ in D_Z evaluates in the expected way to $(\gamma(a_1), \gamma(a_2))$ and

$$f_Z(\mathcal{E}_Z[\![(x_1, x_2)]\!]\rho_Z) = \gamma(a_1) \cup_Z \gamma(a_2) = \mathcal{E}_Z[\![x_1 \cup x_2]\!]\rho_Z.$$

Since $\cup_{LP}$ is set-theoretic then $\gamma(a_1 \cup_{LP} a_2) = \gamma(a_1) \cup_Z \gamma(a_2)$ and **Condition 3** becomes:

$$\begin{aligned}&\gamma\ (f_{LP}\ (\mathcal{E}_{LP}[\![(x_1, x_2)]\!]\rho_{LP}))\\ &\quad = \gamma(a_1 \cup_{LP} a_2) = \gamma(a_1) \cup_Z \gamma(a_2) = f_Z(\gamma(a_1, (a_2))) =\\ &\quad f_Z(\gamma(\mathcal{E}_{LP}[\![(x_1, x_2)]\!]\rho_{LP})).\end{aligned}$$

In other words the computation is exact for terminating computations. There are two ways of extending the result to non terminating computations.

1. Provide an extension of union to incomplete sets and use **AR2** as the proof rule. If

$$x_1 = a_{\cup\perp}, x_2 = b,$$

[4]The reason for following this route rather than commencing from $\bigcup$ and specialising is that $\cup$ is easier to demonstrate.

we define the extension for union:

$$(a_{\cup\perp} \cup_Z b) = (a \cup_Z b_{\cup\perp}) = (a \cup_Z b)_{\cup\perp}$$

which is pointwise monotonic in the Z domain for non-standard sets. **Conditions 1–3** thus hold when 'f' is '$\cup$', thus: since **AR2** holds for x_1, x_2, then **AR2** holds for $\epsilon = x_1 \cup x_2$, where $\cup$ is pointwise monotonic for both standard and non-standard sets.

2. Interpret figure 1 directly and we see that in D_{LP}, if x_1 is not a standard finite set it can only have the value $x_1 = Null_\perp$ and the left hand side of **AR1** for $\epsilon = x_1 \cup x_2$ is

$$\gamma\ ((Null_\perp) \cup_{LP} b) = \gamma\ (Null_\perp) = \varnothing_{\cup\perp},$$

since all set terms in the LP involving $Null_\perp$ evaluate to $Null_\perp$. Then since

$$x_1 \mapsto \varnothing_{\cup\perp}, x_2 \mapsto \gamma(b)$$

are both members of environment $\rho_Z (= \gamma \circ \rho_{LP})$, the right hand side of the ordering relationship becomes:

$$\varnothing_{\cup\perp} \cup_Z \gamma(b)$$

which will, in any case always exceed $\varnothing_{\cup\perp}$, whatever its value, provided that it is still type correct. Since we have established conditions (1 – 3) for complete sets and **AR1** *directly* for incomplete or infinite sets, then **AR1** holds when 'f' is '$\cup$'.

10.2 Distributed Union

We denote distributed union in the LP by $\bigcup_{LP}$ and it is defined so that it implements $\bigcup$ for finite sets in the same manner as $\bigcup$ for ZF. The argument that the LP interpretation underestimates the Z interpretation follows in a similar fashion to the argument for $\cup$.

Consider, first, the evaluation of $\bigcup x$ where $x = \{a_1 \ldots a_n\}$ is a complete, finite set in the LP environment. Then x is $\gamma(\{a_1 \ldots a_n\}) = \{\gamma(a_1) \ldots \gamma(a_n)\}$ in the Z environment.

$$f_{LP}(\mathcal{E}_{LP}[\![(x)]\!]\rho_{LP}) = \bigcup_{LP}(\{a_1 \ldots a_n\}) = \mathcal{E}_{LP}[\![\bigcup x]\!]\rho_{LP} = \mathcal{E}_{LP}[\![fx]\!]\rho_{LP}.$$

Thus **Condition 1** will hold for set operations for terminating computations.
Condition 2
If x is complete and involves only complete sets, the interpretation in D_Z evaluates in the expected way, $\gamma(x) = \{\gamma(a_1) \ldots \gamma(a_n)\}$ and we have

$$f_Z(\mathcal{E}_Z[\![(x)]\!]\rho_Z) = \bigcup_Z \{\gamma(a_1) \ldots \gamma(a_n)\} = \bigcup_Z \gamma(\{a_1 \ldots a_n\}) = \mathcal{E}_Z[\![\bigcup x]\!]\rho_Z.$$

Since $\bigcup_{LP}$ is set-theoretic then: $\gamma(\bigcup_{LP}(\{a_1 \ldots a_n\}) = \bigcup_{Z}(\{\gamma(a_1) \ldots \gamma(a_n)\})$ and **Condition 3** becomes:

$$\gamma\ (f_{LP}\ (\mathcal{E}_{LP}[\![(x)]\!]\rho_{LP}))$$
$$= \gamma(\bigcup_{LP}\{a_1 \ldots a_n\}) = \bigcup_{Z}\{\gamma(a_1) \ldots \gamma(a_n)\} = f_Z(\gamma(\mathcal{E}_{LP}[\![(x)]\!]\rho_{LP})).$$

Since it is equivalent to the set-theoretic definition then $\bigcup_{LP}$ approximates exactly in its interpretation of $\bigcup x$ in the case where x is complete. Thus **AR1** is true for complete sets.

In [3], the operation $\bigcup_{Z}$ is extended to incomplete sets or sets containing incomplete sets so

$$\bigcup_{Z}\{u_{\cup\perp}, v, w\} = \bigcup_{Z}\{u, v, w\}_{\cup\perp},$$
$$\bigcup_{Z}(t_{\cup\perp}) = (\bigcup_{Z} t)_{\cup\perp}$$

and is thus monotonic.

The case for *incomplete sets* follows for $\bigcup_{LP} x = Null_{\perp}$ for x non-standard for the LP and we use the direct method rather than **AR1**. Evaluating left and right hand sides $\bigcup_{LP} x$, where x is incomplete, contains incomplete elements or is infinite. Thus if $(x \mapsto Null_{\perp}) \in \rho_{LP}$, then

$$\text{LHS } = \gamma(\mathcal{E}_{LP}[\![\bigcup x]\!]\rho_{LP})$$
$$= \gamma(\bigcup_{LP} x) = \gamma(Null_{\perp}) = \varnothing_{\cup\perp}$$
$$\text{RHS } = \mathcal{E}_{Z}[\![\bigcup x]\!](\gamma \circ \rho_{LP})$$
$$= \bigcup_{Z} \varnothing_{\cup\perp}$$

and the LP interpretation underestimates whatever the value of the right hand side of the order relationship **AR1**.

Thus $\bigcup$ is interpreted exactly for complete sets and underestimates where sets involved are infinite or non-standard.

11 Predicate Expressions

The evaluator $\mathcal{P}_{LP}$ interprets syntactic predicates p in the LP domain in the manner expected. However the usual boolean set is augmented by an additional 'undefined' element, the output when a program flounders or fails to terminate during its evaluation. This domain element is distinguished from other 'undefined' elements from other types. Thus if

$$Bool_Z = \{tt, ff, \perp_Z^P\}$$
$$Bool_{LP} = \{true, false, \perp_{LP}^P\}$$

then

$$\gamma(true) = tt, \gamma(false) = ff, \gamma(\perp_{LP}^P) = \perp_Z^P.$$

We also have:

$$\mathcal{P}_{LP}[\![P_1 \wedge P_2]\!]\rho_{LP} = (\mathcal{P}_{LP}[\![P_1]\!]\rho_{LP} \ \& \ \mathcal{P}_{LP}[\![P_2]\!]\rho_{LP} = \mathit{true}) \Leftrightarrow$$
$$(\ (\mathcal{P}_{LP}[\![P_1]\!]\rho_{LP} = \mathit{true}) \ \& \ (\mathcal{P}_{LP}[\![P_2]\!]\rho_{LP} = \mathit{true})\ )$$

The approximation requirement, **AR1** for predicates becomes:

$$\gamma(\mathcal{P}_{LP}[\![\epsilon]\!]\rho_{LP}) \sqsubseteq \mathcal{P}_{Z}[\![\epsilon]\!](\gamma \circ \rho_{LP}).$$

Examples of predicates to be interpreted are infix predicates $=$, $\subseteq$ and $\in$. Quantification predicates $\forall\, D \mid p \bullet q$ and $\exists\, D \mid p \bullet q$, where D is a declaration will be treated later, after we have covered declarations.

In an LP, infix predicates $p \in \Sigma_2$ of the form $p(x_1, x_2),\ x_1, x_2 \in \Sigma_1$ are interpreted in such a way that they potentially provide *enhancements* to the existing environment as well as evaluating to boolean values. There are three constraint properties associated with predicate evaluation. Suppose $\mathcal{I}$ is an infix predicate, standing for equality, subset or membership. Then if either (or both) x_1 or x_2 is undefined or only partially defined they can become ground through resolution. We call this property:
Constraint Property 1:

$$\mathcal{P}_{LP}[\![x_1 \mathcal{I} x_2]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x_1 \mathcal{I} x_2]\!]\rho'_{LP} = \mathit{true}$$

where $\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1, x_2 \mapsto a_2\}$.
The environments of predicates conjoined to the infix predicates are also enhanced:
Constraint Property 2:

$$\mathcal{P}_{LP}[\![P \wedge (x_1 \mathcal{I} x_2)]\!]\rho_{LP} = \mathcal{P}_{LP}[\![(x_1 \mathcal{I} x_2) \wedge P]\!]\rho_{LP} = \mathcal{P}_{LP}[\![P]\!]\rho'_{LP}$$

where $\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1, x_2 \mapsto a_2\}$. The same constraint properties can be extended to Z: **Constraint Property 1**:

$$\mathcal{P}_{Z}[\![x_1 \mathcal{I} x_2]\!]\rho_{Z} = \mathcal{P}_{Z}[\![x_1 \mathcal{I} x_2]\!]\rho'_{Z} = \mathit{true}$$

where $\rho'_{Z} = \rho_{Z} \oplus \{x_1 \mapsto \gamma(a_1), x_2 \mapsto \gamma(a_2)\}$.
The environments of predicates conjoined to the infix predicates are also enhanced:
Constraint Property 2:

$$\mathcal{P}_{Z}[\![P \wedge (x_1 \mathcal{I} x_2)]\!]\rho_{Z} = \mathcal{P}_{Z}[\![(x_1 \mathcal{I} x_2) \wedge P]\!]\rho_{Z} = \mathcal{P}_{Z}[\![P]\!]\rho'_{Z}$$

where $\rho'_{Z} = \rho_{Z} \oplus \{x_1 \mapsto \gamma(a_1), x_2 \mapsto \gamma(a_2)\}$.

An extension of these properties is the case where x_1 can take many values. We call this:
Constraint Property 3

$$\mathcal{P}_{LP}[\![x_1 \mathcal{I} x_2]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x_1 \mathcal{I} x_2]\!]\rho'_{LP} = \mathit{true}$$
$$\mathcal{P}_{LP}[\![P \wedge (x_1 \mathcal{I} x_2)]\!]\rho_{LP} = \mathcal{P}_{LP}[\![(x_1 \mathcal{I} x_2) \wedge P]\!]\rho_{LP} = \mathcal{P}_{LP}[\![P]\!]\rho'_{LP}$$

where

$$\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1\} \vee \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_2\} \vee \ldots$$
$$\vee \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_n\}$$

where $\rho'_{LP} \in Env_{LP}$. **Constraint Property 3** can similarly be applied to the Z interpretation. The first infix predicate we shall examine is equality, which considers the equality or otherwise of expressions x_1, x_2.

11.1 Infix Predicate: Equality

The interpretation of $x_1 = x_2$ in the LP not only evaluates to true or false but also potentially provides values for the environment. We need to examine three cases, where *both* (x_1, x_2) are defined, where *only one* is defined and where *neither* are defined.

(i) Both (x_1, x_2) **defined**:

We assume that both (x_1, x_2) are defined, or that there is sufficient information in ρ_{LP} for their evaluation. The truth or falsity of $x_1 = x_2$ in both the LP and in Z is determined by the value of both expressions:

$$\mathcal{P}_{LP}[\![x_1 = x_2]\!]\rho_{LP} \Leftrightarrow (\mathcal{E}_{LP}[\![x_1]\!]\rho_{LP} = \mathcal{E}_{LP}[\![x_2]\!]\rho_{LP})$$
$$\mathcal{P}_{Z}[\![x_1 = x_2]\!]\rho_{Z} \Leftrightarrow (\mathcal{E}_{Z}[\![x_1]\!]\rho_{Z} = \mathcal{E}_{Z}[\![x_2]\!]\rho_{Z})$$

When both variables are defined, it is appropriate to consider **AR1**.

Condition 1 will hold for set operations for terminating computations, for if 'f' is equality, then

$$f_{LP}(\mathcal{E}_{LP}[\![x_1]\!]\rho_{LP}, \mathcal{E}_{LP}[\![x_2]\!]\rho_{LP})$$

is the boolean value of

$$(\mathcal{E}_{LP}[\![x_1]\!]\rho_{LP} = \mathcal{E}_{LP}[\![x_2]\!]\rho_{LP})$$

and so

$$f_{LP}(\mathcal{P}_{LP}[\![(x_1, x_2)]\!]\rho_{LP}) = f_{LP}(\mathcal{E}_{LP}[\![x_1]\!]\rho_{LP}, \mathcal{E}_{LP}[\![x_2]\!]\rho_{LP})$$
$$= \mathcal{P}_{LP}[\![x_1 = x_2]\!]\rho_{LP}.$$

Condition 2 follows in a similar manner because if 'f' is equality, then

$$f_Z(\mathcal{E}_Z[\![x_1]\!]\rho_Z, \mathcal{E}_Z[\![x_2]\!]\rho_Z) \Leftrightarrow (\mathcal{E}_Z[\![x_1]\!]\rho_Z = \mathcal{E}_Z[\![x_2]\!]\rho_Z)$$

and so

$$f_Z(\mathcal{P}_Z[\![(x_1, x_2)]\!]\rho_Z) = f_Z(\mathcal{E}_Z[\![x_1]\!]\rho_Z, \mathcal{E}_Z[\![x_2]\!]\rho_Z)$$
$$= \mathcal{P}_Z[\![x_1 = x_2]\!]\rho_Z.$$

Condition 3:

We require for f syntactic infix '=':

$$\gamma(f_{LP}(\mathcal{P}_{LP}[\![(x_1, x_2)]\!]\rho_{LP})) = f_Z(\gamma(\mathcal{P}_{LP}[\![(x_1, x_2)]\!]\rho_{LP})).$$

Suppose that $(a_1 = a_2)$ evaluates to *true* and since from finite theory of sets, equality is set-theoretic for finite sets, then $(\gamma(a_1) = \gamma(a_2))$ evaluates to tt and we have

$$\gamma(a_1 = a_2) = \gamma(true)$$
$$= tt = \gamma(a_1) = \gamma(a_2) = f_Z(\gamma(a_1), \gamma(a_2)).$$

The same holds if $(a_1 = a_2)$ evaluates to *false*. Thus when x_1, x_2 are both defined and termination is successful, the predicate evaluation in the LP is an exact approximation to evaluation in the Z domain.

(ii) One of (x_1, x_2) defined:
Supposing that x_1 is undefined, $x_1 \mapsto \perp \in \rho_{LP}$ and $x_2 \mapsto a$ then the equality predicate results in the evaluation of x_1 via unification (and we obtain a similar result for x_1 defined, x_2 undefined).

If either of x_1, x_2 is undefined, it is more appropriate to consider **AR1** directly. Thus assuming that the execution terminates, and **Constraint Property 1** applies, then the approximation is again exact, for the left and right hand sides of **AR1** are equal:

$$\textbf{LHS of AR1} = \gamma((\mathcal{P}_{LP}[\![(x_1 = x_2)]\!]\rho_{LP}))$$
$$= \gamma(true) = tt$$
$$\textbf{RHS of AR1} = (\mathcal{P}_Z[\![(x_1 = x_2)]\!]\rho_Z)) = tt$$

It is possible that the computation fails to terminate and the LHS to evaluate to $\perp$. In that case the execution underestimates the Z interpretation.

(iii) Both (x_1, x_2) undefined:
If both x_1, x_2 are undefined or incomplete, then it is still possible for the environment to be enhanced since both of x_1, x_2 may become ground through unification. This is another example of **Constraint Property 1** : where x_1, x_2 both unify to the same ground term a and $\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a, x_2 \mapsto a\}$.

AR1 can be evaluated for the case where x_1, x_2 are undefined, and there are three possibilities:

Program terminates This occurs when both x_1, x_2 become ground through unification, as explained above. **Conditions 1–3** hold in the same manner as when x_2 is defined and the approximation is exact.

Execution does not terminate, but variables become ground in Z This occurs when (for example) constraint properties on sets are such that x_1, x_2 in Z, but non-ground in the LP. The approximation underestimates for

$$\textbf{LHS of AR1} = \gamma((\mathcal{P}_{LP}[\![(x_1 = x_2)]\!]\rho_{LP}))$$
$$= \gamma(\perp) = \perp$$
$$\textbf{RHS of AR1} = (\mathcal{P}_Z[\![(x_1 = x_2)]\!]\rho_Z)) = tt$$

Execution does not terminate and neither variable becomes ground in Z In that case the approximation is exact for

$$\textbf{LHS of AR1} = \gamma(\perp) = \perp$$
$$\textbf{RHS of AR1} = (\mathcal{P}_Z[\![(x_1 = x_2)]\!]\rho_Z)) = \perp$$

We can summarise, thus. Three cases have been examined, depending on whether or not x_1, x_2 are defined prior to execution of equality function and in each case **AR1** is true where 'f' is the syntactic predicate $=$ for variable (x_1, x_2): **AR1** holds for $(x_1 = x_2)$.

11.2 Infix Predicate: Subset

The proof of correctness for predicate $\subseteq x$ where $x = (x_1, x_2)$ follows a similar structure to that for $=$. However the subset predicate potentially provides *more than one possible value* for the environment. In this case we must have x_2 defined. However, before the computation commences it may be the case that $x_1 \mapsto \perp \in \rho_{LP}$. After the computation, x_1 takes values from set x_2. The proof follows that for equality, with the difference that x_1 can take many values. The different values contribute to different answer substitutions. It can be shown that the predicate evaluation in the LP underestimates the evaluation in the Z domain. Again there are three cases, where we shall assume that the environment does not include incomplete sets and that sets are finite

(i) Both (x_1, x_2) defined:

Condition 1-2 will hold for set operations for terminating computations: the interpretation of ZF predicates on sets is built recursively in both the LP and Z.

$$\begin{aligned} f_{LP}(\mathcal{P}_{LP}[\![(x_1, x_2)]\!]\rho_{LP}) &= f_{LP}(\mathcal{E}_{LP}[\![x_1]\!]\rho_{LP}, \mathcal{E}_{LP}[\![x_2]\!]\rho_{LP}) \\ &= (\mathcal{E}_{LP}[\![x_1]\!]\rho_{LP} \subseteq_{LP} \mathcal{E}_{LP}[\![x_2]\!]\rho_{LP}) = \mathcal{P}_{LP}[\![x_1 \subseteq x_2]\!]\rho_{LP}. \\ f_Z(\mathcal{P}_Z[\![(x_1, x_2)]\!]\rho_Z) &= f_Z(\mathcal{E}_Z[\![x_1]\!]\rho_Z, \mathcal{E}_Z[\![x_2]\!]\rho_Z) \\ &= (\mathcal{E}_Z[\![x_1]\!]\rho_Z \subseteq_Z \mathcal{E}_Z[\![x_2]\!]\rho_Z) = \mathcal{P}_Z[\![x_1 \subseteq x_2]\!]\rho_Z. \end{aligned}$$

Condition 3:

We require for f syntactic infix '$\subseteq$':

$$\gamma(f_{LP}(\mathcal{E}_{LP}[\![(x_1, x_2)]\!]\rho_{LP})) \sqsubseteq f_Z(\gamma(\mathcal{E}_{LP}[\![(x_1, x_2)]\!]\rho_{LP})).$$

Suppose that $(a_1 \subseteq_{LP} a_2)$ evaluates to *true* and since 'subset' for finite sets in the LP is set-theoretic, then $(\gamma(a_1) \subseteq_Z \gamma(a_2))$ evaluates to tt and we have

$$\begin{aligned} \gamma(a_1 \subseteq_{LP} a_2) &= \gamma(true) \\ &= tt = \gamma(a_1) \subseteq_Z \gamma(a_2) = f_Z(\gamma(a_1), \gamma(a_2)). \end{aligned}$$

The same holds if $(a_1 \subseteq_{LP} a_2)$ evaluates to *false*. Thus when termination is successful, the predicate evaluation in the LP is an exact approximation to evaluation in the Z domain.

(ii) Variable x_1 undefined

If variable x_1 is undefined, then we have an example of **Constraint Property 3** for suppose $\{a_1 \ldots a_n\}$ are subsets of x_2 in the LP then

$$\begin{aligned} &\mathcal{P}_{LP}[\![x_1 \subseteq x_2]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x_1 \subseteq x_2]\!]\rho'_{LP} = true \\ &\mathcal{P}_{LP}[\![P \wedge (x_1 \subseteq x_2)]\!]\rho_{LP} = \mathcal{P}_{LP}[\![(x_1 \subseteq x_2) \wedge P]\!]\rho_{LP} = \mathcal{P}_{LP}[\![P]\!]\rho'_{LP} \end{aligned}$$

where

$$\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1\} \vee \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_2\} \vee \ldots$$
$$\vee \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_n\}.$$

In a similar fashion suppose $\{\gamma(a_1) \ldots \gamma(a_n)\}$ are subsets of x_2 in Z then

$$\mathcal{P}_Z[\![x_1 \subseteq x_2]\!]\rho_Z = \mathcal{P}_Z[\![x_1 \subseteq x_2]\!]\rho'_Z = \mathit{true}$$
$$\mathcal{P}_Z[\![P \wedge (x_1 \subseteq x_2)]\!]\rho_Z = \mathcal{P}_Z[\![(x_1 \subseteq x_2) \wedge P]\!]\rho_Z = \mathcal{P}_Z[\![P]\!]\rho'_Z$$

where

$$\rho'_Z = \rho_Z \oplus \{x_1 \mapsto \gamma(a_1)\} \vee \rho'_Z = \rho_Z \oplus \{x_1 \mapsto \gamma(a_2)\} \vee \ldots$$
$$\vee \rho'_Z = \rho_Z \oplus \{x_1 \mapsto \gamma(a_n)\}.$$

Where there is only one way the environment can be enhanced, then we can consider **AR1**. However where there is more than one way of enhancing the environment, the comparison between the Z and LP domains will be deferred to Section 12 for in that case the values contribute to a set expression.

Thus assuming that the execution terminates, and x_1, x_2 take unique values then the left and right hand sides of **AR1** are as follows

$$\textbf{LHS of AR1} = \gamma((\mathcal{P}_{LP}[\![(x_1 \subseteq x_2)]\!]\rho'_{LP}))$$
$$= \gamma(\mathit{true}) = tt$$
$$\textbf{RHS of AR1} = (\mathcal{P}_Z[\![(x_1 \subseteq x_2)]\!]\rho'_Z)) = tt$$

(iii) Both Variables x_1, x_2 undefined
If both x_1, x_2 are undefined, then the computation in an LP such as Gödel will fail to terminate and the output is $\perp^P_{LP}$.

$$\textbf{LHS of AR1} = \gamma((\mathcal{P}_{LP}[\![(x_1 \subseteq x_2)]\!]\rho_{LP}))$$
$$= \gamma(\perp^P_{LP}) = \perp^P_Z$$

which will underestimate the RHS. Thus the interpretation in the LP underestimates the interpretation in Z for all three cases above.

If either of x_1, x_2 are incomplete sets, then $x_1 \subseteq x_2$ evaluates to '$\perp^P_{LP}$' in the LP. which underestimates the interpretation in Z. Thus **AR1** is true where 'f' is the syntactic predicate $\subseteq$ for variable (x_1, x_2), and the LP interpretation of $\subseteq$ underestimates the Z interpretation.

11.3 Infix Predicate: Membership

The last infix predicate is membership, $x_1 \in x_2$ and the proof follows that for $\subseteq$. If $x_1 \in x_2$ then x_1 has potentially many values for x_2 defined and not empty. The three cases can be summarised:
(i) Both x_1, x_2 are defined

$$\mathcal{P}_{LP}[\![x_1 \in x_2]\!]\rho_{LP} = \mathit{true}$$

where $x_1 \in x_2$. Otherwise the value if *false*. The predicate is interpreted as tt, ff respectively when evaluated in D_Z for $x_1 \in x_2, x_1 \notin x_2$.

If the computation terminates then the approximation is exact, as for $=$ and $\subseteq$. The result follows similarly for Z, where as before we extend the interpretation of predicates in D_Z to cover constraint satisfaction. Thus for x_2 defined and not empty:

$$\mathcal{P}_Z[\![x_1 \in x_2]\!]\rho_Z = \mathcal{P}_Z[\![x_1 \in x_2]\!]\rho'_Z \ \& \ true$$

where $\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a\}, a \in x_2$. If x_2 is defined and empty, then the predicate evaluates to ff in Z:

$$\mathcal{P}_Z[\![x_1 \in \varnothing]\!]\rho_Z = \mathcal{P}_Z[\![x_1 \in x_2]\!]\rho_Z \ \& \ ff$$

Thus the approximation for $\in$ is exact for both x_1, x_2 defined

(ii) for x_1 undefined and x_2 defined.

Constraint Properties 2-3 apply also for 'membership': Suppose $x_2 = \{a_1 \ldots a_n\}$, then

$$\mathcal{P}_{LP}[\![x_1 \in x_2]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x_1 \in x_2]\!]\rho'_{LP}$$
$$\mathcal{P}_{LP}[\![P \wedge (x_1 \in x_2)]\!]\rho_{LP} = \mathcal{P}_{LP}[\![(x_1 \in x_2) \wedge P]\!]\rho_{LP} = \mathcal{P}_{LP}[\![P]\!]\rho'_{LP}$$

where

$$\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1\} \vee \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_2\}, \ldots \vee \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_n\}.$$

Thus assuming that the execution terminates, the situation is the same as for subset in that the choice of the binding for x_1 is non-deterministic for sets with more than one member. Where the choice is deterministic then

$$\textbf{LHS of AR1} = \ \gamma((\mathcal{P}_{LP}[\![(x_1 \in x_2)]\!]\rho_{LP}))$$
$$= \gamma(true) = tt$$
$$\textbf{RHS of AR1} = \ (\mathcal{P}_Z[\![(x_1 \in x_2)]\!]\rho_Z)) = tt$$

(iii) Both x_1, x_2 undefined

For x_2 undefined the LP interpretation results in $\perp^P_{LP}$ and underestimates the Z interpretation, however it evaluates.

We have examined all possibilities for values of x_1, x_2 and in all cases **AR1** is true where 'f' is the syntactic predicate $\in$ for variable (x_1, x_2); the LP interpretation of $\in$ underestimates the Z interpretation as required.

12 Set Comprehension and Variable Declarations

Set Comprehension is defined in terms of declarations $D_1; \ldots; D_n$, a constraining predicate p and an expression t involving the declared variables:

$$x_1 : \tau_1; \ x_2 : \tau_2; \ \ldots x_n : \tau_n \mid p \bullet t$$

We first present the interpretation of declarations, then within the context of a set declaration.

12.1 Variable Declarations

Variable declarations occur within bound expressions with structure: $\_ \; D \mid p \bullet t \; \_$ where D is a declaration,p is a predicate and t a term. D is of the form:

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n\}$$

These include *set comprehensions, quantified expressions, lambda expressions* and *schemas.* The declaration results in a single tuple of values $(x_1, \ldots x_n)$ being generated (or tested in the case of schemas). Each value is constrained by p and used to evaluate t.

An evaluation function $\mathcal{D}_{LP}$ gives the interpretation in D_{LP} of syntactic declarations $x : \tau$, where x is a variable and τ is set-valued with value provided by ρ_{LP}. The declarations considered in this section do not include schema references, for these are treated separately.

1. τ is a *set:*

   $$\mathcal{D}_{LP}[\![x : \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho_{LP}$$

   '$x : \tau$' has the effect of either testing a value or updating the environment as in the case of the membership predicate.

2. τ is a *Power Set*, $\tau = \mathbb{P}\,\tau'$ say:

   $$\mathcal{D}_{LP}[\![x : \mathbb{P}\,\tau']\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \subseteq \tau']\!]\rho_{LP}$$

   '$\tau : \mathbb{P}\,\tau'$ ' uses a 'subset' test rather than a 'membership of power set' test for reasons of efficiency. It has the same effect on the environment as the subset predicate.

3. τ is a *Cartesian Product*, $\tau_1 \times \tau_2$:

   $$\mathcal{D}_{LP}[\![x : \tau_1 \times \tau_2]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x = (x_1 \mapsto x_2)]\!]\rho_{LP}$$
   $$\quad \& \; \mathcal{P}_{LP}[\![x_1 \in \tau_1]\!]\rho_{LP}) \; \& \; \mathcal{P}_{LP}[\![x_2 \in \tau_2]\!]\rho_{LP})$$

   'T2' captures a representation of ordered pair (as an example of a tuple) in the LP. In our Gödel library this is 'OrdPair'. The following shows the implementation of $\twoheadrightarrow$, which illustrates the interpretation of cartesian product.

```
PF(pf, s1, s2) <- ALL [z,x,y] (z In pf &
                                  (z = OrdPair(x,y))
                    ->  (x In s1) & (y In s2)  &
                    ALL [u] (OrdPair(x, u) In pf -> u = y)).
```

Thus a single declaration (such as $x : \tau$) has the effect of enhancing the environment as for the membership predicate:

$$\mathcal{D}_{LP}[\![x : \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho'_{LP}$$

where if $\tau = \{a_1 \ldots a_n\}$ then

$$\rho'_{LP} = \rho_{LP} \oplus \{x \mapsto a_1\} \vee \rho'_{LP} = \rho_{LP} \oplus \{x \mapsto a_2\}, \ldots \vee \rho'_{LP} = \rho_{LP} \oplus \{x \mapsto a_n\}.$$

In general, if $x : \tau$ is a declaration, then

$$\mathcal{D}_{LP}[\![x : \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho'_{LP}$$

where it is possible for ρ'_{LP} to take many values determined by the nature of the type τ.

A sequence of declarations is evaluated in the LP as a conjunction

$$\mathcal{D}_{LP}[\![D_1;\ \ldots;\ D_n]\!]\rho_{LP} = \mathcal{D}_{LP}[\![D_1]\!]\rho_{LP}\ \&\ \ldots\ \&\ \mathcal{D}_{LP}[\![D_n]\!]\rho_{LP}.$$

Declarations can be represented in a simpler manner in Z, where again values are chosen from some set-valued τ. However in this case τ is a type constructor thus

$$\mathcal{D}_Z[\![x : \tau]\!]\rho_Z = \mathcal{P}_Z[\![x \in \tau]\!]\rho'_Z$$

where τ is a set, or a power set, $\mathbb{P}\,\tau'$, or a cartesian product $\tau' \times \tau''$ and ρ'_Z takes its values from τ.

12.2 Interpretation of Set Comprehension

A set comprehension is

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}$$

where each $x_i : \tau_i$ provides a value which contributes to the tuple $(x_1, \ldots x_n)$ which is used to evaluate t. Thus if $s = \{d \mid p \bullet t\}$ is a syntactical set comprehension it is interpreted in D_{LP} as $\mathcal{E}_{LP}[\![s]\!]\rho_{LP}$ and in D_Z as $\mathcal{E}_Z[\![s]\!]\rho_Z$. Since declarations in the LP are treated as predicates, then the set comprehension of s is interpreted in the LP

$$\mathcal{E}_{LP}[\![s]\!]\rho'_{LP} = \{\mathcal{D}_{LP}[\![d]\!]\rho'_{LP}\ \ \&\ \ \mathcal{P}_{LP}[\![p]\!]\rho'_{LP} \bullet \mathcal{E}_{LP}[\![t]\!]\rho'_{LP}\}$$

The environment ρ'_{LP} inside the comprehension is the variable which acts as a set generator, for recall that $\mathcal{D}_{LP}[\![x : \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho'_{LP}$. A similar interpretation is true for D_Z.

We assert that for terminating computations, **AR1** is true, since the interpretation of set comprehension is exact. We initiate an induction process over the set generators (as in [3]).

A set with *no* set generators, is defined in the LP domain:

$$\mathcal{E}_{LP}[\![\{|\ p \bullet t\}]\!]\rho'_{LP} = \{\mathcal{E}_{LP}[\![t]\!]\rho'_{LP}\} \ \text{ where } \mathcal{P}_{LP}[\![p]\!]\rho'_{LP} = \mathit{true}$$
$$\mathcal{E}_{LP}[\![\{|\ p \bullet t\}]\!]\rho'_{LP} = \{\} \ \text{ where } \mathcal{P}_{LP}[\![p]\!]\rho'_{LP} = \mathit{false}$$

In the Z domain

$$\mathcal{E}_Z[\![\{\mid p \bullet t\}]\!]\rho'_Z = \{\mathcal{E}_Z[\![t]\!]\rho'_Z\}, \quad \text{where } \mathcal{P}_Z[\![p]\!]\rho'_Z = tt$$
$$\mathcal{E}_Z[\![\{\mid p \bullet t\}]\!]\rho'_Z = \{\} \quad \text{where } \mathcal{P}_Z[\![p]\!]\rho'_Z = ff$$

Induction is based on the equivalence:

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\} = \bigcup\{x_1 : \tau_1 \bullet \{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}\}$$

We first consider terminating computations where the interpretation is proposed as exact. The induction process depends on showing that if we assume that **AR1** holds for

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}$$

then it holds for

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n;\ x_{n+1} : \tau_{n+1} \mid p \bullet t\}$$

For the induction process we first consider the base case
Base Case: no set generators
We consider **AR1** for the base case where there are no set generators. **Conditions 1–2** are true for standard sets since the interpretation is built recursively in both the LP and Z domains:

$$\mathcal{E}_{LP}[\![\{\mid p \bullet t\}]\!]\rho'_{LP} = \{\mathcal{E}_{LP}[\![t]\!]\rho'_{LP}\} \quad \text{where } P_{LP}[\![p]\!]\rho'_{LP} = \mathit{true}$$
$$\mathcal{E}_{LP}[\![\{\mid p \bullet t\}]\!]\rho'_{LP} = \{\} \quad \text{where } \mathcal{P}_{LP}[\![p]\!]\rho'_{LP} = \mathit{false}$$
$$\mathcal{E}_Z[\![\{\mid p \bullet t\}]\!]\rho'_Z = \{\mathcal{E}_Z[\![t]\!]\rho'_Z\}, \quad \text{where } \mathcal{P}_Z[\![p]\!]\rho'_Z = tt$$
$$\mathcal{E}_Z[\![\{\mid p \bullet t\}]\!]\rho'_Z = \{\} \quad \text{where } \mathcal{P}_Z[\![p]\!]\rho'_Z = ff$$

Assuming that the calculation of p, t for environment ρ'_{LP} terminates, the approximation of '$\{\mid p \bullet t\} = f(p,t)$' in the LP domain is exact, since **Condition 3** becomes:

$$\gamma(f_{LP}(\mathcal{E}_{LP}[\![(p,t)]\!]\rho'_{LP})) = f_Z(\gamma(\mathcal{E}_{LP}[\![(p,t)]\!]\rho'_{LP})).$$

If the calculation of p, t fails to terminate, then the LHS of the approximation evaluates to $\gamma(\mathit{Null}_\perp) = \varnothing_{\cup\perp}$ and thus underestimates the RHS, however it is evaluated.

$$\text{LHS} = \gamma(\mathcal{E}_{LP}[\![f\,(p,t)]\!]\rho_{LP}) = \gamma(\mathit{Null}_\perp) = \varnothing_{\cup\perp}$$
$$\text{RHS} = \mathcal{E}_Z[\![f\,(p,t)]\!](\gamma \circ \rho_{LP})$$

Set Comprehension – Induction on Declaration Sequence
Induction is based on the equivalence:

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\} = \bigcup\{x_1 : \tau_1 \bullet \{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}\}$$

for values of $\tau_1, \ldots \tau_n$ in the environment. Write the interpretation of $\bigcup$ in Z domain and LP domains as $\bigcup_Z$ and $\bigcup_{LP}$ as in Section 10.

The equivalence means that the set comprehension with one generator, '$\{x : \tau \mid p \bullet t\}$', can be evaluated in the LP environment:

$$\mathcal{E}_{LP}[\![\{x : \tau \mid p \bullet t\}]\!]\rho_{LP} = \bigcup_{LP} \mathcal{E}_{LP}[\![\{\mid p \bullet t\}]\!]\rho'_{LP}$$

where τ is $\{a_1 \ldots a_n\}$ in ρ_{LP} and $\rho'_{LP} = \rho_{LP} \oplus \{x \mapsto a_i\}$. The interpretation is similar for D_Z.

For n generators, $x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n$, if $\tau_1 \mapsto s \in \rho_{LP}$, then $\tau_1 \mapsto \gamma(s) \in \rho_Z$ and the set comprehensions in both the Z and LP domains can be represented as the distributed union of a family of sets indexed by i where $a_i \in s, b_i \in \gamma(s)$ respectively:

$$\begin{aligned}
&\mathcal{E}_{LP}[\![\{x_1 : \tau_1;\ \ldots x_n : \tau_n \mid p \bullet t\}]\!]\rho_{LP} = \\
&\quad \bigcup_{LP}\{a_i \in s \bullet \mathcal{E}_{LP}[\![\{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}]\!](\rho_{LP} \oplus \{x_1 \mapsto a_i\})\} \\
&\mathcal{E}_Z[\![\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}]\!]\rho_Z = \\
&\quad \bigcup_Z\{b_i \in \gamma(s) \bullet \mathcal{E}_Z[\![\{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}]\!](\rho_Z \oplus \{x_1 \mapsto b_i\})\}
\end{aligned}$$

For finite sets, the approximation of **Condition 3** is exact. Thus since **AR1** holds for the empty sequence and assuming it holds for the sequence

$$\{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}$$

in environments

$$\rho_Z \oplus \{x_1 \mapsto b_i\},\ \rho_{LP} \oplus \{x_1 \mapsto a_i\}$$

it then holds for

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}$$

in environments ρ_Z, ρ_{LP}. Thus **AR1** holds for $\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}$ since it holds for each of its components τ_i, p, t.

For infinite sets, or if any set is non-standard in the LP, the induction process depends on whether we are addressing set terms or sets of answer substitutions:

- For set terms the LP interpretation of s evaluates to $Null_\perp$ and underestimates the Z interpretation in the same manner as the 'no set generator case'.
- For the 'answer set' the result depends on distributed union, where incomplete sets are involved. This is because we can equivalently express a set comprehension as a distributed union. We see that this underestimates for incomplete sets.

Thus set comprehension in the LP is an exact interpretation for finite or complete sets. For infinite or incomplete sets the LP interpretation is an underestimation. This is true for either set terms or sets of answer substitutions.

12.3 Set Operations Power Set, Set Intersection

Other set operations $\epsilon = f(x_1, x_2, \ldots, x_n)$ can be expressed via set comprehensions. Examples are set intersection and power set:

Set Intersection $s = x_1 \cap x_2$ in the LP is part of the library of set operations. However $\cap$ can be expressed as

```
s = {x : (x In x1) & (x In x2 )}.
```

where we are assuming that x_1, x_2 are appropriately typed. In Z this last condition is expressed explicitly so

$$s = x_1 \cap x_2 = \{x : X \mid (x \in x_1 \land x \in x_2) \bullet x\}$$

This is treated as a set comprehension where $p = x \in x_1 \land x \in x_2$ Thus for terminating computations, the interpretation in the LP approximates exactly, and for non-terminating computations, the LP interpretation underestimates.

Power Set in the LP, $s = \mathbb{P}\, x$ can be be expressed in Gödel as

```
s = {z : z Subset x }.
```

Its generic 'LP form' is as a set comprehension with predicate *true*:

$$s = \mathcal{E}_{LP}[\![\{z \subseteq x \mid true \bullet z\}]\!]\rho_{LP}.$$

Since 'power set' is a type in Z, there is no specific definition for it, however the power set axiom of ZF provides a definition in Z: The power set set $s = \mathbb{P}\, x$ is such that

$$\forall z \bullet (z \in s \Leftrightarrow z \subseteq x)$$

and this set and the interpretation in the LP can be shown to be equal. Thus the interpretation of power set is exact for finite sets. For infinite sets, the LP interpretation is $Null_\perp$ which always underestimates.

The interpretation is exact if the computations terminate. For infinite or incomplete sets, the interpretation in the LP evaluates to $Null_\perp$ and so underestimates the interpretation in Z.

12.4 Quantifiers

Universal Quantification
The syntactic predicate '$\forall x : s \mid p \bullet q$' is interpreted in the LP :

```
ALL [x] (x In s <-> p => q )
```

and in Z $\forall x : s \mid p \Rightarrow q$ and is evaluated for finite sets s on an element by element basis for values of s. Its interpretation can be denoted in the LP as $\mathcal{P}_{LP}[\![fx]\!]\rho_{LP}$, where x is the tuple s, p, q and 'f' the syntactic '$\forall$'. For terminating computations, **Condition 1–2** hold in the LP and in Z. If '$\forall x : s \mid p \Rightarrow q$' is *true* then **Condition 3** becomes

$$\begin{aligned}\textbf{LHS} &= \gamma(f_{LP}(\mathcal{P}_{LP}[\![(s,p,q)]\!]\rho_{LP})) \\ &= \gamma(true) = tt \\ \textbf{RHS} &= f_Z(\gamma(\mathcal{P}_{LP}[\![(s,p,q)]\!]\rho_{LP})) = tt.\end{aligned}$$

and is thus exact for each p, q in an environment containing s. The result follows similarly if '$\forall x : s \mid p \bullet q$' is *false*.

For infinite sets the truth value in the LP will be $\bot^P_{LP}$ i.e. it will fail to terminate and the LHS of Condition 3 will evaluate to $\bot^P_{LP}$. For infinite sets, the Z interpretation of the quantification will result in the value tt or ff, and $\gamma(\bot^P_{LP}) = \bot^P_Z$ and $\bot^P_Z \sqsubseteq ff, \bot^P_Z \sqsubseteq tt$. For cases where s is incomplete, or not fully defined, then the LP interpretation results in $\bot^P_{LP}$, which either underestimates the interpretation in Z, if it is ff, tt, or is exact, if the interpretation in Z is $\bot^P_Z$. Thus in all cases, the interpretation in the LP of universal quantification adheres to **AR1**.

Existential Quantification

A similar interpretation is true for $\exists$ where '$\exists x : s \mid p \bullet q$' is interpreted in the LP

```
SOME [x] (x In s <-> p & q )
```

and in Z: $\exists x : s \mid p \wedge q$ The LP interpretation evaluates to $true, false, \bot^P_{LP}$, which always underestimates its interpretation in Z, as for $\forall$.

13 Function Application and Lambda Expressions

Function application of t_1 to t_2 assumes that t_1 is appropriately typed, as a set of pairs. It is interpreted in the LP by

$$\mathcal{E}_{LP}[\![t_1\, t_2]\!]\rho_{LP} = a \Leftrightarrow t_2 \mapsto a \in t_1$$

It is mapped in a similar way in Z. For terminating computations, where set t_1 is finite, the interpretation is exact. Where t_1 is infinite or incomplete, the LP underestimates the Z interpretation for

$$\mathcal{E}_{LP}[\![t_1\, t_2]\!]\rho_{LP} = \bot.$$

Lambda expressions require evaluation individually:
$\lambda x_1 : \tau_1; \ldots x_n : \tau_n \mid p \bullet t$ where t is a term can be expressed (in Z) as a set of

maplets $x \mapsto a$ where the x is a tuple $(x_1, \ldots, x_n)$ and a is the term t evaluated at $(x_1, \ldots, x_n)$:

$$\{x_1 : \tau_1; \ \ldots x_n : \tau_n \mid p \bullet (x_1, \ldots, x_n) \mapsto t)\}$$

It is interpreted as the equivalent set expression in the LP:

$$\begin{aligned}
&\mathcal{E}_{LP}[\![\lambda\, x_1 : \tau_1; \ \ldots x_n : \tau_n \mid p \bullet t]\!]\rho_{LP} = \\
&\mathcal{E}_{LP}[\![\{x_1 : \tau_1; \ \ldots x_n : \tau_n \mid p \bullet (x_1, \ldots, x_n) \mapsto t)\}]\!]\rho_{LP} = \\
&\{\mathcal{D}_{LP}[\![x_1 : \tau_1; \ \ldots x_n : \tau_n]\!]\rho_{LP} \ \&\ \mathcal{P}[\![p]\!] \mid T2(Tn(x_1, \ldots, x_n) \mapsto t)\}
\end{aligned}$$

An example can be seen in Appendix C of [18]. The approximation is exact for terminating computations and underestimates for the rest.

14 Interpretation of Schemas and Schema Expressions

Suppose that the syntactic objects $schema, axdef \in \Sigma_3$ are interpreted in the LP and in Z by $\mathcal{S}_{LP}$, $\mathcal{S}_Z$ respectively. A schema can be represented (in its horizontal form) by the following syntactic object:

$$Sch \mathrel{\widehat=} [\, D_1; \ \ldots; \ D_n \mid CP \,]$$

where $D_i = X_i : \tau_i$, and $CP \ ::= \ CP_1 \wedge \ldots \wedge CP_m$.

Sch evaluates to a set expression, of bindings of variable name(s) to values. The bindings are constrained by the variable declarations and by the schema predicate. Suppose GCP is defined as CP where all the free occurrences of $X_1 \ldots X_n$ are replaced by $x_1 \ldots x_n$

$$GCP(x_1, \ldots, x_n) = CP(X_1/x_1, \ldots, X_n/x_n)$$

and any bound variables replaced by arbitrary local variables. *A set of schema bindings* of *Sch* can be represented in Z (as suggested in [3]) by a set expression:

$$\{x_1 : \tau_1; \ \ldots; \ x_n : \tau_n \mid GCP \bullet \{X_1 \mapsto x_1, \ldots, X_n \mapsto x_n\}\}$$

There is a similar representation in the LP where $[Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)]$ replaces $\{X_1 \mapsto x_1, \ldots, X_n \mapsto x_n\}$. If we assume that the set of bindings is constrained by an initial imposed environment ρ^o then the interpretation of the schema $Sch \mathrel{\widehat=} [\, D \mid CP \,]$ is the interpretation of a set expression

$$\begin{aligned}
&\mathcal{S}_{LP}[\![[\, X_1 : \tau_1; \ \ldots; \ X_n : \tau_n \mid CP \,]]\!]\rho^o_{LP} \\
&= \mathcal{E}_{LP}[\![\{x_1 : \tau_1; \ \ldots; \ x_n : \tau_n \mid GCP \\
&\qquad \bullet [Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)]\}]\!]\rho^o_{LP}
\end{aligned}$$

The interpretation of schemas and schema expressions is in terms of a *characteristic predicate*, providing a single binding for a schema expression.

14.1 Characteristic Predicate for a Schema Expression

A schema binding is obtained by providing the schema with some initial environment, ρ^0_{LP}. In its initial state a schema is interpreted as:

$$\mathcal{S}_{LP}[\![[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP \,]]\!] \rho^o_{LP}$$

and this evaluates in the LP to bindings of variable names to values. During the execution the environment has been enhanced to ρ This binding is a member of the set defined previously:

$$\begin{aligned}
&\mathcal{S}_{LP}[\![[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP \,]]\!] \rho^o_{LP} \\
&= \mathcal{E}_{LP}[\![\{ x_1 : \tau_1;\ \ldots;\ x_n : \tau_n \mid GCP \\
&\quad \bullet\ [Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)] \}]\!] \rho^o_{LP}
\end{aligned}$$

where each x_i satisfies

$$\mathcal{D}_{LP}[\![D_1;\ \ldots;\ D_n]\!] \rho_{LP}\ \&\ \mathcal{P}_{LP}[\![GCP]\!] \rho_{LP}.$$

where each enhanced environment $\rho_{LP} \in Env_{LP}$. The characteristic schema predicate of Sch is as follows:

$$\begin{aligned}
&SchemaType(binding, Sch) \Leftrightarrow \\
&\quad (binding = [Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)])\ \& \\
&\qquad \mathcal{D}_{LP}[\![D_1;\ \ldots;\ D_n]\!] \rho_{LP}\ \&\ \mathcal{P}_{LP}[\![GCP]\!] \rho_{LP}.
\end{aligned}$$

The values $x_1 \mapsto a_1 \ldots x_n \mapsto a_n$ which satisfy $SchemaType$ have been either generated or were part of the initial environment. Note that although the schema definition in the LP uses 'if' ($\Leftarrow$), by the CWA, this has the same effect as 'if an only if' ($\Leftrightarrow$).

The Z interpretation can similarly be represented by a set of bindings where

$$binding = \{ X_1 \mapsto \gamma(x_1), \ldots, X_n \mapsto \gamma(x_n) \}$$

The values $\gamma(x_i) \in \operatorname{ran} \rho_Z$ satisfy

$$(\mathcal{D}_Z[\![x_1 : \tau_1]\!] \rho_Z)\ \&\ \ldots (\mathcal{D}_Z[\![x_n : \tau_n]\!] \rho_Z)\ \&\ \mathcal{P}_Z[\![GCP]\!] \rho_Z.$$

AR1 can now be considered for schemas and is worth restating. If ϵ is a syntactic Z expression for a set of schema bindings then condition **AR1** must hold for a correct animation of Z in D_{LP}:

Approximation Rule 1 (AR1)

$$\gamma(\mathcal{S}_{LP}[\![\epsilon]\!] \rho_{LP}) \sqsubseteq \mathcal{S}_Z[\![\epsilon]\!] (\gamma \circ \rho_{LP}).$$

where

$$\epsilon = \{ X_1 : \tau_1 \ldots X_n \tau_n \mid CP \bullet \{ X_1 \mapsto x_1, \ldots X_n \mapsto x_n \} \}$$

The structural induction rule states that if it can be shown that **AR1** holds for syntactic variable $\epsilon = x$, then it also holds for syntactical expression $\epsilon = fx$, where in this case, f is a syntactic operator which forms a schema from tuple $\epsilon = (D, CP)$, where D is a declaration and CP is a predicate. We denote by f_Z, f_{LP} the interpretation in the Z domain and LP domain respectively of the syntactic expression fx. Thus the left hand side of **AR1** is

$$\gamma(\mathcal{S}_{LP}[\![[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP \,]]\!] \rho^o_{LP})$$
$$= \gamma(\mathcal{E}_{LP}[\![\{ x_1 : \tau_1;\ \ldots;\ x_n : \tau_n \mid GCP \bullet [Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)] \}]\!] \rho^o_{LP})$$

The right hand side of **AR1** is:

$$\mathcal{S}_Z[\![[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP \,]]\!] \rho^o_Z$$
$$= \mathcal{E}_Z[\![\{ x_1 : \tau_1;\ \ldots;\ x_n : \tau_n \mid GCP \bullet \{ X_1 \mapsto x_1, \ldots, X_n \mapsto x_n \} \}]\!] \rho^o_Z$$

These are *set comprehension*, which have been treated in Section 12. These interpret exactly where components are finite and complete. Where the answer sets is incomplete the LP interpretation underestimates.

14.2 Schema Conjunction and Disjunction

We now interpret syntactical objects such as $Sch \mathrel{\widehat{=}} Sch^1 \wedge Sch^2$ and $Sch \mathrel{\widehat{=}} Sch^1 \vee Sch^2$. Provided that Sch^1, Sch^2 have compatible declarations their conjunction and disjunction can be defined. These are modelled by conjunction and disjunction of the LP predicates of Sch^1, Sch^2 with lists of bindings appended. This does cause duplication but has not (so far) been found a practical problem. Suppose Sch^1 has predicate CP^1 and declaration sequence D^1 where

$$D^1 = X^1_1 : \tau^1_1;\ \ldots;\ X^1_n : \tau^1_n$$

modelled by Gödel list b_1:

$$[\ Bind^1_1(X^1_1, x^1_1), \ldots, Bind^1_n(X^1_n, x^1_n)\ ]$$

and Sch^2 has a predicate CP^2 and compatible declaration sequence D^2 where

$$D^2 = X^2_1 : \tau^2_1;\ \ldots;\ X^2_n : \tau^2_n$$

modelled by Gödel list b_2. Given that the characteristic predicates of Sch^1, Sch^2 are respectively

$$SchemaType(binding, Sch^1), SchemaType(binding, Sch^2)$$

then the characteristic predicate of Sch is

$$SchemaType(binding, Sch) \Leftrightarrow (binding = b_1 \frown b_2)\ \&$$
$$\mathcal{D}_{LP}[\![D^1;\ D^2]\!] \rho^o_{LP}\ \&\ \mathcal{P}_{LP}[\![GCP^1 \wedge GCP^2]\!] \rho^o_{LP}.$$

We show that this interprets exactly the Z interpretation where the program terminates, and provides an incomplete set of answer substitutions when the program fails to terminate.

We show this by showing that the above definition gives the same value(s) as the value(s) obtained by 'expanding out' the version of $Sch \mathrel{\widehat=} Sch^1 \land Sch^2$. which is interpreted in Z as

$$\begin{aligned}&\mathcal{S}_Z[\![Sch^1 \land Sch^2]\!]\rho_Z \\ &\quad = \mathcal{S}_Z[\![[\, D^1;\ D^2 \mid CP^1 \land CP^2\,]]\!]\rho_Z^0\end{aligned}$$

The above represents the declaration sequences before they are merged, so some repetitions would be expected. The above represents the declaration sequences before they are merged, so some repetitions would be expected. Sch is then expressed as a set comprehension in the usual way:

$$\begin{aligned}&\mathcal{E}_Z[\![\{x_1^1 : \tau_1^1;\ \ldots;\ x_n^1 : \tau_n^1;\quad x_1^2 : \tau_1^2;\ \ldots;\ x_n^2 : \tau_n^2 \mid GCP^1 \land GCP^2 \\ &\quad \bullet \{X_1^1 \mapsto x_1^1, \ldots, X_n^1 \mapsto x_n^1;\quad X_1^2 \mapsto x_1^2, \ldots, X_n^2 \mapsto x_n^2\}\}]\!]\rho_Z^o\end{aligned}$$

Then $Sch^1 \land Sch^2$ in the LP is:

$$\mathcal{S}_{LP}[\![Sch^1 \land Sch^2]\!]\rho_{LP}$$

where given that the characteristic predicates of Sch^1, Sch^2 are respectively

$$SchemaType(binding, Sch^1), SchemaType(binding, Sch^2).$$

then the characteristic predicate of Sch evaluates to

$$\begin{aligned}&SchemaType(binding, Sch) \Leftrightarrow \\ &(binding = [\ Bind_1^1(X_1^1, x_1^1), \ldots, Bind_n^1(X_n^1, x_n^1), \\ &Bind_1^2(X_1^2, x_1^2), \ldots, Bind_n^2(X_n^2, x_n^2)]\ ) \\ &\qquad \& \ \mathcal{D}_{LP}[\![D^1;\ D^2]\!]\rho_{LP}^o \ \& \ \mathcal{P}_{LP}[\![GCP^1 \land GCP^2]\!]\rho_{LP}^o\end{aligned}$$

which is the same as if the expression had been expanded first. The criteria for exactness or underestimation for each of these interpretations has already been discussed. In general, where each component of an expression is exact, the whole expression is exact,but where one component underestimates, the whole underestimates.

Schema disjunction is defined in a similar manner. If $Sch \mathrel{\widehat=} Sch^1 \lor Sch^2$ then the bindings are appended and the LP interpretations of the schema predicates are disjoined. In section [] the convention for naming variables is further refined, so that priming, input, output becomes apparent. The formalism is not explored here.

However the naming convention enables schema composition and piping to be accomplished. An outline is presented in [19].

14.3 Schema Reference in a Declaration

A declaration can contain a schema reference. If $x_i \in VAR, t, t_i \in expr, S_i \in NAME$, where Sch is a schema reference then recall that:

$$basic\_decl ::= x_1, \ldots, x_n : t \mid Sch$$

The interpretation of this in Z is that its declarations are merged with the declarations of the schema which reference it and its predicate is conjoined. Thus if

$$Sch_1 \mathrel{\widehat{=}} [\, X_1 : \tau_1;\ \ldots X_n : \tau_n;\ S \mid \ predicate\ of\ Sch_1 \,]$$

Then this is equivalent to $Sch_1 \mathrel{\widehat{=}} Sch \wedge Sch_2$ where

$$Sch_2 \mathrel{\widehat{=}} [\, X_1 : \tau_1;\ \ldots X_n : \tau_n \mid \ predicate\ of\ Sch_1 \,]$$

Thus if a schema Sch appears as a reference in the declarations of schema Sch_1, then this is treated as for schema conjunction above: Sch is removed from the declarations and conjoined to the predicate of Sch and its remaining declarations.

14.4 Binding Formation θ

The binding formation θSch can be used to form a binding. Its value depends on the environment. However we interpret it here in the same context as in the Unix file system case study. In that case study $\{Sch \bullet \theta Sch\}$ was constructed first and θSch was interpreted as a member of that set. The set $\{Sch \bullet \theta Sch\}$ in the LP is the 'same' set as $\mathcal{S}_{LP}[\![Sch]\!]\rho_{LP}$ however in this case it is a set *term* and not an answer set. It is the set comprehension S defined by

$$S = \{SchemaType(binding, Sch) \bullet binding\}$$

so that the binding formation $\theta Sch \in S$ where the code can be found in Appendix C of my Ph.D. thesis [18].

This means that if the computation terminates its interpretation is exact, and if one of the members of s fails to terminate then the output of the whole computation is $\perp^P_{LP}$.
Check!!

14.5 Axiomatic and Generic Definitions

Axiomatic and Generic Definitions require individual definitions; they are interpreted in such a way that they are exact where the computations terminate.

Axiomatic Definitions are modelled in the same way as a schema, and suitable *names* must be generated for them `Axiom1, Axiom2`.... They must then be conjoined to the schema which refer to them, as in the assembly case study in my thesis [18]. Their interpretation is the same as for schemas,

Generic definitions are treated in the same way as the parametrised definitions of partial function etc, i.e. by using parameters `a,b..` They are instantiated when the set is instantiated, and are defined by a predicate in the LP, as for Sequence in Appendix C.

References

[1] J. H. Andrews. Proof-Theoretic Characterisations of Logic Programming. In *Proceedings of the 14th International Symposium on the Mathematical Foundations of Computer Science, vol. 379 of LNCS*, pages 145–154. Springer, 1990.

[2] S. Austin and G. I. Parkin. *Formal Methods: A Survey (NPL).* DITC Office, Teddington, Middlesex, TW11 0LW, UK, March 1993.

[3] P. T. Breuer and J. Bowen. Towards Correct Executable Semantics for Z. In J. P. Bowen and J. A. Hall, editors, *Z User Workshop, Cambridge, 1994*, pages 185–209. Springer-Verlag, 1994.

[4] K. L. Clark, F. G. McCabe, and S. Gregory. IC-Prolog Language Features. In K. L. Clark and S. A. Tarnlund, editors, *Logic Programming*, pages 253–266. Academic Press, London, 1982.

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. 4th ACM Symposium on the Principles of Programming Languages*, pages 238–252, 1977.

[6] A. J. Dick, P. J. Krause, and J. Cozens. Computer Aided Transformation of Z into Prolog. In *Proceedings of the 4th Annual Z Users Meeting, Oxford University Computing Laboratory PRG*, pages 71–85. Springer-Verlag, December 1989.

[7] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *HandBook of Theoretical Computer Science: Formal Models and Semantics (Vol B)*, pages 635 – 674. Elsevier, 1990.

[8] I. Hayes, editor. *Specification Case Studies (Second Edition).* Prentice Hall International (UK) Ltd, 1993.

[9] D. Jackson. Alloy: A logical modelling language. In D. Bert, J.P. Bowen, S. King, and M. Waldn, editors, *ZB 2003: Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003*, pages 1–1. Lecture Notes in Computer Science 2651, Springer-Verlag, Germany, 2003.

[10] Michael Jackson. What can we expect from program verification. FACS Evening Seminar Series, BCS Headquarters, Southampton Street, London, UK, February 2007.

[11] K. Marriot and H. Søndergaard. Bottom-up Dataflow Analysis of Normal Logic Programs. *The Journal of Logic Programming*, 13:181–204, 1992.

[12] T. L. McCluskey and M. M. West. The automated refinement of a requirements domain theory. *Journal of Automated Software Engineering, Special Issue on Inductive Programming*, 8(2):193 – 216, 2001.

[13] T. Nicholson and N. Foo. A denotational semantics for prolog. *ACM Trans. on Programming Languages and Systems*, 11(4):650–665, 1989.

[14] J. M. Spivey. *The Z Notation: A Reference Manual (2nd ed.)*. Prentice Hall International (UK) Ltd, UK, 1992.

[15] S. Stepney. New horizons in formal methods. *The Computer Bulletin*, pages 24–26, January 2001.

[16] Mark Utting. Data structures for Z testing tools. In *FM-TOOLS 2000, Germany, July 2000*, 2000.

[17] M. M. West. Types and Sets in Gödel and Z. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95 – 9th International Conference of Z User's, September 1995, Limerick, Ireland*, pages 389–407. Lecture Notes in Computer Science 967, Springer-Verlag, Heidelberg, 1995.

[18] M. M. West. *Issues in Validation and Executability of Formal Specifications in the Z Notation*. PhD thesis, School of Computing, University of Leeds, 2002.

[19] M. M. West and B. M. Eaglestone. Software Development: Two Approaches to Animation of Z Specifications Using Prolog. *Software Engineering Journal*, 7(4):264–276, July 1992.