Using a Logic Programming Language to Animate Z: Correctness Criteria

Margaret West
School of Computing and Engineering, University of Huddersfield
14th July 2004

The is the second of two seminars describing work presented in my PhD thesis. This talk is concerned with the *correctness* aspects of animating Z using the Gödel logic programming language.

Background
In my previous talk I looked at the practical aspects of animating Z via a logic programming language (viz. Gödel). The rules (called 'structure simulation') were applied to two substantial case studies as 'proof of concept'.

The simulation rules were found to be practical, and to have a potential for real world applications. but lacked any formal framework for proving correctness. The next few slides examine approaches to establishing correctness of the method.

Correctness - Program Synthesis
• 'Deductive synthesis' is a method of obtaining a 'correct' program from a specification;
• A program is 'partially correct with regard to its specification' when it is derived logically from a specification [Hog84];
• The logic programming language Prolog is an example of a 'Horn Clause' program of the form $A \longleftarrow B_1 \ \& \ B_2 \ .. \ \& \ B_n$;
• A systematic method of obtaining a Horn Clause program from an arbitrary logic specification is the Lloyd-Topor transformation via logical equivalences [Llo87].

• It was found that for Prolog predicates involving set operations we need recursion;
• However the techniques for automatically producing recursive logic programs are still problematic [PP99].
• For these (and other reasons) the method was abandoned. and the method eventually chosen 'structure simulation' was described in the previous talk;
• The correctness criteria eventually chosen is *Abstract Approximation* and this is described in rest of this talk, together with a demonstration of correctness of 'structure simulation'.

Order of Work Presented

(1) Correctness - Abstract Approximation;
(2) Z Syntax and Interpretation(s);
(3) The Z domain;
(4) LP Domain;
(5) Loss of Information and Ordering;
(6) Sets - undefinedness and ordering;
(7) Proof Method - induction;
(8) Proof(s);
(9) File System Example.

(1) Correctness: Abstract Approximation

A different approach to correctness is *abstract approximation*, introduced by Breuer and Bowen [BB94] to provide a formal framework and some proof rules for the correct animation of Z.

• The method has similarities to *abstract interpretation*, [CC77];
• Abstract interpretation was initially used for static analysis of imperative programs.
• Cousot and Cousot related a concrete semantics with an abstract semantics;
• 'Abstract interpretation' formalised a commonly used technique - e.g. calculating the dimensions of a physical expression.

Example of Abstract Interpretation

Given a formula, e.g. the formula for the period, T of a simple pendulum of length l is

$$T = 2\pi(l/g)^{1/2}, \text{where } g \text{ is gravity.}$$

The dimension of length is $[L]$, the dimension of time $[T]$, and the dimensions of acceleration are $[L][T]^{-2}$ and '2π' is a scalar quantity and dimensionless. We use 'dimension calculus' as an abstraction. Thus the dimensions on the right hand side of the formula are

$$\left([L] \ / \ [L][T]^{-2}\right)^{\frac{1}{2}}$$

which evaluates to $[T]$, the dimension of the left hand side. This means that the formula is 'possibly_correct'. The only other answer we could have obtained is 'wrong'.

• The two answers are a way of ensuring that the interpretation is 'safe'. This means that if a property of the concrete interpretation is promised, then it is guaranteed.
• Since the original paper, the work of the Cousots has been extended to declarative languages, including the application to groundness analysis in logic programming [CC92].
• The Cousots have published many papers on abstract interpretation which can be found at http://www.di.ens.fr/c̃ousot/COUSOTpapers/.

Abstract approximation
- was suggested by [BB94] to determine the correctness of animations of Z. The idea is that Z is the 'concrete domain' and the logic programming domain D_{LP} is the 'abstract domain'.

• This compares the interpretation of Z syntactical objects in both the execution language (in our case the LP) and in Z.
• We compare the interpretation in the LP and in Z in 'equivalent' environments.
• A concretisation function γ relates the abstract with the concrete.

(2) Interpretation of Z Syntax
• The environments in the LP and in Z are denoted by: ρ_{LP}, ρ_Z respectively.
• The environments are *functions* from variable (names) VAR to domain values: $\rho_{LP} : VAR \nrightarrow D_{LP}$, $\rho_Z : VAR \nrightarrow D_Z$;
• $\rho_Z = \gamma \circ \rho_{LP}$;
Interpretations of *expressions* are denoted $\mathcal{E_{LP}}[\![\ldots]\!]\rho_{LP}$; $\mathcal{E_Z}[\![\ldots]\!]\rho_Z$
Example: The LP interpretation is according to the LP semantics: the syntactic expression: '$x + y$' in the LP is interpreted as a term and

$$\mathcal{E_{LP}}[\![x + y]\!]\{x \mapsto 2, y \mapsto 4\}$$

is implemented by

```
<- exp = x + y  & (x = 2) & (y = 4)
```

and evaluated by means of $\{x/2, y/4\}$ to '6'.

Comparison of Interpretations The Z interpretation is the interpretation we would expect if we had been evaluating the objects using set theoretic (ZF) considerations -it also evaluates to 6.
• For terminating computations (in the LP) the two interpretations should be equal and this can be formally expressed by the following equality, where 'ϵ' is a piece of syntax.

$$\gamma(\mathcal{E_{LP}}[\![\epsilon]\!]\rho_{LP}) = \mathcal{E_Z}[\![\epsilon]\!](\gamma \circ \rho_{LP}) \qquad *.$$

• To illustrate we apply this to the above example:

$$\begin{aligned} LHS &= \gamma(\mathcal{E_{LP}}[\![x + y]\!]\{x \mapsto 2, y \mapsto 4\}) = \gamma(6) = 6 \\ RHS &= \mathcal{E_Z}[\![x + y]\!](\gamma \circ \{x \mapsto 2, y \mapsto 4\}) \\ &= \mathcal{E_Z}[\![x + y]\!]\{x \mapsto \gamma(2), y \mapsto \gamma(4)\} = 6. \end{aligned}$$

• The syntax above is an example of an arithmetic expression - other syntactic expressions of Z to be interpreted are set expressions. The interpretation of schemas and predicate expressions are presented later.
• However some of the computations (and hence interpretations in the LP) will be non-terminating - so we need to extend the LP domain - see later;
• Since a comparison is to be made - the Z domain also needs extending to accommodate non-terminating computations.
• Formula * is a particular case for terminating computations. For non-terminating executions we introduce a 'bottom' element $\perp$ for each type in both Z and the LP. The next slide shows *approximation* in a pictorial fashion.

$$
\begin{tikzcd}
Env_{LP} \arrow[r, "\mathcal{E}_{\mathcal{LP}}[\![\epsilon]\!]\_"] \arrow[d, dashed, "\gamma \circ \_"'] & D_{LP} \arrow[d, dashed, "\gamma"] \\
Env_Z \arrow[r, "\mathcal{E}_{\mathcal{Z}}[\![\epsilon]\!]\_"'] & D_Z \arrow[u, phantom, "\sqsupseteq"]
\end{tikzcd}
$$

Figure 1: Approximation Diagram - where ρ is a variable

$Env_{LP} == VAR \twoheadrightarrow D_{LP}$ is the set of all possible LP environments associated with a specification, and $\rho_{LP} \in Env_{LP}$.
($Env_Z == VAR \twoheadrightarrow D_Z$ is defined similarly.)

• 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. The comparison is in the Z domain;
• Abstract approximation and abstract interpretation are similar in that they both represent an abstract and concrete interpretation of a piece of syntax;
• 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.

Z Syntax
• We consider the following four parts of the Z syntax: Expressions, Predicates, Declarations Schemas and Axiomatic Definitions denoted $expr, pred, decl, schema, axdef$ respectively;
• It is convenient to treat declarations as syntactic objects, as suggested in [BB94];
• Suppose the set of schema names is $NAME$, and the set of variable names (within a schema) are VAR and the set of given set names (and enumerated free types) is $GIVEN$. The following is an outline summary of the Z syntax to be interpreted.

Numerical and Set Expressions in Z Syntax

$$
\begin{aligned}
expr ::= {} & \mathbb{Z} \mid n \in \mathbb{Z} \text{ the integers and integer values} \\
& \mid t_1 + t_2 \mid t_1 - t_2 \ldots \mid \text{ an integer expression} \\
& \mid G_i \text{ where } \mathrm{G_i} \in \mathrm{GIVEN} \text{ a given set reference} \\
& \mid x_i \text{ where } \mathrm{x_i} \in \mathrm{VAR} \\
& \mid \{x_1 \ldots x_n\} \text{ an enumerated set} \\
& \mid (t_1, \ldots, t_n), \text{ a tuple} \\
& \mid t_1 \cup t_2 \mid t_1 \cap t_2 \mid \bigcup t \mid \ldots \\
& \qquad \text{set union, intersection, distributed union etc.} \\
& \mid \text{“}Enum\_Type ::= x_1 \mid \ldots \mid x_n\text{”} \\
& \qquad \text{where } \mathrm{x_i} \in \mathrm{VAR}, \mathrm{Enum\_Type} \in \mathrm{GIVEN} \\
& \qquad\quad \text{an enumerated free type} \\
\ldots\, etc
\end{aligned}
$$

NB Set comprehensions of the form $\{decl \mid pred \bullet term\}$ will be treated separately -later.

Declarations and Predicates

Recall that in Z everything is typed - '$x : X$' is a 'basic declaration' and that schemas can also be 'referenced' as part of a sequence of 'declarations':

$$\begin{array}{lll} basic\_decl & ::= & x_1, \ldots, x_n : t \mid Sch \\ decl & ::= & bd_1;\ \ldots;\ bd_n \text{ where } \mathrm{bd_i} \in \mathrm{basic\_decl.} \end{array}$$

Predicates have the following syntax, where
$p_1, p_2 \in pred$, $e_1, e_2 \in expr$:

$$\begin{array}{l} pred ::= p_1 \vee p_2 \mid p_1 \wedge p_2 \\ \qquad\qquad \mid \text{“} \forall\, d \mid p_1 \bullet p_2 \text{”} \mid \text{“} \exists\, d \mid p_1 \bullet p_2 \text{”} \text{ where d} \in \text{decl} \\ \qquad\qquad \mid e_1 = e_2 \ \mid\ e_1 \subseteq e_2 \ \mid e_1 \in e_2. \end{array}$$

Schema Syntax

A 'schema named Sch' is a declaration (d) followed by a predicate (p) :

```
┌─ Sch ─────────────────────────
│ d
├──────────
│ p
└───────────────────────────────
```

and this can be expressed $Sch \mathrel{\widehat=} [d \mid p]$

$$\begin{array}{lll} schema & ::= & \text{“}Sch \mathrel{\widehat=} \big[\, d \mid p \,\big]\text{”} \mid \theta Sch \mid \{Sch \bullet \theta Sch\} \\ & & \text{where } \ \mathrm{d} \in \mathrm{decl}, \mathrm{p} \in \mathrm{pred}, \mathrm{Sch} \in \mathrm{NAME} \\ & & \mid \text{“}Sch \mathrel{\widehat=} Sch^1 \wedge Sch^2\text{”} \mid \text{“}Sch \mathrel{\widehat=} Sch^1 \vee Sch^2\text{”} \\ & & \text{where } \mathrm{Sch}, \mathrm{Sch}^1, \mathrm{Sch}^2 \in \mathrm{NAME} \\ axdef & ::= & \big[\, d \mid p \,\big], \text{ where } \ \mathrm{d} \in \mathrm{decl}, \mathrm{p} \in \mathrm{pred}. \end{array}$$

(3) Z Domain

• The standard Z domain consists of:
(1) Integers and sets of integers;
(2) Tuples ;
(3) Enumerated free types (enumerated sets);
(4) Booleans $Bool_Z = \{tt, ff\}$ (etc.)
• In order to animate given sets the user of the animator is required to instantiate them with suitable values and these are added to the specification - as enumerated types.
• The variable and schema names will subsequently be interpreted as constants in D_{LP}, (which means confining them to upper case).

Schema Bindings

• Consider a schema $Sch \mathrel{\widehat=} [d \mid p]$ whose declaration d involves n variables named $X_i,\ i = 1 \mathrel{..} n$ with their types.
• The 'binding'

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

provides values of x_i which satisfy p.
• This object is represented more simply by the symbol table

$$\{X_1 \mapsto x_1, \ldots, X_n \mapsto x_n\}$$

which is part of the existing syntax of Z. Notice we use X, x for variable name and value. However - in what follows (where possible) we shall just use x as denoting a variable value.

(4) The Logic Programming Domain

• The proposed abstract domain, D_{LP}, includes representations of integer values, instantiated values, tuples, bindings and sets;
• The interpretation is of schemas and results in an *output* - and is confined to schema bindings, as described in seminar 1. Thus evaluations of arithmetical, set and expressions *other than* these, take place *as part of a program execution to determine or check schema bindings*.
• n-Tuples are represented by functions of arity n and sets are represented both as *terms* and as answer sets. (See later.)
• 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 (and therefore upper case).

The LP Domain - Output of expressions

$$
\begin{aligned}
D_{LP} ::= \; & m, m \text{ an integer} \\
& \mid g_i^k, \;\; \text{where each } \mathrm{g_i^k} \text{ is base } \mathrm{G^k} \\
& \mid Tn(x_1, \ldots, x_n) \text{ where } \mathrm{x_k} \in \mathrm{D_{LP}}, \text{ a tuple} \\
& \mid \{x_1, \ldots, x_n\} \text{ where } \mathrm{x_k} \in \mathrm{D_{LP}}, \mathrm{x_k} \neq \bot, \text{ an set term} \\
& \mid Bind_i(X_i, x_i) \text{ where } \mathrm{x_i} \in \mathrm{D_{LP}}, \mathrm{X_i} \in \mathrm{VAR} \\
& \qquad \text{a single variable binding} \\
& \mid [\; b_1, \ldots, b_n \;], \text{ where each } \mathrm{b_i} \text{ is a variable binding} \\
& \qquad \text{this is a schema type} -- \text{a list of variable bindings}
\end{aligned}
$$

Set Objects in the LP

• Set terms: in the LP (finite) set terms are represented (first of all) by terms:

$$\{x_1, x_2, \ldots, x_n\}, \text{ or } x_1 \circ (\ldots(x_n \circ \; Null))))$$

where each x_i is itself a term and $Null = \varnothing$. (See [MW85].)
• Sets of answer substitutions: recall that for some schema Sch, a binding is denoted in the LP:

$$\theta Sch = [Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)]$$
$$\text{where } \mathrm{x_k} \in \mathrm{D_{LP}}, \mathrm{X_i} \in \mathrm{VAR}, \mathrm{Sch} \in \mathrm{NAME}$$

and that the answers to a query concerning the characteristic predicate of Sch provide a set of answers which depends on the values instantiated.

Example: A small file system involves a single given set:

$[FileId]$

of file identifiers. There are a maximum number of files:

$\mid \; MaxFiles : \mathbb{N}_1$

We define the file system in terms of its state variables which are *Files*, and *Count*, a count of the files. The former is a finite subset of Files and the latter a number.

FileSys
$Files : \mathbb{F}\, FileId$
$Count : 0 \,..\, MaxFiles$

$\#Files = Count$

Example:

```
% test of schema FileSys
[Demo2] <- SchemaType(b, FileSys).
% the initial state
b = [Bind1(Files,{}),Bind2(Count,0)] ?
% second schema binding - a further state
b = [Bind1(Files,{F1}),Bind2(Count,1)] ?
% % etc
```

In this case all states will be generated eventually and the possible inputs which are associated - all answers terminate.

The concretisation mapping γ is defined next.

Concretisation Function γ

$\gamma(m)$	$=$	m, m an integer
$\gamma(g)$	$=$	$g, g \in G$,
		a member of a given set G
$\gamma(\{x_1, \ldots, x_n\})$	$=$	$\{\gamma(x_1), \ldots, \gamma(x_n)\}$,
$\gamma(Tn(x_1, \ldots, x_n))$	$=$	$(\gamma(x_1), \ldots, \gamma(x_n))$, a tuple
$\gamma([b_1 \ldots b_n])$	$=$	$\{X_1 \mapsto \gamma(x_1), \ldots, X_n \mapsto \gamma(x_n)\}$
where $\mathrm{b_i = Bind_i(X_i, x_i)}$,		a single schema binding
$\gamma(\perp)$	$=$	$\perp$ non $-$ termination see later

(5) Undefinedness and Ordering

In order to accommodate non-terminating executions, both Z and the LP domains are extended by the inclusion of a '$\perp$' element for each type. For example the booleans:

$$Bool = \{tt, \mathit{ff}, \perp\}$$

This means that **ALL** functions are total. (Z can similarly be extended.) '$\perp$' also denotes a value of an unknown variable during some state of the program - for example initially.
For example:
if VAR is $\{X, Y, Z\}$ with X, Z both instantiated to 0 and Y unknown then the LP environment is '$\{X \mapsto 0, Y \mapsto \perp, Z \mapsto 0\}$'.
The equivalent Z environment is then
'$\{X \mapsto \gamma(0), Y \mapsto \gamma(\perp), Z \mapsto \gamma(0)\} = \{X \mapsto 0, Y \mapsto \perp, Z \mapsto 0\}$'.

Ordering

There is an imposed ordering in respect of all types of domain elements. For example if a, b are integers or members of given sets then the ordering $\sqsubseteq$ is

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

The ordering relation works co-ordinatewise on tuples. Since the formalisation involves *set terms*, we need to consider the ordering with regard to sets

(6) Complete and Incomplete Sets

Sets can be 'complete' but contain incomplete elements:

Complete Sets: For example $\{1, 2, 3, \perp, 4\}$.

The ordering relation 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, $\{1, 2, 3, \perp, 4\} \sqsubseteq \{1, 2, 3, 4, 5\}$.

Incompleteness in Sets: LP examples

Sets can also be 'incomplete'. The notation (by [BB94]) is to 'tag' them, for example $\{1, 2, 3, 4\}_{\cup\perp}$.

(Thus $s_{\cup\perp}$ denotes an incomplete set.) The ordering for 'incomplete sets' is as follows:

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

We 'refine' an incomplete set if we complete it or (in addition) we add some more elements: For example,

$\{1, 2, 3, 4\}_{\cup\perp} \sqsubseteq \{1, 2, 3, 4, 5\}$.

$\{1, 2, 3, 4\}_{\cup\perp} \sqsubseteq \{1, 2, 3, \perp, 4, 5\}$.

In general the incomplete sets are 'non-standard' with respect to ZF.

For example use the definition to compare $\{1, 2, 3, 4\}_{\cup\perp}$ and $\{1, 2, 3, \perp, 4\}_{\cup\perp}$.

Two sets are 'equal' – however they do not have the same elements. (It could be said that the two sets contain the same 'information' – see [GS90].)

Incomplete Sets – examples

The Z and LP domains are both extended to contain incomplete and undefined elements. The LP domain contains 'sets as terms' and also 'answer sets'.

Set Terms

The following are two examples of set 'incompleteness'. In the first case the set contains $\perp$:

$$(\perp \circ (x_1 \circ (\ldots (x_n \circ \mathit{Null}))))$$

In the second case when a computation of a set term fails to terminate, in an attempt to evaluate an infinite set for example, we obtain the set denoted by:

$$(x_1 \circ (\ldots (x_n \circ \ldots))).$$

In both cases the set evaluates to '$\perp$' since functions are strict. This bottom element is designated $\mathit{Null}_\perp$ to distinguish it as a set.

We have, for all set a:

$$a \cup \mathit{Null}_\perp = \mathit{Null}_\perp \cup a = \mathit{Null}_\perp$$
$$a \cap \mathit{Null}_\perp = \mathit{Null}_\perp \cap a = \mathit{Null}_\perp$$
$$a \sqsubseteq \mathit{Null}_\perp$$

(The LP implementation of such an output may be a warning message.) Note that the above applies to terms in an execution which fails to terminate, rather than to terms in their initial program state, for these may very well be undefined.

Answer Sets
An answer set can output some results then fail with an error message. An example would be a schema

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{UnDef}} \\ \hline
X, Y : \mathbb{N} \\ \hline
Y \in \{1,2,3\} \\
((X = 1) \vee (X = 3) \vee (\{X,1,2,3\} = \{1,2,3,4\})) \\ \hline
\end{array}$$

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
```

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 an incomplete set (as above) and is denoted ,

$$\{b_1, b_2\}_{\cup\perp}$$

where b_1, b_2 are the two schema bindings output before the error message.

(7) Proof Method - Structural Induction
Figure 1 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 (**)[a]

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

The strategy for proof involves structural induction and the next slide presents 3 conditions (derived by [BB94]) which form the basis of a structural induction rule.

[a] We need a different approximation rule for 'incomplete sets' but that will not be considered here

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.

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

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

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

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:

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

Proof (for reader)

• This means that if it can be shown that ** 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.

Induction is over each Z construct and is shown in full in the thesis. Only the novel or most salient parts are presented here. The induction takes place in the following order:

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

(8) Proofs – Induction
Example (i) Integers
MaxInt, *MinInt*, are the largest positive and negative integers available. Any attempt to exceed them will cause the computation to terminate. Thus for $m \in \mathbb{Z}$:

$$\mathcal{E}_{\mathcal{LP}}[\![\mathrm{m}]\!]\rho_{\mathrm{LP}} = \mathrm{m} = \mathcal{E}_{\mathcal{Z}}[\![\mathrm{m}]\!]\rho_{\mathrm{Z}}, \quad -\mathrm{MinInt} \le \mathrm{m} \le \mathrm{MaxInt}$$
$$\mathcal{E}_{\mathcal{LP}}[\![\mathrm{m}]\!]\rho_{\mathrm{LP}} = \perp, \mathrm{m} < -\mathrm{MinInt} \text{ or } \mathrm{m} > \mathrm{MaxInt}.$$

($\perp$ 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 $\gamma(\perp) = \perp$:

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

Example (ii) Sets of integers and of given set elements s
For sets of **integers**, the result is similar to the result for single integers: if the memory bounds are exceeded, the computation results in $\perp$ and underestimates the Z interpretation;

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

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

Integer and Set Expressions
- For any computation, if the memory bounds are exceeded, the computation results in $\perp$ and underestimates the Z interpretation;
- For integer expressions, such as addition, subtraction, two cases are considered, the case where all integers are within the *MinInt*, *MaxInt* of the LP implementation, results in an exact approximation.
- In the case where the integers exceed these bounds, the LP evaluates to '$\perp$' and always underestimates the Z interpretation.
- The case is similar for set expressions - we use 'set union' to demonstrate the proof.

Example (iii) Set Union: '$x_1 \cup x_2$'
For terminating computations we use the three conditions.
Condition 1: recursive nature of LP interpretation. The function 'f_{LP} is 'union' acting on the tuple (x_1, x_2) and supposing that $x_1 \mapsto a_1, x_2 \mapsto a_2 \in \rho_{LP}$. In Gödel, 'union' is provided by a function '+' (denoted by $\cup_{LP}$)
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.
Condition 1 becomes:

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

which will hold for set operations for terminating computations.

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

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

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:

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

In other words the computation is exact for terminating computations.

For non-terminating computations we interpret figure 1 directly.
For example suppose:
$x_1 \mapsto \{a, c, \perp\}$, $x_2 \mapsto \{c, d\} \in \rho_Z$, then these become
$x_1 \mapsto \{\gamma(a), \gamma(c), \gamma(\perp)\}$, $x_2 \mapsto \{\gamma(c), \gamma(d)\} \in \rho_{LP}$.
Each of the set expressions is a term so
$x_1 \mapsto \{\gamma(a), \gamma(c), \gamma(\perp)\} = Null_\perp$
and the left hand side of ** for $\epsilon = x_1 \cup x_2$ is

$$\gamma\ (Null_\perp \cup_{LP} \{\gamma(c), \gamma(d)\}) = \gamma\ (Null_\perp) = \varnothing_{\cup\perp}.$$

The RHS is $\{a, c, \perp\} \cup_Z \{c, d\} = \{a, c, \perp, d\}$ (assuming ZF) which will always exceed $\varnothing_{\cup\perp}$. The result is similar for 'incomplete sets'.

Since we have established conditions (1 – 3) for complete sets and ** *directly* for incomplete or infinite sets, then ** holds when 'f' is '$\cup$'.

Predicate Expressions

• We denote by $\mathcal{P}_{\mathcal{LP}}[\![p]\!]\rho_{LP}$ the interpretation of syntactic predicates p in LP domain.
A predicate evaluates to $\perp$ when a program flounders or fails to terminate during its evaluation. Thus if

$$Bool_Z = \{tt, ff, \perp\},\ Bool_{LP} = \{true, false, \perp\}$$

then

$$\gamma(true) = tt,\ \gamma(false) = ff,\ \gamma(\perp) = \perp.$$

We also have:

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

Infix Predicates

These are equality, subset, membership:
$=$, $\subseteq$, $\in$.
• Predicates can both provide a boolean answer *and* update the environment, from ρ_{LP}, to ρ'_{LP} (say).
• It can happen that the environment can be updated in a number of different ways, thus providing a set of answer substitutions.
• As a result of the *resolution inference rule* of logic programming, the update is extended to all literals conjoined to the literal being evaluated.
• It is presented in the form of three constraint satisfaction rules. (See next slide.)

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:

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

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:

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

where $\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1, x_2 \mapsto a_2\}$.

```
%% An  example which illustrates both properties
[Lib] <- ([1, 2, 3, y] = [1, x, 3, 4]) & z = x + y.
x = 2,
y = 4,
z = 6 ? ;
```

An extension of these properties is the case where x_1 can take many values. The different values contribute to different answer substitutions. Examples are subset and membership. We call this:

Constraint Property 3

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

where

$$\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1\} \vee \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_2\} \vee \ldots$$
$$\vee\ \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_n\} \text{where } \rho'_{\mathrm{LP}} \in \mathrm{Env}_{\mathrm{LP}}.$$


```
%%  For example:
[Lib] <- z In {2, 3, 4} & y = 2 *z.

y = 4,
z = 2 ? ;

y = 6,
z = 3 ? ;

y = 8,
z = 4 ? ;
No
```

The same constraint properties can be extended to the Z environment.

Summary - Structural Induction: $x_1 \mathcal{I} x_2$

• Assuming that the execution terminates, and x_1, x_2 take unique values, we can summarise, thus.

• There are three cases for x_1, x_2, depending on whether or not x_1, x_2 are defined prior to execution of equality function and in each case * is true.

Equality: If 'f' is the syntactic predicate = for variable (x_1, x_2):
  * holds for $(x_1 = x_2)$;

Subset If 'f' is the syntactic predicate $\subseteq$ for variable (x_1, x_2):
  * holds for $(x_1 \subseteq x_2)$;

Membership * 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.

Variable Declarations

Variable declarations occur within (for example) schemas:

$\left[\, d \mid p \bullet t \,\right]$ where d is a declaration, p is a predicate and t a term. d is of the form:

$$x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n.$$

Other examples include *set comprehensions*, and *quantified expressions.*

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.

Interpretation of Declarations

• $\mathcal{D}_{\mathcal{LP}}$ 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 evaluation function is built recursively and interprets in a similar manner to the infix predicates defined previously, for variable values generated by the declarations will update the environment.
• The declarations are treated as predicates. (The declarations considered here do not include schema references, for these are treated separately.)
• τ is a *set:*

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

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

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

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

'$\tau : \mathbb{P}\,\tau'$ ' 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;
• τ is a *Cartesian Product*, $\tau_1 \times \tau_2$:

$$\begin{aligned}\mathcal{D}_{\mathcal{LP}}[\![x : \tau_1 \times \tau_2]\!]\rho_{LP} = \mathcal{P}_{\mathcal{LP}}[\![x = (x_1, x_2)]\!]\rho_{LP} \\ \& \ \mathcal{P}_{\mathcal{LP}}[\![x_1 \in \tau_1]\!]\rho_{LP}) \ \& \ \mathcal{P}_{\mathcal{LP}}[\![x_2 \in \tau_2]\!]\rho_{LP}).\end{aligned}$$

'T2' captures a representation of ordered pair (as an example of a tuple) in the LP. In our Gödel library this is 'OrdPair'.
The proof of correctness for declarations is based on an equivalence between a sequence of declarations and an expression involving distributed union.

Set comprehension

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

• Each $x_i : \tau_i$ provides a value which contributes to the tuple $(x_1, \ldots x_n)$ which is used to evaluate t.
• If $s = \{d \mid p \bullet t\}$ is a syntactical set comprehension it is interpreted:
D_{LP} as $\mathcal{E}_{\mathcal{LP}}[\![s]\!]\rho_{LP}$
D_Z as $\mathcal{E}_{\mathcal{Z}}[\![s]\!]\rho_Z$.
Each of these interpretations is respectively dependent on its constituent $\mathcal{D}_{\mathcal{LP}}, \mathcal{D}_{\mathcal{Z}}$ where the declarations act as generators for s.
Since declarations in the LP are treated as predicates, then the set comprehension of s is interpreted in the LP:

$$\mathcal{E}_{\mathcal{LP}}[\![s]\!]\rho_{LP} = \{\mathcal{D}_{\mathcal{LP}}[\![d]\!]\rho_{LP} \ \ \& \ \ \mathcal{P}_{\mathcal{LP}}[\![p]\!]\rho_{LP} \bullet \mathcal{E}_{\mathcal{LP}}[\![t]\!]\rho_{LP}\}.$$

This way of writing a set comprehension in the LP is chosen so that it resembles set comprehension in Z. It differs from the way it would be coded in Gödel `s = {x : p(x)}`.
The environment ρ_{LP} inside the comprehension is the variable which acts as a set generator, for recall that
$\mathcal{D}_{\mathcal{LP}}[\![x : \tau]\!]\rho_{LP} = \mathcal{P}_{\mathcal{LP}}[\![x \in \tau]\!]\rho_{LP}$.
A similar interpretation is true for D_Z.

Interpretation of Schemas

• Suppose that the syntactic object *schema* are interpreted in the LP and in Z by $\mathcal{S}_{\mathcal{LP}}$, $\mathcal{S}_{\mathcal{Z}}$ 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 $CP\ ::=\ CP_1 \wedge \ldots \wedge CP_m$.

• Recall that X_i is a variable name and that the output (interpretation) of the schema is a set of variable bindings, where each binding is denoted respectively by $Bind_i(X_i, x_i)$ and $X_i \mapsto x_i$.

• *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.

• *A set of schema bindings* of *Sch* can be represented in Z (as suggested in [BB94]) by a set expression:

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

We assume that the set of bindings is constrained by an initial imposed environment ρ^o where ρ^o can contain defined values of all the schema variables (as in the case of the assembler or just some of them (as in the case of the Unix file system).

The interpretation $\mathcal{S}_{\mathcal{Z}}[\![Sch]\!]\rho^o_Z$ of the schema $Sch \mathrel{\widehat=} [\, D \mid CP \,]$ is the interpretation of a set expression:

$$\begin{aligned}&\mathcal{S}_{\mathcal{Z}}[\![\,[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP \,]\,]\!]\rho^o_Z\\ &= \mathcal{E}_{\mathcal{Z}}[\![\{X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP\\ &\qquad \bullet \{X_1 \mapsto x_1, \ldots, X_n \mapsto x_n\}\}]\!]\rho^o_Z,\end{aligned}$$

The interpretation in the LP is

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

for $[Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)]$ replaces $\{X_1 \mapsto x_1, \ldots, X_n \mapsto x_n\}$. The interpretation of schemas and schema expressions is in terms of a *characteristic predicate*, providing a single binding for a schema expression.

Characteristic Predicate for a Schema Expression

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

evaluates in the LP to bindings of variable names to values. where the initial environment ρ^0 acts as a generator for other, possible environments: ρ_{LP} where each enhanced environment $\rho_{LP} \in Env_{LP}$ satisfies

$$\begin{aligned}&\mathcal{D}_{\mathcal{LP}}[\![X_1 : \tau_1;\ \ldots;\ X_n : \tau_n]\!]\rho^0_{LP}\ \&\ \mathcal{P}_{\mathcal{LP}}[\![CP]\!]\rho^0_{LP} =\\ &\qquad \mathcal{D}_{\mathcal{LP}}[\![x_1 : \tau_1;\ \ldots;\ x_n : \tau_n]\!]\rho_{LP}\ \&\ \mathcal{P}_{\mathcal{LP}}[\![GCP]\!]\rho_{LP}.\end{aligned}$$

The characteristic schema predicate of *Sch* is as follows:

$$\begin{aligned}&SchemaType(binding, Sch) \Leftrightarrow\\ &\quad (binding = [Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)])\ \&\\ &\qquad \mathcal{D}_{\mathcal{LP}}[\![x_1 : \tau_1;\ \ldots;\ x_n : \tau_n]\!]\rho_{LP}\ \&\ \mathcal{P}_{\mathcal{LP}}[\![GCP]\!]\rho_{LP},\end{aligned}$$

where GCP is defined as CP where all the free occurrences of $X_1 \ldots X_n$ are replaced by $x_1 \ldots x_n$

• Each x_i which satisfy $SchemaType$ has either been generated or was part of the initial environment.
• The generated values have been obtained via the application of **Constraint Properties 1 - 3** defined previously. 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

$$\mathcal{D}_{\mathcal{Z}}[\![D_1;\ \ldots;\ D_n]\!]\rho_Z \wedge \mathcal{P}_{\mathcal{Z}}[\![GCP]\!]\rho_Z.$$

It is worth investigating how the above would apply to a schema, and the one chosen is $FileSys$.

(9) Interpretation of $FileSys$
This can be written horizontally as:

$$FileSys \mathrel{\widehat{=}} [\, Files : \mathbb{F}\, FileId;\ Count : 0 \mathinner{..} MaxFiles \mid \#Files = Count \,].$$

Suppose that $MaxFiles = 10$ is a value provided by the animation user, and that $FileId$ is instantiated as $\{F1, F2, F3\}$. Then

$$\mathcal{D}_{\mathcal{LP}}[\![Files : \mathbb{F}\, FileId;\ Count : 0 \mathinner{..} MaxFiles]\!]\rho^o_{LP}$$
$$= \mathcal{P}_{\mathcal{LP}}[\![files \subseteq \{F1, F2, F3\}]\!]\rho^o_{LP}\ \&\ \mathcal{P}_{\mathcal{LP}}[\![count \in \{0 \mathinner{..} 10\}]\!]\rho^o_{LP}.$$

If we substitute these values, a binding for $FileSys$ is given by:

$$SchemaType(binding, FileSys) \Leftrightarrow$$
$$(binding = [Bind_1(Files, files), Bind_2(Count, count]) \ \&$$
$$\mathcal{P}_{\mathcal{LP}}[\![files \subseteq \{F1, F2, F3\}]\!]\rho^o_{LP}\ \&\ \mathcal{P}_{\mathcal{LP}}[\![count \in \{0 \mathinner{..} 10\}]\!]\rho^o_{LP}\ \&$$
$$\mathcal{P}_{\mathcal{LP}}[\![\#files = count\ ]\!]\rho^o_{LP}.$$

Initially, $\rho^o_{LP} = \{files \mapsto \perp, count \mapsto \perp\}$.
During the execution, $files$ is of type $\mathbb{F}$ so becomes evaluated through the interpretation of its declaration:

$$files \subseteq \{F1, F2, F3\}.$$

Similarly for $count$.
A binding of $FileSys$ can be expressed:

$$binding = [Bind_1(Files, files), Bind_2(Count, count]\ \&$$
$$\mathcal{P}_{\mathcal{LP}}[\![files \subseteq \{F1, F2, F3\}]\!]\rho^o_{LP}\ \&\ \mathcal{P}_{\mathcal{LP}}[\![count \in \{0 \mathinner{..} 10\}.]\!]\rho^o_{LP}\ \&$$
$$\mathcal{P}_{\mathcal{LP}}[\![\#files = count\ ]\!]\rho^o_{LP}.$$

Thus if $files$ evaluates to $\{F1\}$ (say), then in order to satisfy the schema predicate *and* its declaration, $count$, evaluates to '1' since

$$\#files = count\ \&\ count \in \{0 \mathinner{..} 10\}.$$

Substituting these values yields:

$$binding = [Bind_1(Files, \{F1\}), Bind_2(Count, 1]\ \&$$
$$\mathcal{P}_{\mathcal{LP}}[\![\{F1\} \subseteq \{F1, F2, F3\}]\!]\rho_{LP}\ \&\ \mathcal{P}_{\mathcal{LP}}[\![1 \in \{0 \mathinner{..} 10\}]\!]\rho_{LP}\ \&$$
$$\mathcal{P}_{\mathcal{LP}}[\![1 = 1]\!]\rho_{LP},$$

where $\rho_{LP} = \{files \mapsto \{F1\}, count \mapsto 1\}$ is the enhanced value of the environment and

$$binding = [Bind_1(Files, \{F1\}), Bind_2(Count, 1)]$$

which was one of the values actually obtained. The full set of bindings can be obtained from the full set of answer substitutions, as was indicated previously.

Initial Environment

• For the 'complete' assembler, the variables were initially all ground, so that ρ^o_{LP} contains no undefined values and the environment is unaltered: $\rho_{LP} = \rho^o_{LP}$, where similarly $\rho_Z = \rho^o_Z$.
• For the Unix file system, and for the two-phase assembler, for each of the schemas considered some variables are ground in ρ^0_{LP}, and some are determined by the execution.

Approximation for Schemas

** can now be considered for schemas: $Sch \mathrel{\widehat=} [\, D \mid CP \,]$ where 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 ** is

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

The right hand side of ** is

$$\begin{aligned}&\mathcal{S}_{\mathcal{Z}}[\![[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP \,]]\!]\rho^o_Z\\ &= \mathcal{E}_{\mathcal{Z}}[\![\{x_1 : \tau_1;\ \ldots;\ x_n : \tau_n \mid GCP \bullet \{X_1 \mapsto x_1, \ldots, X_n \mapsto x_n\}\}]\!]\rho^o_Z.\end{aligned}$$

This is a *set comprehension*, which has been treated previously. These interpret exactly where components are finite and complete (as in the case of *FileSys*).

For incomplete answer sets the LP underestimates as in the case of *UnDef*.

Conclusions

• A set of translation rules from Z to Gödel was presented in the first talk and shown to be practical. In this talk the rules have been provided with a formal basis.
• Correctness Criteria have been applied and the rules shown to be correct.
• The potential of the rules and the animating language, Gödel for contributing to an effective tool have thus been demonstrated.

Further work

1. Extension of the rules and proofs;
2. The development of meta-interpreters and techniques of inductive logic to trace and correct flaws in the specification as in [MW01];
3. Automation of the rules;
4. A strategy for selecting test cases for animation, including (where possible) the automatic generation of test cases;
5. An interesting area of work would be the investigation of a functional logic language for animation purposes as suggested in [WDK98].

References

[BB94] P. T. Breuer and J. Bowen. Towards Correct Executable Semantics for Z. In *Z User Workshop, Cambridge, June 1994*, pages 185–209. Springer-Verlag, 1994.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fix-points. In *Proc. 4th ACM Symposium on the Principles of Programming Languages*, pages 238–252, 1977.

[CC92] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *The Journal of Logic Programming*, 13:103–179, 1992. The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see http://www.di.ens.fr/c̃ousot.

[GS90] 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.

[Hog84] C. Hogger. *Introduction to Logic Programming.* Academic Press, London, 1984.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming (Second, Extended Edition).* Springer-Verlag, Berlin, 1987.

[MW85] Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming, Vol 1: Deductive Reasoning.* Addison-Wesley, USA, 1985.

[MW01] T. L. McCluskey and M. M. West. The automated refinement of a requirements domain theory. *Journal of Automated Software Enginnering, Special Issue on Inductive Programming*, 8(2):193 – 216, 2001.

[PP99] A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming*, 41:197–230, 1999.

[WDK98] M. Winikoff, P. Dart, and E. Kazmierczak. Rapid prototyping using formal specifications. In *Proceedings of the 21st Australasian Computer Science Conference, ACSC'98*, pages 279–293. Springer, 1998.