Animate Z using the CEGEDIL Logic Programming Language. This talk is concerned with the correctness aspects of my PhD thesis. The is the second of two seminars describing work presented in my 14th July 2004 School of Computing and Engineering, University of Huddersfield Margaret West

Animate Z: Correctness Criteria Using a 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 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.
via logical equivalences [1987].

An arbitrary logic specification is the Lloyd-Taylor transformation

A systematic method of obtaining a Horn clause program from

Horn clause program of the form $A \rightarrow B_1 \land B_2 \land \ldots \land B_n$.

The logic programming language Prolog is an example of a Horn

program which is partially correct with regard to its specification.

when it is derived logically from a specification [Hog84].

A program is partially correct with regard to its specification

Deductive synthesis is a method of obtaining a correct program

Correspondence - Program Synthesis
With a demonstration of correctness of structure simulation.

Approximation and this is described in rest of this talk, together

The correctness criteria ultimately chosen is Abstract

in the previous talk:

the method eventually chosen, structure simulation was described

For these (and other reasons) the method was abandoned, and

Logic programs are still problematic [PP99].

However, the techniques for automatically producing recursive

we need recursion;

It was found that for Prolog predicates involving set operations
(1) Correctness - Abstract Approximation

(2) Z Syntax and Interpretation(s)

(3) The Z Domain

(4) LP Domain

(5) Loss of Information and Order

(6) Sets - Undenumerable and Ordinate

(7) Proof Method - Induction

(8) Proof(s)

(9) File System Example

Order of Work Presented
E.g. calculating the dimensions of a physical expression.

Abstract interpretation, a commonly used technique,

- Abstract interpretation is formally introduced a semantics.
- Consist and consist related a concrete semantics with an abstract imperative programs.

Abstract interpretation was initially used for static analysis of a framework and some proof rules for the correct approximation of Z.

A different approach to correctness is abstract approximation:

1. Correctness: Abstract Approximation
we could have obtained is "wrong."

means that the formula is possibly correct. The only other answer
which evaluates to the left hand side. This

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

Thus the dimensions on the right hand side of the formula are
and dimensions. We use dimension calculus as an abstraction.
dimensions of acceleration are \( \frac{1}{2} \) and, \( \frac{1}{2} \) is a scalar quantity.

The dimension of length is the dimension of time, \([T]\) and the

\[ \frac{g}{2} \left( \frac{6}{1} \right) \frac{1}{2} \frac{1}{2} = L \]

where \( g \) is gravity.

Thus the pendulum of length \( L \) is

Given a formula of the form for the period, \( T \) of a simple

Example of Abstract Interpretation
interpretation which can be found at

The Coq tools have published many papers on abstract

Groundness analysis in Logic Programming [CC92].
e xtended to declarative languages, including the application to

Since the original paper, the work of the Coq tools has been

is promised, then it is guaranteed.

safe. This means that if a property of the concrete interpretation is

The two answers are a way of ensuring that the interpretation is

http://www.diena.fr/Coqform/COQSOFPapers
A concretisation function relates the abstract with the concrete.

We compare the interpretation in the TP and in \( Z \) in equivalent environments.

The execution language (in our case the TP) and in \( Z \).

This compares the interpretation of syntactical objects in both the logic programming domain \( D \) is the abstract domain and automations of \( Z \). The idea is that \( Z \) is the concrete domain and was suggested by \( BB94 \) [BB94] to determine the correctness of abstract approximation.
and evaluated by means of\{\forall/\forall,\exists/x\}$

$$(\forall = \forall) \forall (\exists = x) \forall \land x = \exp \rightarrow$$

is implemented by

$${\forall \leftarrow \forall, \exists \leftarrow x} \llbracket \forall + x \rrbracket_{dJ3}$$

and

the syntactic expression \(\forall + x\)' in the LP is interpreted as a term in the LP. Interpretation is according to the LP semantics:

Example: The LP interpretation is denoted by

Interpretations of expressions are denoted

\[ p_{LP} \circ \land = zd \]

domain values: \(zd, p_{LP} \land = zd\)

The environments are functions from variable (names) to \(\forall\)

respectively:

\(zd, p_{LP}\)

The environments in the LP and in $\mathcal{Z}$ are denoted by

(2) Interpretation of $\mathcal{Z}$ Syntax
\[ 6 = \{(\forall) \leftarrow \bar{h}, (\exists) \leftarrow x\} [\bar{h} + x] \mathcal{Z} \mathcal{J} = \]
\[ (\forall \leftarrow \bar{h}, \exists\leftarrow x) \circ \lambda [\bar{h} + x] \mathcal{Z} \mathcal{J} = \text{SHR} \]
\[ 9 = (9) \leftarrow (\forall \leftarrow \bar{h}, \exists\leftarrow x) [\bar{h} + x] \mathcal{J} \mathcal{J} \mathcal{J} \lambda = \text{SHT} \]

To illustrate we apply this to the above example:

\[ (d \mathcal{T}d \circ \lambda) [\varepsilon] \mathcal{Z} \mathcal{J} = (d \mathcal{T}d[\varepsilon] \mathcal{J} \mathcal{J}) \lambda \]

Equality, where \( \varepsilon \) is a piece of syntax, should be equal and this can be formally expressed by the following.

For terminating computations (in the TP) the two interpretations are equal.

6. Comparison of Interpreters: The interpretation \( \mathcal{Z} \) interprets the objects using set theoretic considerations. It also evaluates to the interpreter we would expect if we had been evaluating the interpretation is the

\[ 11 \]
Approximation in a pictorial fashion.

Each type in both Z and the Lp. The next slide shows non-terminating executions we introduce a 'bottom' element \( L \).

Formula \( * \) is a particular case for terminating computations. For extending to accommodate non-terminating computations.

Since a comparison is to be made - the Z domain also needs domain - see later. 'The Lp) will be non-terminating - so we need to extend the Lp. However some of the computations (and hence interpretations in expressions) are presented later. The interpretation of schemas and predicates other syntactic expressions of \( Z \) to be interpreted are set - The syntax above is an example of an arithmetic expression.
\( V A R \Leftrightarrow Z \mu \in \mathcal{E} \)

associated with a specification, and \( p \in \mathcal{E} \) is the set of all possible LP environments

**Figure 1: Approximation Diagram** - where \( d \) is a variable
abstract correspond to integers, sets, tuples in the concrete.
whereas for abstract approximation, the abstract interpretation is a set description.

• For abstract interpretation, the abstract interpretation of a piece of syntax:
that they both represent an abstract and concrete interpretation of
abstract approximation and abstract interpretation are similar in
information is output. The comparison is in the Z domain.
This is in order that no incorrect
evaluation of an expression in Z. This is in order that no incorrect
never provide more information than the result obtained by the

• In abstract approximation, that a computation in $\mathcal{A}$ should
The approximation expresses the underlying concept of safeness.
outline summary of the Z syntax to be interpreted.

nam (and enumerated free types) is GIVEN. The following is an
variable names (within a schema) are VAR and the set of given set
Suppose the set of schema names is NAME, and the set of

[BB94]

It is convenient to treat declarations as syntactic objects, as
Definitions denoted expr, pred, goal, schema, axdef respectively;
Expressions, Predicates, Declarations, Schemata and Axiomatic
We consider the following four parts of the Z syntax:

Z Syntax
treated separately-later.

NB Set comprehensions of the form $\{ \text{term} \mid \text{pred} \}$ will be

an enumerated free type

where $x! \in \text{VAR}$

$\{\mathcal{Enum}\text{-Type} \in \text{GIVEN } \bar{x}_i \mid x_i = :: \}$

set union, intersection, distributive union etc.

\[
\cdots | \{ t \cap t_i \} | \{ t \cup t_i \} | \{ \bar{t}_i \} | \text{ a tuple} \\
\{ \bar{t}_i \}^n | \text{ an enumerated set} \{\bar{x}_i \} \\
\text{where } x! \in \text{VAR} \\
\text{where } G! \in \text{GIVEN a Given set reference} \\
\text{an integer expression} | \cdots t_1 | t_1 + t_2 | t_1 - t_2 | t_1 * t_2 \\
\text{the integers and integer values} \\
\text{where } \mathbb{Z} \in \text{the integers and integer values} \\
\mathbb{Z} = :: \}

\text{Numerical and Set Expressions in Z Syntax}
Predicates have the following syntax, where

\[ \text{pred} \]

\[ \text{expr} \]

::

\[ \text{deep} \]

\[ \text{basic-deep} \]

Declarations:

and that schemas can also be referenced as part of a sequence of

Recall that in \( Z \) everything is typed - \( X \), is a basic declaration

Declarations and Predicates
where $d \in \text{decl}, p \in \text{pred}$.

\[
\text{where } \text{Sch}, \text{Sch}_1, \text{Sch}_2 \in \text{NAME} \\
\text{where } \text{Sch} \land \text{Sch}_1 \equiv \text{Sch}_2 \lor \text{Sch}_1 \equiv \text{Sch}_2 \lor \text{Sch}_2 \equiv \text{Sch}_1 \lor \text{Sch}_1 \equiv \text{Sch}_2 \lor \text{Sch}_2 \equiv \text{Sch}_1 \lor \text{Sch}_1 \equiv \text{Sch}_2
\]

where $d \in \text{decl}, p \in \text{pred}, \text{Sch} \in \text{NAME}$

\[
\{\eta \cdot \eta \theta \cdot \eta \text{Sch}\} \mid \eta \theta \cdot \eta \text{Sch} \mid \eta \theta \cdot \eta \text{Sch} \equiv \eta \text{Sch}, \quad \equiv \eta \text{Sch}, \quad \equiv \eta \text{Sch}
\]

and this can be expressed as:

\[
[d \mid p] \equiv \eta \text{Sch}
\]

\[
\text{where Sch, Sch}_1, \text{Sch}_2 \in \text{NAME}
\]

\[
\begin{array}{c}
[d \mid p] \\
\hline
\end{array}
\]

A schema named \text{Sch} is a declaration followed by a predicate:

\[
\text{Schema Syntax}
\]
as constants in \( D \), which means combining them to upper case.

The variable and schema names will subsequently be interpreted as enumerated types.

- In order to animate given sets the user of the animator is
  
  \[
  \{ \text{foo, moo} \} = \text{Booleans} \text{ Bool}\text{ars} \text{ (etc.)}
  \]

- Enumerated free types (enumerated sets):
  
  \[
  \text{(3) Tuples : (2) Integers and sets of integers}
  \]

- The standard domain \( \mathbb{Z} \) consists of:
  
  \[
  \text{(3) \ Domain \ Z}
  \]
possible we shall just use $x$ as denoting a variable value. However - in what follows (where
variable name and value. Notice we use $X$, for which is part of the existing syntax of $Z$.
\[
\begin{align*}
\{ u_x & \leftarrow u_X , \cdots , u_I & \leftarrow I_X \}
\end{align*}
\]

This object is represented more simply by the symbol table

\[
\begin{align*}
\cdot & \prec u_a \equiv u_x , \cdots , u_a \equiv I_x \succ
\end{align*}
\]

The binding

variables named $X$, $u$ with their types.

Consider a schema $\beta$ whose declaration involves $u$

\begin{align*}
\text{Schema Bindings}
\end{align*}
case) constant names in the programming language (and therefore upper as constants in DLP, which means combining them to allowed
The variable and schema names will subsequently be interpreted

• The represented both as terms and as answer sets. (See later)
  • n-tuples are represented by functions of arity n and sets are
    schema bindings:

take place as part of a program execution to determine or check
evaluations of arithmetical, set and expressions other than these
as described in seminar I. Thus
  • The interpretation is of schemas and results in an output and is
    integer values, instantiated values, tuples, bindings and sets;
  • The proposed abstract domain, DLP, includes representations of
    (4) The Logic Programming Domain
this is a schema type -- a list of variable bindings

where each p! is a variable binding binding

a single variable binding

\[ \text{Bind}(\bar{X}, \bar{x}) \text{ where } \bar{X} \in \text{DLP}, \bar{x} \in \text{VAR} \]

an set term

\[ \{ x_1, \ldots, x_n \} \]

tuple

\[ (x_1, \ldots, x_n) \]

where each \( x_i \) is base \( C_k \)

\( D^m \equiv m \text{ an integer} \)

The LP Domain - Output of Expressions
values instantiated.
and that the answers to a query concerning the characteristic

where \( x^K \in \mathcal{D} \), \( x^I \in \mathcal{VAR} \), \( \mathcal{SCH} \in \mathcal{NAME} \)

\[
\left[ \left( \mu x, \mu X \right)^{\mathcal{B} \mathcal{I} \mathcal{N} \mathcal{D}} \cdot (x^1, x^1) \right] = \mathcal{SCH} \theta
\]

binding is denoted in the LP:

Sets of answer substitutions: recall that for some schema \( \mathcal{SCH} \),

where each \( x^i \) is itself a term and \( \emptyset = \mu \mathcal{N} \).

(See [MW85],)

\[
\left( \left( \mu \mathcal{N} \circ \mu x \right) \cdots \right) \circ \left\{ \mu x, \cdots, \mu x, x^1, x^1 \right\}
\]

all \( \mathcal{B} \mathcal{I} \mathcal{N} \mathcal{D} \) terms:

\( \cdot \) Set terms: in the LP (finite) set terms are represented (first of

Set objects in the LP
\[ \text{Faces} = \text{Count} \]
\[ \text{Count} : 0 \]
\[ \text{MaxFaces} \]
\[ \text{Faces} \]
\[ \text{Field} \]

We define the file system in terms of its state variables which are:

\[ \text{MaxFaces} : \mathbb{N}^1 \]
\[ \text{Field} \]

Example: A small file system involves a single given set:
The conceptualization mapping γ is defined next.

Inputs which are associated - all answers terminate.

In this case all states will be generated eventually and the possible

% etc 

? [ bind1(f1es, f1), bind2(count, t) ] 

% second schema binding - a further state

? [ bind1(f1es, {}), bind2(count, 0) ] 

% the initial state

% test of schema f1esys

% Example:
Concretisation Function $\gamma$

\[
\gamma(\biguplus)
\]

\[
\gamma(b_1 \ldots b_n)
\]

where $b_i = \text{Bind}_i(X_i, x_i)$,

\[
\gamma(I_{\{x_1, \ldots, x_n\}})
\]

\[
\gamma(\{x_1, \ldots, x_n\})
\]

\[
\gamma(I_n(x_1), \ldots, x_n)\]

\[
\gamma(I_{\{x_1\}}(x_1), \ldots, x_n)\]

\[
\{\gamma(x_1), \ldots, \gamma(x_n)\},
\]

a single schema binding

\[
\{X_i \mapsto \gamma(x_i), \ldots, X_n \mapsto \gamma(x_n)\},
\]

tuple

\[
\{\gamma(x_1), \ldots, \gamma(x_n)\},
\]

a member of a given set $G$

\[
g, g \in G,
\]

\[
m, m \text{ an integer}
\]

\[
\downarrow \text{ non-termination see later}
\]
\[
\{0 \leftarrow Z, \top \leftarrow \lambda, 0 \leftarrow X\} = \{(0) \leftarrow Z, (\top) \leftarrow \lambda, (0) \leftarrow X\},
\]

The equivalent \( Z \) environment is then
\[
\{0 \leftarrow Z, \top \leftarrow \lambda, 0 \leftarrow X\}.
\]

unknown then the \( \text{LP} \) environment is then
\( \lambda \) with \( Z, X \) both instantiated to 0 and \( \lambda \) if \( \text{VAR} \) is.

\[\text{For example:}\]

\[\text{The program - for example initially, some state of the program -}\]

\[\text{This means that \( \text{TTT} \) functions are total, \( Z \) can similarly be}\]

\[
\{\top, \text{fun} \} = \text{bool}
\]

\[\text{each type. For example the booleans:}\]

\[\text{the \( \text{LP} \) domains are extended by the inclusion of a} \quad ' \top', \text{for}\]

\[\text{In order to accommodate non-terminating executions, both} \quad Z \text{ and}\]

\[\text{Undecidability and Ordering (5)}\]
With regard to sets, formalisation involves set terms. We need to consider the ordering of these terms co-ordinately-wise on tuples. Since the ordering relation works co-ordinately-wise on tuples, we can express this as:

\[(q = a \lor \top = a) \Rightarrow q \sqsubseteq a\]

Then the ordering is \(\sqsubseteq\). For example, if \(a\), \(q\) are integers or members of given sets, there is an imposed ordering in respect of all types of domain elements. Hence ordering.
\((\exists p \equiv \exists q \cdot q \in \exists d : \exists p \in \exists d \cdot \exists p A) \iff \cap \exists d \equiv \cap \exists p \iff \exists d \equiv \cap \exists p\)

Incomplete sets are as follows:

Thus \(\cap\) denotes an incomplete set. The ordering for them, for example \(\{ 1, 2, 3, 4, 5 \} = \{ 5, 4, 3, 2, 1 \} \subseteq \{ 1, 2, 3, 4, 5 \}\) is to 

**Incompleteness in Sets: TP Examples**

For example: \(\{ 1, 2, 3, 4, 5 \} \subseteq \{ 1, 2, 3, 4, 5 \}\), \(\exists d \equiv \{ 1, 2, 3, 4, 5 \}\) and \(\exists d \equiv \{ 1, 2, 3, 4, 5 \}\), \(\exists d \equiv \{ 1, 2, 3, 4, 5 \}\).

The ordering relation can be expressed formally:

**Complete Sets**:

For example: \(\{ 1, 2, 3, 4, 5 \}\).

Sets can also be complete but contain incomplete elements:

(6) Complete and Incomplete Sets
We 'refine' an incomplete set if we complete it or (in addition) add some more elements. For example, 
\{1, 2, 3, 4\} \cup \{1, 2, 3, 4, 5\} \cup \{1, 2, 3, 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 \{1, 2, 3, 4\} and \{1, 2, 3, 4\} \cup \{1, 2, 3, 4\}.

(If could be said that the two sets contain the same 'information' - see [GS90].)
This bottom element is denoted \( \mathbb{N} \) to distinguish it as a set.

In both cases the set evaluates to \( \top \) since functions are strict.

\[
((\cdots \circ u x) \cdots) \circ I x
\]

obtain the set denoted by:

terminate, in an attempt to evaluate an infinite set for example, we

In the second case when a computation of a set term fails to

\[
((\exists N \circ u x) \cdots) \circ I x
\]

case the set contains \( \top \).

The following are two examples of set incompleteness. In the first

Set Terms

Set also answer sets.

and unordered elements. The \( \mathbb{P} \) domain contains sets as terms

The \( \mathbb{Z} \) and \( \mathbb{P} \) domains are both extended to contain incomplete

Incomplete Sets – examples
Programming state, for these may very well be undefined, which fails to terminate, rather than to terms in their initial state. Note that the above applies to terms in an execution message.

The LP implementation of such an output may be a warning.

\[ \text{TLP}_{\text{in}N} \equiv a \]
\[ \text{TLP}_{\text{in}N} = a \cup \text{TLP}_{\text{in}N} = \text{TLP}_{\text{in}N} \cup a \]
\[ \text{TLP}_{\text{in}N} = a \cap \text{TLP}_{\text{in}N} = \text{TLP}_{\text{in}N} \cap a \]

We have, for all set \( a \):
\[
\{ < 3 \subseteq A, 4 \subseteq X > \cdots, < 2 \subseteq A, 1 \subseteq X >, < 1 \subseteq A, 4 \subseteq X >, < 1 \subseteq A, 2 \subseteq X >, < 1 \subseteq A, 1 \subseteq X > \}
\]

This should result in the set of bindings:

\[
\{(\{4, 2, 1\} = \{3, 1, 2, 4, X\}) \land (3 = X) \land (1 = X)\}
\]

\[
\{3, 2, 1\} \subseteq A
\]

\[
\mathbb{N} : A, X
\]

\[
f \in \mathbb{D} \cup \mathbb{N}
\]

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

Whether the answer set contains some or indeterminate answers

\[
\text{\% delayed on \( \land \) and \( \lor \)}
\]

\[
\{ \neg \land \} \text{ unsolved goals are:}
\]

\[
\text{\%}
\]

\[
\text{\%}
\]

When animated this results in:

\[
\text{\%}
\]
message.

where \( q_1, q_2 \) are the two scheme bindings output before the error

\[ \cup \{q_1, q_2\} \]

denoted

This set is an example of an incomplete set (as above) and is the rest of the set.

Thus there is no way of knowing from the output, the nature of
not be considered here.

We need a different approximation rule for incomplete sets, but that will

provide a basis of a structural induction rule.

The strategy for proof involves structural induction and the next

\[(d_r d_0 \circ \eta) [\varepsilon] \subseteq (d_0 d_r [\varepsilon] d_r \varepsilon) \cdot (d_r 0) \eta] \subseteq (d_r d_0 [\varepsilon] d_r \varepsilon) \cdot \eta\]

Approximation Rule I

**(*)

Proof Method - Structural Induction

**Proof** I represents the fact that it is a syntactic Z expression

\[Z \subseteq \eta\]
Proof (for reader)

\[ ((d \Pi d[x] dJ_3 \land) Zf) \subseteq ((d \Pi d[x] dJ_3) d \Pi f) \land \]

This is the method of approximating mechanisms.

The third condition is the key one, which

\[ Zd[x] Zf = (Zd[x] Zf) Zf \]

However, this condition is only true for complete sets and is not

A further condition is a property of \( Z \), i.e., the

\[ d \Pi d[x] dJ_3 = (d \Pi d[x] dJ_3) d \Pi f \]

In order to prove correctness it is necessary to show

Condition 1

Condition 2

Condition 3
Thus if $f$ is set union of the interpretations of the syntactic expressions of the domain and $D$ denotes by $dD$, the tuple $e = (x_1, x_2, \ldots, x_n)$. For example, $f$ might be the syntactic operator on variables $x$. Then it also holds for syntactic expression $e$.

This means that if it can be shown that **

5. Predicates: quantified expressions (which depend on
declarations).

4. Set comprehension and variable declarations:

3. Predicative expressions: imix:

2. Set expressions:

1. Numbers and numeric expressions:

Induction takes place in the following order:

Only the novel or most salient parts are presented here. The
Induction is over each Z construct and is shown in full in the thesis.
Thus, \( \forall z, m \in \mathbb{Z}, z d[w] z_2 \subseteq (d \pi d[w] d^2 z) \)

\( \top = (\top) \) since IEEE floating point standard. Thus the latter is suggested by the alternative to the character \( \infty \). The latter may be implemented by the output of an error message, or

\[
\text{Max} < m - \text{Min} \text{int} \Rightarrow m = d \pi d[w] d^2 z
\]

\( \text{Max} \geq m \geq \text{Min} \text{int} \) to terminate. Thus, for \( m \in \mathbb{Z} \),

Available. Any attempt to exceed them will cause the computation

Example (i) Integers

(8) Proofs – Induction
\[ \mathcal{G} = \{(x)_{\mathcal{D}\mathcal{I}} : x \} = (d_{\mathcal{D}}[\mathcal{G}]d_{\mathcal{J}})_{\mathcal{I}} \]

\[ b = (d_{\mathcal{D}}[b]d_{\mathcal{J}})_{\mathcal{I}} \]

Predicate \( \mathcal{I} \). In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:

In each case the abstract interpretation is exact for:
deomonstrate the proof.

The case is similar for set expressions - we use \( \text{set union} \) to evaluate to \( \top \) and always underestimate the \( \mathbb{Z} \) interpretation.

In the case where the integers exceed these bounds, the LP approximation

\[ \text{MinInt, MaxInt of the LP implementation, results in an exact} \]

For integer expressions such as addition, subtraction, two cases computation results in \( \top \) and underestimate the \( \mathbb{Z} \) interpretation;

For any computation, if the memory bounds are exceeded, the

Integers and Set Expressions
which will hold for set operations for terminating computations.

\[ \tau \in \{ d \} \cap \{ x \} \in \{ d \} \cap \{ x \} \Rightarrow (d \cap \{ x \}) \cap \{ d \} \cap \{ x \} \]

**Condition 1 becomes:**

impliments \( \cup \) for finite sets in the same manner as for \( \cap \).

We assume that \( d \cap \{ x \} \) is set-theoretic and \( \{ x / d \cap \{ x \} \} \) is a substitution that \( \cap \) \( x \) \( d \cap \{ x \} \) \( \{ x / d \cap \{ x \} \} \) so that \( \cap \) \( x \) \( d \cap \{ x \} \) \( \{ x / d \cap \{ x \} \} \)

The expression \( x \) \( d \cap \{ x \} \cap \{ x \} \cap \{ x \} \) is evaluated using the \( \cap \) ground

\[ (d \cap \{ x \} \cup \{ x \}) \]

Example (III) Set Union: \( x \cap \{ x \} \cap \{ x \} \cap \{ x \} \)
computations.

In other words, the computation is exact for terminating

\[
\cdot ((dT \ll ([z, x]') d\mathcal{J} Z) \land) Zf = (\neg \nu) \land Z \cap (\neg \nu) \land = (\neg \nu) dT \cap (\neg \nu) \land =
\]

\[
((dT \ll ([z, x]') d\mathcal{J} Z) dTf) \land
\]

Condition \( 3 \) becomes:

Since is set-theoretic then \( dT \cap \)

\[
\cdot Zd([z \cap x]') d\mathcal{J} Z = (\neg \nu) \land Z \cap (\neg \nu) \land = (Zd([z, x]') d\mathcal{J} Z) Zf
\]

and \( ((\neg \nu) \land (\neg \nu) \land) \) way to complete sets, \( (z, x) \) in the expected \( Z \mathcal{J} \) if \( x, x \) are complete sets, \( \land \) in the expected.

Condition 2.
\[ S, \quad \forall \]

Since we have established conditions for complete sets and will always exceed \( \Gamma \). The result is similar for incomplete sets. (Assuming \( p \neq c \)).

\[ \text{RHS is } \exists \quad \forall \]

\[ \text{LHS is } \exists \quad \forall \]

\[ \forall x \in \mathbb{N} \]

**directly for incomplete or infinite sets, then holds when **

\[ \mathbb{N} \]

\[ \forall x \in \mathbb{N} \]

For example, suppose: For non-termination computations we interpret figure 1 directly.
\[
\left( \text{true} = \mathcal{P}[\mathcal{P}][\mathcal{P}] \right) \land \left( \text{false} = \mathcal{P}[\mathcal{P}][\mathcal{P}] \right)
\]
\[
\Leftrightarrow \left( \text{true} = \mathcal{P}[\mathcal{P}][\mathcal{P}] \right) \land \left( \text{true} = \mathcal{P}[\mathcal{P}][\mathcal{P}] \right) = \mathcal{P}[\mathcal{P}][\mathcal{P}][\mathcal{P}]
\]

We also have:

\[
\top = (\top) \land \left( \text{false} \right) \land \left( \text{true} \right) = (\top) \land
\]

then

\[
\{ \top, \text{true}, \text{false} \} = \mathcal{P}[\mathcal{P}][\mathcal{P}][\mathcal{P}], \{ \top, \text{false} \}, \{ \text{true} \}, \text{false} \} = \mathcal{P}[\mathcal{P}][\mathcal{P}][\mathcal{P}][\mathcal{P}]
\]

A predicate evaluates to \( \top \) when a program has no failures or fails to terminate during its evaluation. Thus if a predicate evaluates to \( \top \) when a program has no failures or fails to terminate during its evaluation, \( \mathcal{P} \) and \( \mathcal{P} \) denote by \( \mathcal{P} \) the interpretation of syntactic predicates in \( \mathcal{P} \) domain.
(See next slide.)

- It is presented in the form of three constraint satisfaction rules.

- It is evaluated.

- The update is extended to all literals connected to the literal being

As a result of the resolution inference rule of logic programming,

of different ways, thus providing a set of answer substitutions.

- It can happen that the environment can be updated in a number

environment, from \( \mathcal{D} \), to \( \mathcal{D} \), (say)

- Predicates can both provide a boolean answer and update the

- These are equality, subset, membership:

Prefix Predicates
\[ \{ \forall a \leftrightarrow \exists x, \forall a \leftrightarrow \exists x \} \oplus dT \theta = dT \theta \]

\[ dT \theta [d] d\mathcal{J} d = dT \theta [d \lor (\exists x \mathcal{I} x)] d\mathcal{J} d = dT \theta [(\exists x \mathcal{I} x) \lor d] d\mathcal{J} d \]

**Constraint Property 2:**

are also enhanced:

The environments of predicates composed to the index predicates

\[ \{ \forall a \leftrightarrow \exists x, \forall a \leftarrow \exists x \} \oplus dT \theta = dT \theta \]

\[ \text{en}_I = dT \theta [\exists x \mathcal{I} x] d\mathcal{J} d = dT \theta [\exists x \mathcal{I} x] d\mathcal{J} d \]

**Constraint Property 1:**

called this property:

partially defined they can become ground through resolution. We

then if either (or both) \( x^1 \) or \( x^2 \) is undefined or only

Suppose \( \mathcal{I} \) is an index predicate, standing for equality, subset or
\[
\begin{align*}
&\text{where } E \in \mathcal{E} 
&\forall a \leftrightarrow \exists x \oplus dT \theta = dT \theta \\
&\exists a \leftrightarrow \exists x \oplus dT \theta = dT \theta \\
&\forall a \leftrightarrow \exists x \oplus dT \theta = dT \theta \\
&\exists a \leftrightarrow \exists x \oplus dT \theta = dT \theta
\end{align*}
\]

where

\[
\begin{align*}
&T \theta \{d\} T \theta \{d \vee (\exists x \bar{T} x)\} T \theta \{d \vee (\exists x \bar{T} x) \vee d\} T \theta \{d \vee (\exists x \bar{T} x) \vee d\} T \theta \{d \vee (\exists x \bar{T} x) \vee d\} T \theta \{d \vee (\exists x \bar{T} x) \vee d\}
\end{align*}
\]

**Constraint Property 3**

Consider substitutions. Examples are subset and membership. We call this:

The different values contribute to different answers. An extension of these properties is the case where \( x \) can take many values.

\[
\begin{align*}
&\forall a \leftrightarrow \exists x \\
&\exists a \leftrightarrow \forall y \\
&\exists z \leftrightarrow x
\end{align*}
\]

\( \forall + x = z \implies (a \exists, z, x) [1, x] = [\forall, z, x] [1, x] \implies \text{example which illustrates both properties} \)
The same constraint properties can be extended to the Z environment.

\[ \top \cdot \downarrow = \downarrow \quad \top \cdot \downarrow = \downarrow \quad \top \cdot \downarrow = \downarrow \quad \top \cdot \downarrow = \downarrow \]
Z interpretation as required.

\exists \text{ Interpretation of } \in \text{ underestimates the}

Membership * is true where \( f \)'s syntactic predicate \( \in \) for

\( x_1 \subseteq x_2 \); * holds for \( (x_1, x_2) \):

\( (x_1, x_2) \) is the syntactic predicate \( \subseteq \) for variable \( x_1 \) for variable \( x_2 \) for variable \( x_1 \) for variable \( x_2 \)

Equality: * is true. 

\( x_1, x_2 \) are defined prior to execution of equality function and in each

\bullet There are three cases for \( x_1, x_2 \); depending on whether or not

values, we can summarise: thus.

\bullet Assuming that the execution terminates, and \( x_1, x_2 \) take unique

\( x_2 \subseteq x_1 \).
constrained by \( p \) and used to evaluate \( t \).

Each value is generated (or tested in the case of schemas). The declaration results in a single tuple of values \((x_1', \ldots, x_n')\) being expressed.

Other examples include set comprehensions, and quantified:

\[
\forall \, x_1 : T_1; x_2 : T_2; \ldots; x_n : T_n . \quad p
\]

\( p \) is of the form where \( p \) is a declaration, \( d \) is a predicate and \( t \) a term.

\[
\forall \, \text{Variable Declarations} \quad \text{(for example) schemas:}
\]

\[\text{Variable Declarations}\]
environment as in the case of the membership predicate. If this happens, the effect of either testing a value or updating the

\[ \mathcal{D}(p \in L \land \mathcal{D}(x : L) = \mathcal{D}(p \in L \land \mathcal{D}(x)) \]

is a set. (treated separately.)

The declarations are treated as predicates. The declarations are treated as predicates. (treated separately.)

The environment variable values generated by the declarations will update the environment. The environment is built recursively and interpreted in a similar manner to the inlin predicates defined previously.

The evaluation function is built recursively and interprets in a similar manner to the inlin predicates defined previously.

By \( \mathcal{D} \).

\[ \mathcal{D}(x : L) \text{ where } x \text{ is a variable and } L \text{ is a set-valued with value provided.} \]

\[ \mathcal{D} \text{ gives the interpretation in } \mathcal{D}(x \in L) \text{ of syntactic declarations.} \]
distributed union.

The proof of correctness for declarations is based on an equivalence

tuple: (in the TP. In our Godo library this is OndPa1).

TL2 captures a representation of ordered pair (as an example of a

\[
\begin{align*}
\text{\( p \in \mathcal{D}[\mathcal{T}_2 \in \mathcal{T}_1 \in \mathcal{T}_1 \times \mathcal{T}_2] \quad \mathcal{D} \subset \mathcal{D} \) & \quad \mathcal{D} \in \mathcal{D} \\
\text{\( p \) is a Cartesian product, \( \mathcal{T}_1 \times \mathcal{T}_2 \)} & \quad \text{environment as the subset predicate.}
\end{align*}
\]

set test for reasons of efficiency. It has the same effect on the

\[
\begin{align*}
\text{\( p \in \mathcal{D}[\mathcal{T}_1 \in \mathcal{T}_1 \times \mathcal{T}_2] \quad \mathcal{D} \subset \mathcal{D} \) & \quad \mathcal{D} \in \mathcal{D} \\
\text{\( p \) is a subset test rather than a membership of power}
\end{align*}
\]

\[
\begin{align*}
\text{\( p \in \mathcal{D}[\mathcal{T}_1 \in \mathcal{T}_1 \times \mathcal{T}_2] \quad \mathcal{D} \subset \mathcal{D} \) & \quad \mathcal{D} \in \mathcal{D} \\
\text{\( p \) is a power set, } \mathcal{T} = \mathcal{T}[\mathcal{T}]
\end{align*}
\]
\{d \mid d \in J \land \neg \forall \phi \in \Phi : \phi \not\in \Phi\} \cap \{d \mid d \in J \land \neg \exists \phi \in \Phi : \phi \not\in \Phi\} = \{d \mid d \in J \land \neg \exists \phi \in \Phi : \phi \not\in \Phi\}

Comprehension of \(s\) is interpreted in the Lp:

Since deklarations in the Lp are treated as predicates, then the set $z \mathcal{D}, p \mathcal{J} \mathcal{D}$ where the deklarations act as generators for $s$. Each of these interpretations is respectively dependent on its $z \mathcal{D}$ as $z \mathcal{D}$:

Interpreted:

Each $x^? : \bot$ provides a value which contributes to the tuple:

$\{d \mid d \not\in \bot : u \not\in \bot \land x^? : \bot, x^? : \bot \}$

Set comprehension
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 $\mu\rho$. The environment $\rho\rho$ inside the comprehension is the variable which acts as a set generator, for recall that

$$Dcp[x : \tau]\rho\rho = Pc\mu[x \in \tau]\rho\rho.$$ 

A similar interpretation is true for $D\mu$. 
each binding is denoted respectively by \( \text{Bind}_{\mathcal{I}^m} X \) and \( \text{Bind}_{\mathcal{I}^m} X \), where \( \mathcal{I}^m \) is a set of variable bindings. Where

Recall that \( X \) is a variable name and that the output

\[ \uparrow \mathcal{I}^m \cap \mathcal{C} \cap \mathcal{P} \triangleq \mathcal{C} \cap \mathcal{P} \]

\[ \text{Scheme} \]

\[ \text{Suppose that the syntactic object schema are interpreted in the} \]

\[ \text{Interpretation of Schemes} \]
of them (as in the case of the Unix file system).

the schema variables (as in the case of the assembler or just some 
imposed environment) where ϕ can contain defined values of all

We assume that the set of bindings is constrained by an initial

\[ \{ \{ u_x \leftarrow u_X, \ldots, x \leftarrow X \} \mid \Delta : u_X, \ldots, \bot : X \} \]

suggested in \cite{BB94} by a set expression:

A set of schema bindings of \texttt{sch} can be represented in \texttt{Z} as

\texttt{sch evaluates to a set expression of bindings of variable name(s)}

and by the schema predicate.

The bindings are constrained by the variable declarations to values.
Providing a single binding for a schema expression, and schema expressions is in terms of a characteristic predicate, the interpretation of schemas

\[ \{u \leftarrow uX, \ldots, x \leftarrow xX \} \]

replaces

\[ \left( uX, uB \right) \text{ in } B \]

for

\[ d\mathcal{J}_\theta^p \left[ \left( uX, uB \right) \text{ in } B \right] \cdot \]

\[ d\mathcal{J}_\theta^p \left[ d\mathcal{C} \mid u \downarrow : uX, \ldots, x \downarrow : xX \right] \mathcal{J}_S = \]

\[ \mathcal{J}_\theta^p \left[ \left[ d\mathcal{C} \mid u \downarrow : uX, \ldots, x \downarrow : xX \right] \mathcal{S} \right] \]

The interpretation in the LP is

\[ \mathcal{J}_\theta^p \left[ \right] \cdot \]

\[ d\mathcal{J}_\theta^p \left[ d\mathcal{C} \mid u \downarrow : uX, \ldots, x \downarrow : xX \right] \mathcal{J}_S = \]

\[ \mathcal{J}_\theta^p \left[ \left[ d\mathcal{C} \mid u \downarrow : uX, \ldots, x \downarrow : xX \right] \mathcal{S} \right] \]

The interpretation of a set expression:

\[ d\mathcal{C} \mid \mathcal{A} \] is

\[ \mathcal{J}_\theta^p \left[ \right] \mathcal{S} \text{ of the schema } \mathcal{S} \]
\[ u_x \cdots \cdot u_x \text{ are replaced by } u_X \cdots \cdot u_X \text{ where all the free occurrences of } \]
\[ \mathcal{G} \text{ is defined as } \mathcal{G}p \]
\[ \gamma \left( \left[ \left( u_x \cdot u_X \right)^{\text{bind}} \right]_X \right) = \text{binding} \]
\[ \iff \left( \text{schema type(binding, } \mathcal{S} \text{) schema} \right) \]

The characteristic schema predicate of \( \mathcal{S} \) is as follows:

\[ \mathcal{G}p \left[ u_x \cdots \cdot u_x \right]_{\mathcal{T}} d \mathcal{J} \]

satisfies

\[ \mathcal{G}p \left[ u_x \cdot u_X \right]_{\mathcal{T}} d \mathcal{J} \]

where each enhanced environment \( \mathcal{G}p \in \mathcal{T}_d \) evaluates in the LP to bindings of variable names to values, where

\[ \mathcal{G}p \left[ u_x \cdot u_X \right]_{\mathcal{T}} d \mathcal{J} \]

Characteristic Predicate for a Schema Expression
and the one chosen is \( \text{Pileup} \).

It is worth investigating how the above would apply to a schema,

\[
\forall \phi \in \text{Pileup}, \ Zd[\phi \land c] \vdash Zd[\phi], \ Zd[\phi], \ Zd[\psi] \vdash \text{Pileup}
\]

The values satisfy \( Zd \) can \( \equiv \) \( \text{ran} (\exists x) \exists \}

\[
\cdot \{(u_\text{X})_\exists \iff u_\text{X}, \cdots, (u_\text{1})_\exists \iff \text{1}_\exists \} = \text{binding}
\]

bindings where

The \( Z \) interpretation can similarly be represented by a set of

\[
(\leftrightarrow) \iff \text{VWA, this has the same effect as } \iff \text{an only if }, \equiv \text{the}
\]

although the schema definition in the \( \text{L} \) uses \( \equiv \text{the}

\text{Constraint Properties 1 - 3 defined previously. Note that}

The generated values have been obtained via the application of

was part of the initial environment.

\[ \forall x: \text{which satisfy SchemaType has either been generated or} \]
If we substitute these values, a binding for \( \text{Fields} \) is given by:

\[
\{ \text{Fields} \} 
\]

Then, and that \( \text{Fields} \) is instantiated as \( \{ F1, F2, F3 \} \). Suppose that \( \text{MaxFields} = 10 \) is a value provided by the animation.

This can be written horizontally as:

(9) Interpretation of \( \text{Fields} \)
\{0 \ldots 10\} \ni \text{count} \land \text{files} \neq \# \\
\text{since schema predicate and its declaration, count, evaluates to 'I', then in order to satisfy the thus \text{files} evaluates to } \{F_1\} \text{ (say), then in order to satisfy the}
\begin{align*}
\text{\begin{align*}
\& \land \text{count} \ni \text{files} \neq \# dP \\
\& \land \text{files} \in \{F_1, F_2, F_3\} \subseteq P \text{, files] } dP \\
\& \text{binding } [B\text{ind}^1(\text{files}, \text{files}), B\text{ind}^2(\text{count}, \text{count})] = \text{bind}\end{align*}}
\end{align*}
A binding of \text{files} can be expressed:

Similarly for \text{count}.

\begin{align*}
\{F_1, F_2, F_3\} \subseteq \text{files} \\
\text{through the interpretation of its declaration:}
\end{align*}

During the execution, \text{files} is of type \mathcal{I} so becomes evaluated:

\begin{align*}
\{\top \leftarrow \text{count}, \top \leftarrow \text{files}\} = P dP \\
\text{Initially, 'files}
as was indicated previously, bindings can be obtained from the full set of answer substitutions, which was one of the values actually obtained. The full set of

\[ ([\text{Bind}(\text{Count}, I), \{I, \varphi\}, \text{Bind}(\varphi, I)] = \text{binding} \]

the environment and

\[ \text{where} \]

\[ d^0 \{I, \varphi\} = I \]

Substituting these values yields:
and some are determined by the execution

\( d^T_0 \) ground in each of the schemas considered, some variables are ground in

For the Unix file system, and for the two-phase assembler, for

\( Z_0 d = zd \), where similarly \( d^T_0 d = d^Td \) as that contains no undefined values and the

for the complete assembler, the variables were initially all
\[ (d^I d)[\underbrace{(u_x, u_X)^{\text{upr}^1 B \ldots \text{upr}^{11} B}}_{\text{upr}^1 B} + \cdots + \underbrace{u_x, u_X}_{\text{upr}^1 B}] \]

- \( d \mathcal{C} \) \bigg| \underbrace{u_v: u_x \cdot \cdots \cdot \text{upr}^{11} X}_{\text{upr}^1 X} \bigg\} \mathcal{J}(\mathcal{Z}) \rangle = (d^I d) \bigg[ (d \mathcal{C} \bigg| \underbrace{u_v: u_X, \cdot \cdots \cdot \text{upr}^{11} X}_{\text{upr}^1 X} \bigg\} \mathcal{J}(\mathcal{S}) \rangle \]

Thus the left hand side respectively of the syntactic expression \( f \) is ** of **.

The interpretation in the domain and LP domain of \( f \) if \( f \) is a declarative and \( \mathcal{C} \) is a predicate. We denote by \( (d \mathcal{C} \bigg| \underbrace{D}_{\text{upr}^1 D} \bigg) = \underbrace{\text{de}}_{\text{upr}^1 D} \) a syntactic operator which forms a schema from tuple \( e \) where is is ** can now be considered for schemas: **

Aproximation for Schemas
\[ \text{For incomplete answer sets the LP underestimates as in the case of} \]

\[ \text{as in the case of } \mathcal{P}(\mathcal{S}(S)) \text{.} \]

These interpret exactly where components are finite and complete.

This is a set comprehension which has been treated previously.

\[
\begin{aligned}
\mathcal{Z} \mathcal{d} \left[ \left\{ u_x \leftarrow u_X ; \cdots ; i_x \leftarrow i_X \right\} \cdot d \mathcal{C} \mid u \perp : u_x ; \cdots ; i \perp : i_x \right\} \right] \mathcal{Z} = \\
\mathcal{Z} \mathcal{d} \left[ \left[ d \mathcal{C} \mid u \perp : u_X ; \cdots ; i \perp : i_X \right] \right] \mathcal{Z} \\
\end{aligned}
\]

The right hand side of
contributing to an effective tool have thus been demonstrated.

- The potential of the rules and the arithmetic language, G \( \text{\textcopyright} \),
correct.

- Correctness criteria have been applied and the rules shown to be provided with a formal basis.

First talk and shown to be practical. In this talk the rules have been A set of translation rules from Z to G \( \text{\textcopyright} \) was presented in the

Conclusions
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
   where possible) the automatic generation of test cases;

3. Automation of the rules;

4. A strategy for selecting test cases for animation, including
   An interesting area of work would be the investigation of a
   functional logic language for animation purposes as suggested

In [WVK98],
References

[99]


[00]

USA, 1985.

[85]

1987.

[87]

1984.

[84]

1990.

[90]