Analysing Modes and Subtypes in Z Specifications

Michael Winikoff

Technical Report 98/2

Department of Computer Science
The University of Melbourne
Parkville, Victoria 3052
Australia

Abstract

Poor requirements are the cause of a large proportion of defects in released software. Requirements can be improved by using mathematical modelling techniques. These have a number of advantages and a single major disadvantage–it is difficult to demonstrate a formal specification to a client or user. One proposed solution to this problem is to automatically derive (executable) prototypes of a system from a formal specification by using analysis. This process is known as *animation*. In this report we focus on the analyses used in an animation tool for the specification language Z. Specifically, we look at *mode* and *subtype* analyses. More extensive motivation and discussion of other aspects of the tool can be found elsewhere.

Contents

1 Introduction

A majority of defects in software can be traced back to poor requirements [2, 5, 8]. Two approaches to improving requirements are *prototyping* and *mathematical modelling*. Mathematical models have the advantage of being more abstract. This enables them to avoid design biases and makes them more suitable as a basis for subsequent system development. Additionally, it is easier to analyse a mathematical model than to analyse a prototype and it is also easier to rigorously prove that certain system properties hold.

Mathematical models suffer from one significant drawback compared with prototypes—a prototype can easily be demonstrated to a client or end user in order to obtain feedback. *Animation* has been proposed as a way of obtaining this advantage for mathematical models. Animation involves analysis of a specification in order to determine information which can be used to automatically derive a prototype.

Designing an animation system involves a number of basic questions:

(a) What additional information needs to be determined in order to make a specification executable?

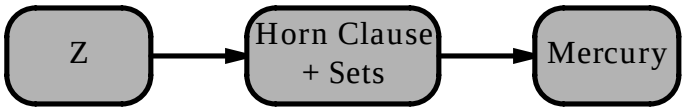
(b) How can this information be determined?

(c) How can the information be (automatically) combined with the specification to yield a prototype?

Another important question is how can the prototype be used to explore the specification.

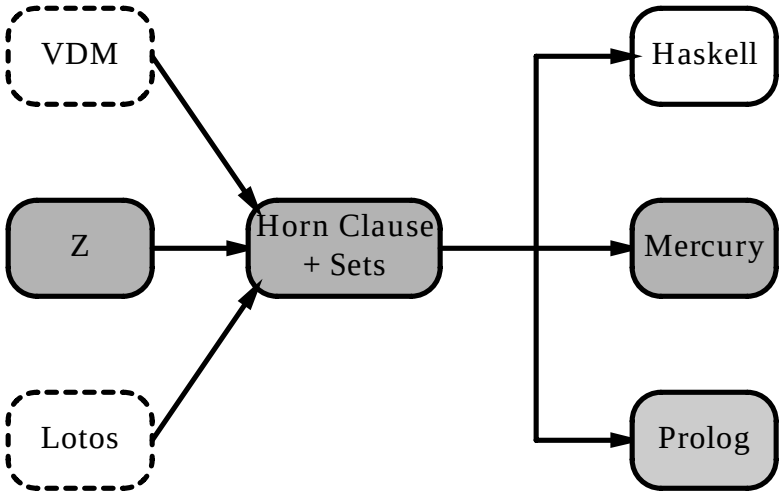
In this report we look at the animation tool MUZAK and at the analyses used. We present the design of MUZAK and describe briefly what information we need in order to translate specifications to executable code and how the translation is performed. We then focus on how the information is determined. The complete details for two analyses are presented.

MUZAK animates Z [1, 12] using Mercury [9–11]. The Z specification to be animated is translated to first order set theory (in Horn clause form) and then analysed. The results of the analysis are used to generate Mercury code.



The advantages of dividing a compiler into a source-dependent front end, a target-dependent back end, and an intermediate processing stage which is independent of the source and target are well known. This structure makes it easier to extend MUZAK to handle new source languages and new targets.

The figure below indicates how the system could be extended to handle other targets and other specification languages. Although first order set theory is not a perfect match with every specification notation, it is flexible enough to allow a reasonable representation of many input notations.



In the next section we discuss the design choices that need to be made and present arguments for our choices. We then present in section 3 the mode analysis and in section 4 the subtype analysis. Section 5 gives a complete example.

Motivation, results, and discussions of other aspects of MUZAK can be found in [13, 14] For up to date information on MUZAK and the pipedream project see:

`http://www.cs.mu.oz.au/~winikoff/pipe`

2 Overview

Input to the analyses

Before proceeding to discuss design issues we briefly describe the input to the analyses. The translation from Z to first order set theory is given in [14]. Note that the translation produces a set of Horn clauses.

For the purposes of the analysis we can consider certain constructs to be expressible in terms of other constructs:

$$\begin{aligned} F \Rightarrow G &\Leftrightarrow (\neg F) \vee G \\ \forall x : T \bullet p^x &\Leftrightarrow \neg\ \exists x : T \bullet \neg\ p^x \\ \exists_1 x : T \bullet p^x &\Leftrightarrow \#\{x : T \bullet p^x\} = 1 \\ R = (\mu x : T \bullet p^x) &\Leftrightarrow \exists y \bullet y = \{x : T \bullet p^x\} \wedge \#y = 1 \wedge R \in y \end{aligned}$$

Additionally λ can be expressed in terms of a set comprehension.[1]

[1] Note that an additional annotation is needed to indicate that the result is guaranteed to be a function. Likewise, a μ is guaranteed to generate a single value.

Most of the library predicates and the primitives $=$ and $\in$ can simply be viewed as predicates with a given mode/subtype declaration.

This leaves the following core to be analysed:

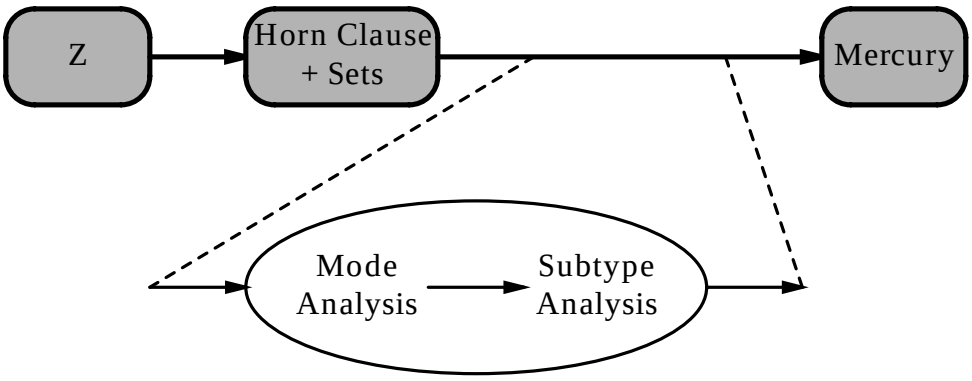
$$
\begin{array}{lll}
\mathcal{L} & ::= & \mathit{atom} \mid X \in X \mid X = X \\
 & & \mid \mathcal{L} \wedge \mathcal{L} \mid \mathcal{L} \vee \mathcal{L} \mid \neg\, \mathcal{L} \\
 & & \mid \mathcal{L} \Leftrightarrow \mathcal{L} \mid \exists D\ \mathcal{L} \\
 & & \mid \mathit{if}\ \mathcal{L}\ \mathit{then}\ \mathcal{L}\ \mathit{else} \mathcal{L} \mid X = \{D \mid \mathcal{L}\}
\end{array}
$$

Note that although we retain $\Leftrightarrow$ and if-then-else[2], the relevant analysis rules can be derived as if they were defined using:

$$
\begin{array}{rcl}
F \Leftrightarrow G & \Leftrightarrow & (F \Rightarrow G) \wedge (G \Rightarrow F) \\
 & \Leftrightarrow & (F \wedge G) \vee ((\neg F) \wedge (\neg G)) \\
\mathit{if}\ F\ \mathit{then}\ G\ \mathit{else}\ H & \Leftrightarrow & (F \wedge G) \vee (\neg F \wedge H)
\end{array}
$$

Design Questions

The translation from Logic to Mercury involves two analyses as depicted in the following diagram. In this section we briefly explain why we have chosen to perform analysis at all, and why these particular two analyses. Other questions such as "Why use Z?" and "Why use Mercury?" are addressed elsewhere [13].



[2]Since expanding them out would increase the size of the formula

Why analyse?

There is a gap between a set of logical formulae (in Horn clause form) and an executable logic program. The goal of the analyses (in particular, the mode analysis) is to derive information about the specification which can be used to close this gap.

It is possible to simply attempt to execute the specification and determine information at runtime. There are a number of reasons why this is less desirable. Firstly, determining information can be done better using global analysis than local execution. Secondly, we assume that the animation tool will be used as part of an iterative process. Thus, the tool will be given non-animatable programs. In this case, it is preferable to detect this as early as possible. In some cases, early detection can prevent a non-terminating computation from occuring. In other cases, compile-time analysis can give more accurate feedback as to *why* the specification could not be animated. Additionally, by detecting problems with executing the specification earlier, a larger range of options for dealing with the problem are available. For example, if a non-executable part of a specification is detected only at run-time then it is difficult to attempt a different execution strategy. On the other hand, if the problem is detected at compile time then the user could be consulted and a different version of the prototype generated. Finally, the mode analysis can sometimes detect errors in the specification–in some cases it behaves like a form of type checking.

Which analyses?

The two types of information we have chosen to focus on are *mode information* and *subtype information*. Mode information is essential since Mercury is strongly moded. It is also important since it guarantees that a sequential execution strategy will not flounder and it avoids violations of soundness due to, for example, negation as failure binding variables. Subtype information is useful since Z's type checking reduces everything to base types—a function declared to operate over sequences is only guaranteed to be passed a set of pairs of numbers and objects. By checking subtypes at compile time we can reduce the overhead of runtime checks.

Determinism analysis is omitted[3] (at this stage) for two reasons. Firstly, it is possible to declare everything as `nondet`. This amounts to determining determinacy at runtime. Secondly, Mercury can be fussy about what it will and won't accept as being deterministic. In order to be able to produce a `det` declaration and be confident that Mercury will accept it, we need to essentially copy Mercury's determinism analysis algorithm.

Why aren't you using a constraint-based execution model?

Constraint programming languages are quite poor at reporting causes for failure. Since in animation we deal with specifications which may not be entirely executable

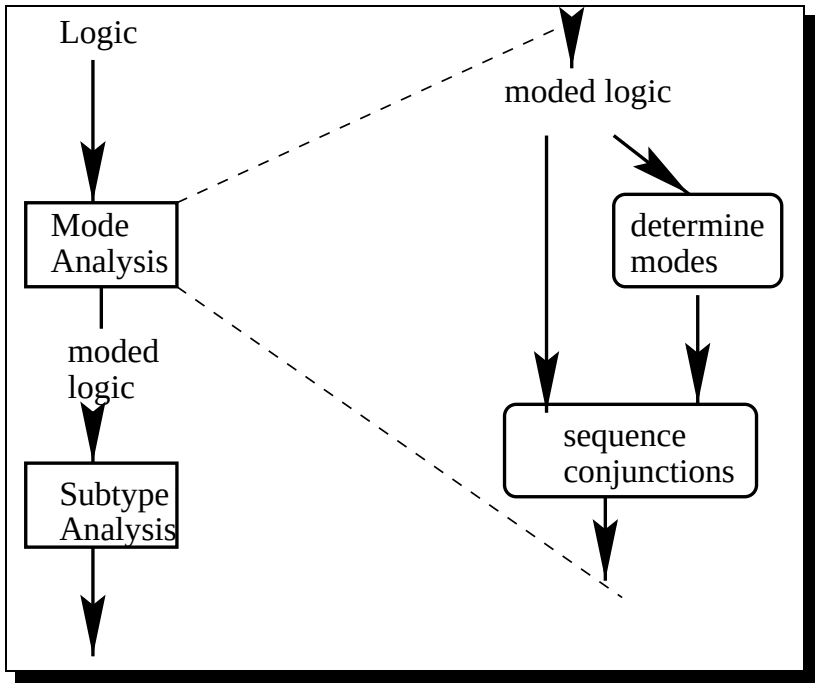
[3]The subtype analysis actually includes a form of determinism analysis which is needed in order to determine the size of set comprehensions. This could be used to derive determinism if the Mercury compiler is capable of also deriving the same information. If not, we could have a situation where a generated program doesn't compile.

it is important that if an execution attempt fails the user can be given a reason. In particular, the user needs to know whether the failure was due to a bug in the specification or due to a limitation of the animation tool. Note also that constraint-based logic programming languages tend to use unification without an occur check and hence are unsound.

Note that our execution model in conjunction with a pre-execution mode analysis (and conjunction reordering) can handle a range of modes and is thus more flexible than Prolog's execution model.

3 Mode Analysis

The idea behind mode analysis is that we have a formula F which may or may not be executable. We analyse it (using conventional data-flow analysis techniques) to determine how it could potentially be executed. Finally, we modify the formula using the derived information to obtain an executable formula. This process is illustrated in the following diagram:



A key point is that the design of a mode analysis (and of the modification of F) is done with respect to a given execution model. In this work we assume a left to right sequential execution model. The execution of a logical formula can be presented as a set of inference rules. The notation $\psi\{F\}\phi$ indicates that if F is executed with the initial binding ψ then the binding ϕ results. We use $\psi + x$ to add a fresh (free) variable to the environment and we use $\psi \Leftrightarrow x$ to project out a variable. For example, $\{y \mapsto 3\} + x = \{y \mapsto 3, x \mapsto \mathit{free}\}$ and $\{y \mapsto x, z \mapsto x\} \Leftrightarrow x$ is either $\{y \mapsto z\}$ or $\{z \mapsto y\}$.

$$\frac{\psi\{F_i\}\phi}{\psi\{F_1 \vee F_2\}\phi} \qquad \frac{\psi\{F_1\}\theta \quad \theta\{F_2\}\phi}{\psi\{F_1 \wedge F_2\}\phi}$$

$$\frac{\psi + x\{F\}\phi}{\psi\{\exists x \bullet F\}\phi \Leftrightarrow x} \qquad \frac{\neg\ \exists \phi \bullet \psi\{F\}\phi}{\psi\{\neg F\}\psi}$$

$$\frac{\psi\{\exists \overline{x} \bullet A = A' \wedge G\}\phi}{\psi\{A\}\phi}\ A' \leftarrow G \in \mathcal{P}$$

$$\overline{\psi\{A = A'\}\mathit{unify}(A, A', \psi)}$$

A mode [7] is an abstraction of the operational semantics of a formula. A mode $I \Rightarrow O$ describes the effect of executing a formula on (abstracted) substitutions.

In particular we use a simple abstraction which records the *ground* variables. Thus a mode $F :: I \Rightarrow O$ indicates that if the variables in I are ground then after executing F the variables in O will be ground. For example, equality ($=$) can be given the two modes:

$$\begin{aligned} X = Y &:: \{X\} \Rightarrow \{X, Y\} \\ X = Y &:: \{Y\} \Rightarrow \{X, Y\} \end{aligned}$$

In practice, we maintain the set of variables which are required to be bound prior to execution (I) and the set of variables which are made bound by the execution (O). The above example is then written as:

$$\begin{aligned} X = Y &:: \{X\} \Rightarrow \{Y\} \\ X = Y &:: \{Y\} \Rightarrow \{X\} \end{aligned}$$

If A is the set of arguments of the procedure and M_{in} and M_{bind} are the required variables and the variables made bound then we have that

$$\langle M_{in}, M_{bind} \rangle \text{ partitions } A$$

So for example:

$$\begin{aligned} X = Y &:: \{X\} \Rightarrow \{Y\} \\ X = Y &:: \{Y\} \Rightarrow \{X\} \\ X \subseteq Y &:: \{X, Y\} \Rightarrow \varnothing \\ X \subseteq Y &:: \{Y\} \Rightarrow \{X\} \\ X \notin Y &:: \{X, Y\} \Rightarrow \varnothing \\ \mathit{add}(X, Y, Z) &:: \{X, Y\} \Rightarrow \{Z\} \\ \mathit{add}(X, Y, Z) &:: \{Y, Z\} \Rightarrow \{X\} \\ \mathit{add}(X, Y, Z) &:: \{X, Z\} \Rightarrow \{Y\} \\ \mathit{makenum}(N, R) &:: \{N\} \Rightarrow \{R\} \end{aligned}$$

In the rest of this section we systematically derive a mode derivation algorithm which given the modes of primitives derives the modes of compound formulae. Note that for efficiency reasons the mode analysis and the sequencing of conjunctions (ie. modifying the formula) will be done in a single pass.

3.1 Specification

Given a logic formula and an input mode M_{in} we say that the formula is *mode correct* if it satisfies the definition below. The definition can be viewed as an interpreter which takes a formula and an abstract substitution (the set of ground variables) and executes the formula over abstracted substitutions to determine the resulting set of ground variables O.

mode $mode\_correct(in, in, out)$

$mode\_correct((F_1 \wedge F_2), M_{in}, M_{out}) \Leftarrow$
$\quad mode\_correct(F_1, M_{in}, M_{tmp}) \wedge$
$\quad mode\_correct(F_2, M_{tmp}, M_{out}).$

$mode\_correct((F_1 \vee F_2), M_{in}, M_{out}) \Leftarrow$
$\quad mode\_correct(F_1, M_{in}, M_1) \wedge$
$\quad mode\_correct(F_2, M_{in}, M_2) \wedge M_1 = M_2 = M_{out}.$

$mode\_correct((\neg F_1), M_{in}, M_{out}) \Leftarrow$
$\quad mode\_correct(F_1, M_{in}, M_{tmp}) \wedge$
$\quad M_{tmp} = M_{in} = M_{out}.$

$mode\_correct(p(args), M_{in}, M_{out}) \Leftarrow$
$\quad p(args) :: In \Rightarrow Add \wedge$
$\quad In \subseteq M_{in} \wedge Add \cap M_{in} = \varnothing \wedge$
$\quad M_{out} = M_{in} \cup Add.$

$mode\_correct(X = \{D \bullet F\}, M_{in}, M_{out}) \Leftarrow$
$\quad mode\_correct(F, M_{in}, M_{tmp}) \wedge$
$\quad M_{tmp} = M_{in} \cup D \wedge M_{out} = M_{in} \cup \{X\}$

$mode\_correct(\exists D \bullet F, M_{in}, M_{out}) \Leftarrow$
$\quad mode\_correct(F, M_{in}, M_{tmp}) \wedge M_{out} \cup D = M_{tmp}.$

The rule for conjunction specifies sequential execution – given an input instantiation M_{in}, we find the mode of F_1 and then the resulting instantiation is passed to F_2.

The rule for disjunction is based on the intuition that the two formulae are interchangeable. There is an execution of $F_1 \vee F_2$ if there is an execution of F_1 or there is an execution of F_2.

The mode for a negation is based on the mode of the subformula. However, a negation can not bind variables and so the rule insists that the input and output instantiations are identical.

The atomic case involves a number of conditions. The first condition ($In \subseteq M_{in}$) checks that the variables needed by the atom's known mode are indeed ground. The second condition ($Add \cap M_{in} = \varnothing$) prevents any variables which are bound by the atom

from being given as input. This prevents derived modes (sometimes also known as implied modes). Implied modes (which exist in Mercury) are implemented by a transformation which calls the predicate in the original mode and then uses equality to test that the supplied value and the returned value match. This is undesirable since in the presence of infinite data types equality can not be assumed to be an available operation.

There is another reason why derived modes are a bad idea in this context. Derived modes are relatively harmless for the mode analysis, but they cause problems with the subtype analysis. Consider a predicate *append*, which requires (for type correctness) that all three arguments are sequences. The subtype analysis (see section 4) will replace $\mathit{append}(x, y, z) :: z \Rightarrow x, y$ with $\mathit{seq}(z) \land \mathit{append}(x, y, z) :: z \Rightarrow x, y$. If we now make use of an implicit mode and supply a value for y we have a call to *append* with an unchecked argument.

A set comprehension is executed by executing F and collecting the resulting values for D. Note that the execution of F must ground D and may not bind any other variables. The result of executing the set comprehension is simply to bind X.

Finally, the mode of an existentially quantified formula is based on the mode of the subformula. The introduced variables are initially unbound and since they are not visible after execution (since they are not in scope) we remove them from the output mode. Note that since D and M_{out} are disjoint the condition $M_{out} \cup D = M_{tmp}$ is equivalent to $M_{out} = M_{tmp} \setminus D$.

Mode correctness is defined with respect to an order of execution and the one order is not always appropriate. Hence after determining modes we need to modify the formula so that conjunctions are in the correct order.

It is probably desirable to convert to disjunctive normal form. The two arguments against are that (a) the size of the expression can grow, and (b) the relationship between the generated Mercury and the Z specification can become more obscure. The single major argument for the normalisation is that it increases the number of programs that can pass mode analysis. For example $(x = c \lor y = c) \land x = y$ cannot be given a consistent moding. However, by duplicating the equality yielding $(x = c \land x = y) \lor (y = x \land y = c)$ a moding becomes possible. One possible compromise is to only normalise a formula if the mode analysis fails.

3.2 Problems with the specification

The above definition (*mode_correct*) has three combinatorial inefficiencies:

1. The input mode, M_{in}, needs to be given (for $In \subseteq M_{in}$ in the atomic case);

2. There is a large number of variants of a formula; and

3. Predicates can have multiple modes.

We deal with the first by deriving a mode rather than working with instantiations. Effectively this corresponds to changing the mode declaration of *mode_correct* from (in, in, out)

to (in, out, out). This can be done systematically with minor changes to some of the conditions.

We deal with the second problem by generating a single variant which is representative of the equivalence class of variants (see section 3.3). This is done by generating the variants on the fly during the analysis of conjunctions.

Putting together these two changes we have an algorithm which works in two steps:

1. Assign modes to atoms

2. Derive the modes bottom-up, generating variants on the fly

The third inefficiency in the specification can be reduced using constraint propagation. For example if we have $x = t_1 \wedge compound([y, z, r], t_1)$ and t_1 is a local variable then t_1 *must* be bound by *compound* and hence the only mode possible for $=$ is the one which binds x. We can view the conjunction as shorthand for $x = compound([y, z, r])$. Identifying these chains (which result from the conversion of expressions to predicates) should be done in the general case.

3.3 Using topological sorting to reduce the number of variants

The second problem with the specification is that a given formula has a large number of variants. Many of these will be equivalent in the sense that two variants have the same modings. We thus would like to deal with *equivalence classes* of variants. In fact, for a given selection of modings for atoms the conjunct $p_1 \wedge \ldots \wedge p_n$ has at most a single unique mode. We can generate a single variant and ignore all others.

Proof: Consider $F :: X \Rightarrow Y = F_1 \wedge \ldots \wedge F_n$ where $F_i :: X_i \Rightarrow Y_i$. Given a variable x there are three cases: (i) $x \in Y_i$ and $x \notin X_j$. Hence $x \in Y$. (ii) $x \in Y_i$ and $x \in X_j$. For F_i to be executable x must be free. Hence $x \notin X$ and since the formula binds x we have $x \in Y$. (iii) $x \notin Y_i$ and $x \in X_j$. Since x is not bound by F and since it is required by F_j we have that $x \in X$. These three cases uniquely determine the mode of F from the modes of the F_i.

Definition: *We define $p \Leftrightarrow\!\!\!\rightarrow q$ ("p depends on q") if and only if $p :: A \Rightarrow B$, $q :: C \Rightarrow D$ and*

1. *$C \cap B = \varnothing$ – i.e. q does not need anything bound by p*

2. *$D \cap A \neq \varnothing$ – i.e. p needs something which is bound by q.*

We can deal with variants as follows:

1. Generate a graph with nodes p_1 to p_n where there is a directed arc from p_i to p_j iff $p_j \Leftrightarrow\!\!\!\rightarrow p_i$.

2. Topologically sort the graph. This gives us a canonical variant.

3. Execute the variant backwards to check whether it can be moded and if so, determine the needed input mode.

To execute backwards we begin with the output mode and apply in sequence each atom. If the atom has mode $X \Rightarrow Y$ then it can be applied if $X \cup Y \subseteq \Gamma$ (where Γ is the current mode) and we replace Γ with $\Gamma \setminus Y$.

Graphs which are not acyclic will fail the third step.

Example 1

Consider the following conjunction:

$$\begin{aligned} p_1 &:: a \Rightarrow b \\ p_2 &:: b, c \Rightarrow d \\ p_3 &:: a \Rightarrow c \\ p_4 &:: e \Rightarrow f \\ p_5 &:: a \Rightarrow e \end{aligned}$$

We have that

$$p_2 \Leftrightarrow\!\!\!\!\rightarrow p_1$$

$$p_2 \Leftrightarrow\!\!\!\!\rightarrow p_3$$

$$p_4 \Leftrightarrow\!\!\!\!\rightarrow p_5$$

One particular topological sort of this graph is the sequence *2, 1, 4, 3, 5.*
The output mode is *a, b, c, d, e, f.* We execute backwards …

a, b, c, d, e, f
$\quad \Rightarrow_2 \{b, c\} \cup \{d\} \subseteq \{a, b, c, d, e, f\}$
a, b, c, e, f
$\quad \Rightarrow_1 \{a\} \cup \{b\} \subseteq \{a, b, c, e, f\}$
a, c, e, f
$\quad \Rightarrow_4 \{e\} \cup \{f\} \subseteq \{a, c, e, f\}$
a, c, e
$\quad \Rightarrow_3 \{a\} \cup \{c\} \subseteq \{a, c, e\}$
a, e
$\quad \Rightarrow_5 \{a\} \cup \{e\} \subseteq \{a, e\}$
a

Conclusion: the conjunction is modable and the needed input is *a.* We have $p :: a \Rightarrow b, c, d, e, f \;=\; p_5 \wedge p_3 \wedge p_4 \wedge p_1 \wedge p_2$.

Example 2

Consider the following conjunction:

$$\begin{aligned} p_1 &:: a \Rightarrow b \\ p_2 &:: b \Rightarrow c \\ p_3 &:: c \Rightarrow a \end{aligned}$$

We have that

$$p_1 \Leftrightarrow\!\!\!\rightarrow p_2$$

$$p_2 \Leftrightarrow\!\!\!\rightarrow p_3$$

$$p_3 \Leftrightarrow\!\!\!\rightarrow p_1$$

One result of the topological sort algorithm is *3,2,1*. However, observe that attempting to derive the overall mode of the conjunction fails:

a, b, c
$\quad \Rightarrow_3 \{c\} \cup \{a\} \subseteq \{a, b, c\}$
b, c
$\quad \Rightarrow_2 \{b\} \cup \{c\} \subseteq \{b, c\}$
b
$\quad \not\Rightarrow_1 \{a\} \cup \{b\} \not\subseteq \{b\}$

3.4 Eliminating modes using the single binding restriction

As mentioned earlier, we do not allow derived (or implied) modes. Thus, we have the *single binding restriction*. Given a conjunction of atoms $p_1 \wedge \ldots \wedge p_n$ where each p_i has the mode $p_i :: A_i \Rightarrow B_i$. We require that:

$$\forall i, j : 1..n \mid i \neq j \bullet B_i \cap B_j = \varnothing$$

Once consequence of this is that certain modes can be eliminated. Consider as an example the conjunction $compound([a, b], c) \wedge c = d$. If *compound* has the single mode $a, b \Rightarrow c$ then we can rule out the mode $=:: d \Rightarrow c$ since this has c being bound twice.

3.5 The algorithm

The algorithm below summarises the improvements to the specification. Note that we treat $x = \{\overline{y} \mid F \bullet z\}$ as $x = \{z \mid (\exists \overline{y} \bullet F)\}$ The algorithm can also be presented as inference rules (see figure 1).

mode $mode(in, out)$

$mode(A, m_A)$

$mode(F_1 \wedge \ldots \wedge F_n, X) \Leftarrow$
$\quad \bigwedge mode(F_i, m_i)$
$\quad \wedge\ T = reverse(topsort([m_1, \ldots, m_n]))$
$\quad \wedge\ compose(T, \varnothing \Rightarrow \varnothing, T_2) \wedge T_2 = X.$

$mode(F_1 \vee \ldots \vee F_n, I \Rightarrow O) \Leftarrow$
$\quad \bigwedge mode(F_i, I_i \Rightarrow O_i)$
$\quad \wedge\ O = O_1 = \ldots = O_n$
$\quad \wedge\ I = \bigcup_{j=1}^{n} I_j$

$mode(\exists D \bullet F, I \Rightarrow (O \setminus D)) \Leftarrow$
$\quad mode(F, I \Rightarrow O) \wedge D \cap I = \varnothing$

$mode(X = \{D \mid F\}, I \Rightarrow X) \Leftarrow$
$\quad mode(F, I \Rightarrow O) \wedge D \cap I = \varnothing \wedge D = O$

$mode(\neg F, I \Rightarrow \varnothing) \Leftarrow mode(F, I \Rightarrow T), \wedge\ T = \varnothing$

$compose([], M, M)$
$compose([I \Rightarrow O \mid Ms], N \Rightarrow P, M) \Leftarrow$
$\quad M_1 = (N \cup (I \setminus P)) \Rightarrow (O \cup P))$
$\quad \wedge\ O \cap P = \varnothing$
$\quad \wedge\ compose(Ms, M_1, M)$

The rule for conjunction uses a topological sort to derive a variant and then uses *compose* to derive the mode and to check that the mode is consistent.

The rule for disjunction analysis presented earlier insists that the disjuncts have the same modes. The second condition in the algorithm actually relaxes this in that instead of requiring equivalent modes we only require *compatible* modes. Two modes are compatible if they can be weakened into the same mode. For example, if we have $F_1 :: x \Rightarrow z \vee F_2 :: y \Rightarrow z$ then the original rule will not be able to mode the formula. However, by weakening F_1 to $x, y \Rightarrow z$ and similarly for F_2 we can derive the mode $x, y \Rightarrow z$ for the disjunction.

The rules for the existential quantifier and for set comprehensions both require that the newly declared variables (D) be unbound ($D \cap I = \varnothing$).

In addition to the basic rules we can also use the definitions of the non-primitive connectives to derive their rules:

$$\frac{F_1 :: I_1 \Rightarrow O \quad \ldots \quad F_n :: I_n \Rightarrow O}{F_1 \vee \ldots \vee F_n :: (I_1 \cup \ldots \cup I_n) \Rightarrow O} \vee \qquad \frac{}{A :: m_A}$$

$$\frac{F_1 :: I_1 \Rightarrow O_1 \quad \ldots \quad F_n :: I_n \Rightarrow O_n \quad \forall j : 2..n \bullet O_j \cap (O_1 \cup \ldots \cup O_{j-1}) = \varnothing}{F_1 \wedge \ldots \wedge F_n :: J_1 \cup \ldots \cup J_n \Rightarrow O_1 \cup \ldots \cup O_n} \wedge$$

Where $J_1 = I_1$ and $J_{i+1} = I_{i+1} \setminus (O_1 \cup \ldots \cup O_i)$.

$$\frac{F :: I \Rightarrow O \quad O = \varnothing}{\neg F :: I \Rightarrow \varnothing} \neg \qquad \frac{F :: I \Rightarrow O \quad I \cap \overline{x} = \varnothing}{\exists \overline{x} \bullet F :: I \Rightarrow (O \setminus \overline{x})} \exists$$

Where $\overline{x}$ is a set of variables.

$$\frac{F :: I \Rightarrow O \quad O = \overline{y} \quad I \cap \overline{y} = \varnothing}{x = \{\overline{y} \mid F\} :: I \Rightarrow x} \textit{ set} \qquad \frac{F :: I \Rightarrow O \quad I \subseteq I'}{F :: I' \Rightarrow O} \textit{ weaken}$$

Figure 1: Rules for Mode Analysis

$$\begin{aligned}
&\mathit{mode}(F \Rightarrow G, (I_1 \cup I_2) \Rightarrow \varnothing) \Leftarrow \\
&\quad \mathit{mode}(F, I_1 \Rightarrow \varnothing) \wedge \mathit{mode}(G, I_2 \Rightarrow \varnothing) \\
&\mathit{mode}(F \Leftrightarrow G, (I_1 \cup I_2) \Rightarrow \varnothing) \Leftarrow \\
&\quad \mathit{mode}(F, I_1 \Rightarrow \varnothing) \wedge \mathit{mode}(G, I_2 \Rightarrow \varnothing) \\
&\mathit{mode}(\forall D \bullet F, I \Rightarrow \varnothing) \Leftarrow \\
&\quad \mathit{mode}(F, I \Rightarrow \varnothing) \wedge D \cap I = \varnothing
\end{aligned}$$

3.6 Example

Consider the formula $\exists z \bullet z = y \wedge x = z$. Let us analyse it.

$$\begin{aligned}
&\mathit{mode}(\exists z \bullet z = y \wedge x = z, I \Rightarrow O) \\
&\Leftrightarrow \mathit{mode}(z = y \wedge x = z, I_1 \Rightarrow O_1) \wedge \{z\} \cap I_1 = \varnothing \wedge I = I_1 \wedge O = O_1 \setminus \{z\} \\
&\Leftrightarrow \mathit{mode}(z = y, m_2 \equiv I_2 \Rightarrow O_2) \wedge \mathit{mode}(x = z, m_3 \equiv I_3 \Rightarrow O_3) \wedge \\
&\quad T = \mathit{topsort}([m_2, m_3]) \wedge \mathit{compose}(T, \varnothing \Rightarrow \varnothing, T_2) \wedge T_2 = I_1 \Rightarrow O_1 \wedge \{z\} \cap I_1 = \varnothing \\
&\quad \wedge I = I_1 \wedge O = O_1 \setminus \{z\}
\end{aligned}$$

There are now four cases depending on which modes we assign to the equality. We assume that $X = Y$ has the two modes $X \Rightarrow Y$ and $Y \Rightarrow X$.

Case 1

$m_2 \equiv I_2 \Rightarrow O_2 = z \Rightarrow y$
$m_3 \equiv I_3 \Rightarrow O_3 = x \Rightarrow z$

We then have that $m_2 \Leftrightarrow\!\!\!\rightarrow m_3$ and hence $\mathit{reverse}(\mathit{topsort}([m_2, m_3])) = [m_3, m_2]$.

$$
\begin{aligned}
&\Leftrightarrow \mathit{compose}([m_3 \equiv x \Rightarrow z, m_2], \varnothing \Rightarrow \varnothing, T_2) \wedge T_2 = I_1 \Rightarrow O_1 \wedge \{z\} \cap I_1 = \varnothing \\
&\qquad \wedge I = I_1 \wedge O = O_1 \setminus \{z\} \\
&\Leftrightarrow M_1 = \varnothing \cup (\{x\} \setminus \varnothing) \Rightarrow \{z\} \cup \varnothing = x \Rightarrow z \wedge \\
&\qquad \{z\} \cap \varnothing = \varnothing \wedge \mathit{compose}([z \Rightarrow y], x \Rightarrow z, T_2) \wedge \ldots \\
&\Leftrightarrow M_2 = \{x\} \cup (\{z\} \setminus \{z\}) \Rightarrow \{z\} \cup \{y\} = x \Rightarrow z, y \wedge \\
&\qquad \{y\} \cap \{z\} = \varnothing \wedge \mathit{compose}([], M_2, T_2) \wedge \ldots \\
&\Leftrightarrow T_2 = x \Rightarrow z, y \wedge T_2 = I_1 \Rightarrow O_1 \wedge \{z\} \cap I_1 = \varnothing \\
&\qquad \wedge I = I_1 \wedge O = O_1 \setminus \{z\} \\
&\Leftrightarrow I_1 = \{x\} \wedge O_1 = \{z, y\} \wedge \{z\} \cap \{x\} = \varnothing \\
&\qquad \wedge I = I_1 \wedge O = O_1 \setminus \{z\} \\
&\Leftrightarrow I = \{x\} \wedge O = \{y\}
\end{aligned}
$$

Thus the formula has the mode $x \Rightarrow y$ and is $\exists z \bullet (x = z :: x \Rightarrow z) \wedge (z = y :: z \Rightarrow y)$

Case 2

$m_2 \equiv I_2 \Rightarrow O_2 = z \Rightarrow y$
$m_3 \equiv I_3 \Rightarrow O_3 = z \Rightarrow x$

In this case the two modes are incomparable and so we can pick an arbitrary ordering.

$$
\begin{aligned}
&\Leftrightarrow \mathit{compose}([m_3 \equiv z \Rightarrow x, m_2], \varnothing \Rightarrow \varnothing, T_2) \wedge T_2 = I_1 \Rightarrow O_1 \wedge \{z\} \cap I_1 = \varnothing \\
&\qquad \wedge I = I_1 \wedge O = O_1 \setminus \{z\} \\
&\Leftrightarrow M_1 = \varnothing \cup (\{z\} \setminus \varnothing) \Rightarrow \{x\} \cup \varnothing = z \Rightarrow x \wedge \\
&\qquad \{x\} \cap \varnothing = \varnothing \wedge \mathit{compose}([z \Rightarrow y], z \Rightarrow x, T_2) \wedge \ldots \\
&\Leftrightarrow M_2 = (\{z\} \cup (\{z\} \setminus \{x\})) \Rightarrow \{y\} \cup \{x\} = z \Rightarrow x, y \wedge \\
&\qquad \{y\} \cap \{x\} = \varnothing \wedge \mathit{compose}([], z \Rightarrow x, y, T_2) \wedge \ldots \\
&\Leftrightarrow T_2 = z \Rightarrow x, y \wedge T_2 = I_1 \Rightarrow O_1 \wedge \{z\} \cap I_1 = \varnothing \wedge \ldots \\
&\Leftrightarrow I_1 = \{z\} \wedge O_1 = \{x, y\} \wedge \{z\} \cap \{z\} = \varnothing \wedge \ldots \\
&\Leftrightarrow \{z\} = \varnothing \wedge \ldots
\end{aligned}
$$

This particular assignment of modes has a single mode for the conjunction which requires that z is given. Since z is introduced by the quantifier it can't be given and hence the formula does not have a mode and can not be executed.

Case 3

$m_2 \equiv I_2 \Rightarrow O_2 = y \Rightarrow z$
$m_3 \equiv I_3 \Rightarrow O_3 = x \Rightarrow z$

Again, the two modes are incomparable and so we can select an arbitrary ordering.

$$\begin{aligned}
&\Leftrightarrow \mathit{compose}([m_3 \equiv x \Rightarrow z, m_2], \varnothing \Rightarrow \varnothing, T_2) \wedge T_2 = I_1 \Rightarrow O_1 \wedge \{z\} \cap I_1 = \varnothing \\
&\qquad \wedge I = I_1 \wedge O = O_1 \setminus \{z\} \\
&\Leftrightarrow M_1 = \varnothing \cup (\{x\} \setminus \varnothing) \Rightarrow \{z\} \cup \varnothing = x \Rightarrow z \wedge \\
&\qquad \{z\} \cap \varnothing = \varnothing \wedge \mathit{compose}([y \Rightarrow z], x \Rightarrow z, T_2) \wedge \ldots \\
&\Leftrightarrow M_2 = (\{x\} \cup (\{y\} \setminus \{z\})) \Rightarrow \{z\} \cup \{z\} = x, y \Rightarrow z \wedge \\
&\qquad \{z\} \cap \{z\} = \varnothing \wedge \mathit{compose}([], x, y \Rightarrow z, T_2) \wedge \ldots \\
&\Leftrightarrow \{z\} = \varnothing \wedge \ldots
\end{aligned}$$

In this case the two sub-formulae both attempt to bind z which violates the conditions – this is equivalent to a derived mode.

Case 4

$m_2 \equiv I_2 \Rightarrow O_2 = y \Rightarrow z$
$m_3 \equiv I_3 \Rightarrow O_3 = z \Rightarrow x$

We then have that $m_3 \Leftrightarrow\!\!\!\!\rightarrow m_2$ and hence $\mathit{reverse}(\mathit{topsort}([m_2, m_3])) = [m_2, m_3]$.

$$\begin{aligned}
&\Leftrightarrow \mathit{compose}([m_2 \equiv y \Rightarrow z, m_3], \varnothing \Rightarrow \varnothing, T_2) \wedge T_2 = I_1 \Rightarrow O_1 \wedge \{z\} \cap I_1 = \varnothing \\
&\qquad \wedge I = I_1 \wedge O = O_1 \setminus \{z\} \\
&\Leftrightarrow M_1 = \varnothing \cup (\{y\} \setminus \varnothing) \Rightarrow \{z\} \cup \varnothing = y \Rightarrow z \wedge \\
&\qquad \{z\} \cap \varnothing = \varnothing \wedge \mathit{compose}([z \Rightarrow x], y \Rightarrow z, T_2) \wedge \ldots \\
&\Leftrightarrow M_2 = \{y\} \cup (\{z\} \setminus \{z\}) \Rightarrow \{x\} \cup \{z\} = y \Rightarrow x, z \wedge \\
&\qquad \{x\} \cap \{z\} = \varnothing \wedge \mathit{compose}([], y \Rightarrow x, z, T_2) \wedge \ldots \\
&\Leftrightarrow T_2 = y \Rightarrow x, z \wedge T_2 = I_1 \Rightarrow O_1 \wedge \{z\} \cap I_1 = \varnothing \\
&\qquad \wedge I = I_1 \wedge O = O_1 \setminus \{z\} \\
&\Leftrightarrow I_1 = \{y\} \wedge O_1 = \{x, z\} \wedge \{z\} \cap \{y\} = \varnothing \\
&\qquad \wedge I = I_1 \wedge O = O_1 \setminus \{z\} \\
&\Leftrightarrow I = \{y\} \wedge O = \{x, z\} \setminus \{z\} = \{x\}
\end{aligned}$$

Thus the formula has the mode $y \Rightarrow x$ and is $\exists z \bullet (z = y :: y \Rightarrow z) \wedge (x = z :: z \Rightarrow x)$

The derivation can also be presented using the rules of figure 1. The fourth case above is[4]:

$$\dfrac{\dfrac{\overline{z = y :: y \Rightarrow z} \quad \overline{x = z :: z \Rightarrow x} \quad x \cap z = \varnothing}{z = y \wedge x = z :: y \cup (z \setminus z) \Rightarrow z \cup x = y \Rightarrow x, z}\,\wedge \quad y \cap z = \varnothing}{\exists z \bullet z = y \wedge x = z :: y \Rightarrow (x, z) \setminus z = y \Rightarrow x}\,\exists$$

[4]By an abuse of notation we write (e.g.) z for $\{z\}$

4 Subtype Analysis

Z's notion of type is fairly coarse. In order to animate a specification more information is required. For example, in Z a sequence is simply a special case of a set of pairs. However, if a function is defined with type $\mathrm{seq}\,\mathbb{Z} \pfun \mathrm{seq}\,\mathbb{Z}$ then any attempt to apply it to a set of pairs which doesn't represent a sequence should fail. Also, when considering whether operations can be animated, some operations can be performed in certain modes only with finite sets.

Although we could perform these checks at run-time it is desirable to derive as much information as possible at compile-time for two reasons. Firstly, efficiency. Many checks are obviously redundant. For example, when executing $\exists t \bullet t = x \frown y \land z = t \frown t :: \{x, y\} \Rightarrow \{z\}$ there is no need to check that t is a sequence. Secondly, we may want to use a different representation for certain subtypes. For example, a sequence could be represented as a list rather than as a set of pairs.

Unfortunately, analysing subtypes is hard, and since Z does not require subtype correctness there are Z specifications which would fail subtype analysis but which could be animated. For example:

S

$$r : \mathbb{P}(X \times Y)$$
$$x : X$$

$$\mathit{if}\ \mathit{hard\_test}\ \mathit{then} \{y : Y \bullet (x, y) \in r\}\ \mathit{else}\ \{r(x)\}$$

This suggests that subtype analysis can not be performed entirely at compile time and that some dynamic (run time) testing is needed.

However, it may be possible to determine both the sufficient and necessary subtypes for a given query. If the sufficient subtype conditions are met then no runtime checks are needed. If the necessary subtypes are not met then the query can be failed outright. Otherwise, runtime checks determine the result.

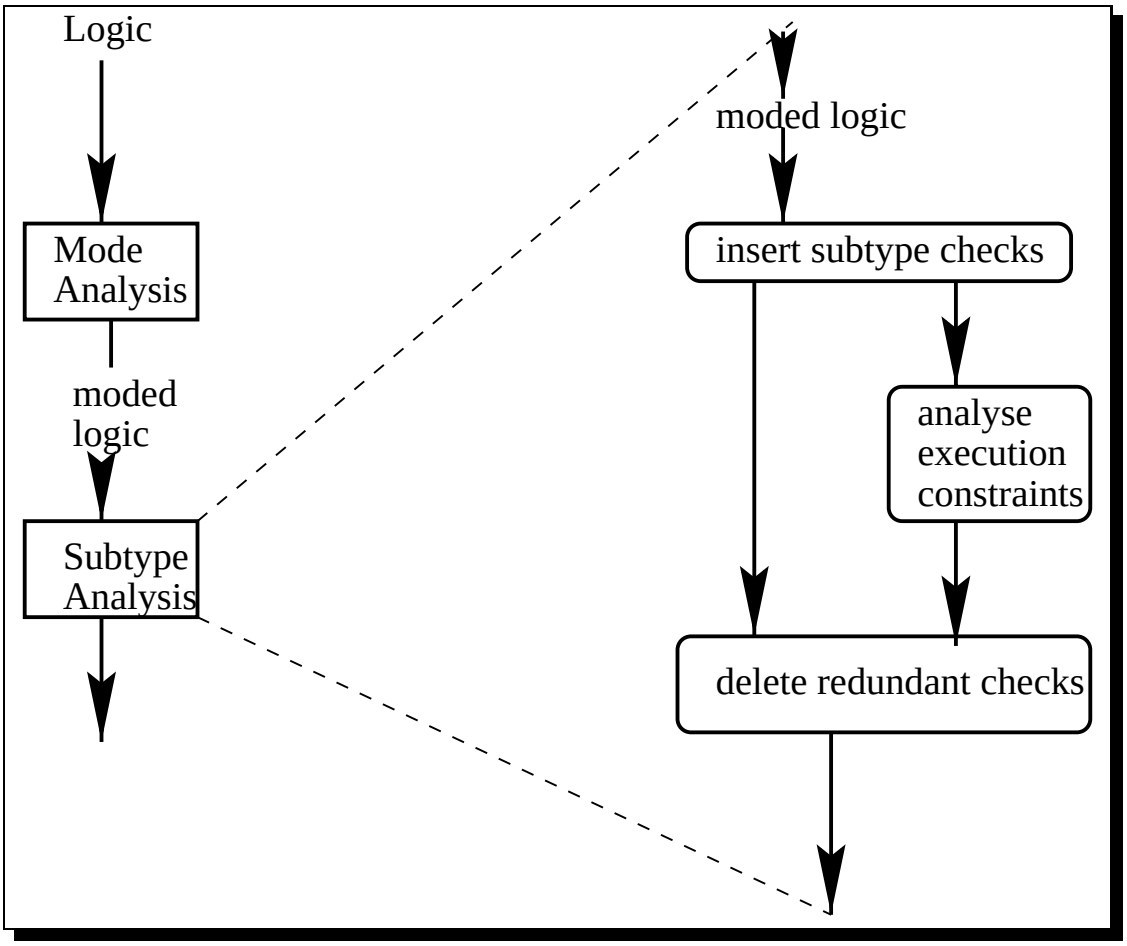
Another way of viewing the role of subtype analysis is the viewpoint of *soft typing*: there are tests which are required to be in the code. The analysis aims at eliminating some subset of these by proving that they can never fail.

The subtype analysis process is outlined in the diagram below. A few points need to be noted:

- Modes are needed in order to perform the subtype analysis.
- The analysis may need to modify the input formula by inserting checks which could not be eliminated.
- We take care that needed checks are inserted where they can be executed. For example, if we had $t = x \frown y :: \{x, y\} \Rightarrow \{t\}$ then the subtype checks would be inserted

to give $\mathrm{seq}\, x \wedge \mathrm{seq}\, y \wedge t = x \frown y \wedge \mathrm{seq}\, t$. Thus sequential executability is preserved by the subtype analysis.

- The analysis part of the process is an abstract interpretation.
- As is the case for mode analysis, the different phases are combined into a single pass.



4.1 What does subtype analysis do?

We begin with a brief motivating example. Consider the following formula:

$$\exists x, r \bullet x = y \frown z \wedge x = q \frown r$$

Suppose that the formula has been given mode $y, z \Rightarrow q$ and that the first conjunct has mode $y, z \Rightarrow x$ and the second conjunct has mode $x \Rightarrow q, r$.

The library operation $\frown$ (cat) is only applicable if its arguments are sequences (that is, partial functions from $\mathbb{N}$ to X with a domain of $1 \,..\, n$ for some n).

Thus, the first conjunct above is *really*

$$\mathrm{seq}\, y \wedge \mathrm{seq}\, z \wedge x = y \frown z \wedge \mathrm{seq}\, x$$

Note that we place constraints on input variables before the predicate and constraints on the output after the predicate. This makes no difference logically (as conjunction is commutative) but retains the ability of $\wedge$ to be treated as sequential conjunction (as in Prolog).

Thus, the whole formula is treated as:

$$\exists x, r \bullet (\mathrm{seq}\, y \wedge \mathrm{seq}\, z \wedge x = y \frown z \wedge \mathrm{seq}\, x \wedge \mathrm{seq}\, x \wedge x = q \frown r \wedge \mathrm{seq}\, r \wedge \mathrm{seq}\, q)$$

One property regarding $\frown$ which we may have is that for $x \frown y = z$ the constraint

$$\mathrm{seq}\, z \Leftrightarrow \mathrm{seq}\, y \wedge \mathrm{seq}\, x$$

holds. This constraints can be assumed to hold *after a successful execution of* $x \frown y = z$. Thus, we cannot eliminate the first two tests (on y and z). However, we can eliminate other tests.

After $x = y \frown z$ has executed we know that

$$\mathrm{seq}\, x \Leftrightarrow \mathrm{seq}\, y \wedge \mathrm{seq}\, z.$$

We also know (since we have explicit tests) that

$$\mathrm{seq}\, y \wedge \mathrm{seq}\, z.$$

We can therefore conclude that x must be a sequence and leave out both tests.

After $x = q \frown r$ has successfully executed we know that

$$\mathrm{seq}\, x \Leftrightarrow \mathrm{seq}\, q \wedge \mathrm{seq}\, r.$$

Since we know that x is a sequence we can conclude that both q and r must be sequences and the two tests can be left out.

This yields the final formula

$$\exists x, r \bullet (\mathrm{seq}\, y \wedge \mathrm{seq}\, z \wedge x = y \frown z \wedge x = q \frown r)$$

We now define the domains of the analysis and some useful operations on these domains.

4.2 Domains and operations

The subtype analysis will return a four tuple consisting of

1. **Requirements:** (R) These are the constraints that are implied by the Z type signature. These must be satisfied and in general this is done by a runtime check. The aim of subtype analysis is to eliminate as many of these as possible. Requirements are actually a part of the formula being analysed. We lift them out in order to allow us to move them around — in particular we want to be able to shift the (necessary) checks as early as possible.

2. **Constraints:** (C) This is a complex logical formula that represents constraints between the subtypes and sizes of various variables. For example if we have $x = y \frown z$ then we know that $\mathrm{seq}\, y \land \mathrm{seq}\, z \Leftrightarrow \mathrm{seq}\, x$. This knowledge is collected and is used to eliminate requirements.

3. **Solutions:** (S) The number of solutions of the predicate. In order to determine the size of a set comprehension we need to know (roughly) how many solutions the predicate has. Note that this requires mode information – in general the number of solutions to a predicate depends on which mode it is being used in: for example, $\mathit{append}(\mathit{in}, \mathit{in}, \mathit{out})$ has exactly a single solution whereas $\mathit{append}(\mathit{out}, \mathit{out}, \mathit{in})$ has a finite (non-zero) number of solutions.

4. **Modified formula:** (F') As noted, subtype checks may need to be inserted.

We define the following domains:

X: Variables.

B: Basic predicates – these are the simple subtype tests.

$$
\begin{array}{lcl}
B & ::= & X_{\mathsf{size}} \mid \mathsf{subtype}\ X \\
\mathsf{size} & ::= & \infty \mid \mathbb{F} \mid 0 \mid 1 \\
\mathsf{subtype} & ::= & \mathrm{reln} \mid \mathrm{seqbag} \mid \mathrm{seq} \mid \mathrm{bag} \mid \mathrm{pfun}
\end{array}
$$

X_∞ – X is definitely infinite

$X_\mathbb{F}$ – X is definitely finite (excluding zero and one)

X_1 – X is definitely of size 1

X_0 – X is definitely of size 0of size 0

$\mathrm{seq}\, X$ – X is a sequence (and hence is finite)

$\mathrm{bag}\, X$ – X is a bag

$\mathrm{seqbag}\, X$ – X is a bag *and* a sequence.

$\mathrm{pfun}\, X$ – X is a (partial) function

reln X – no requirement

R: Requirements[5]

$$R == X \pfun \mathsf{subtype} \times (\mathsf{size} \fun \mathsf{size})$$

C: Constraints.

$$C ::= B \mid C \vee C \mid C \wedge C \mid C \Rightarrow C \mid \ldots$$

S: Number of solutions $S == \mathsf{size} \fun \mathsf{size}$[6]

$\mathcal{L}_1$**:** Input – moded formulae.

$\mathcal{L}_2$**:** Output – moded formulae with needed subtype checks added.

We have the following relationships between the subtypes:

$$\mathrm{disjoint}\langle \mathbb{Z}_1, \mathbb{Z}_0, \mathbb{Z}_\infty, \mathbb{Z}_\mathbb{F} \rangle$$

$$\begin{array}{rcl} \mathrm{seq}\, X & \Rightarrow & X_0 \fun X_\mathbb{F} \\ \mathrm{seq}\, X & \Rightarrow & \mathrm{pfun}\ X \\ \mathrm{bag}\, X & \Rightarrow & \mathrm{pfun}\ X \\ \mathrm{seqbag}\ X & \Leftrightarrow & \mathrm{seq}\, X \wedge \mathrm{bag}\, X \end{array}$$

Having defined the domains we now proceed to define various operations which are needed in the subtype analysis algorithm. For each of **R**equirements, **C**onstraints and **S**izes we shall define a number of operations and an ordering.

We begin with constraints (**C**). The main operation that we have is

$$\_ \models \_ \ :: \ \mathbf{C} \times \mathbf{R} \to \mathit{Boolean}$$

This checks whether a requirement is subsumed by a set of constraints. In order to enable this to be efficiently implemented we shall represent constraints as sets of clauses. Thus, instead of using $\wedge$ and $\vee$ to combine constraints, we define $\dot{\wedge}$ and $\dot{\vee}$ which operate on sets of clauses (see section 4.4). We shall also need to be able to project out variables:

$$\_ \setminus \_ \ :: \ \mathbf{C} \times \mathbb{P}\, \mathbf{X} \to \mathbf{C}$$

We do not define an ordering over constraints.

[5] "$\pfun$" means a finite partial function.

[6] We make use of *ranges* $(s_1 - s_2)$ where the s_i are sizes. We require that $s_1 \leq s_2$.

Now to requirements (**R**). We have two binary connectives which return a requirement. The first returns the requirement R such that R implies both arguments. The second returns R such that R implies the disjunction of the arguments. Let $R_1\ x = (T_1, L_1 \Leftrightarrow H_1)$ and $R_2\ x = (T_2, L_2 \Leftrightarrow H_2)$, then we can define these operators as follows:[7]

$$\begin{array}{rcl} \_ \sqcap \_ & :: & \mathbf{R} \times \mathbf{R} \to \mathbf{R} \\ (R_1 \sqcap R_2)\ x & = & (T_1 \dot{\sqcap} T_2, max(L_1, L_2) \Leftrightarrow min(H_1, H_2)) \\ \_ \sqcup \_ & :: & \mathbf{R} \times \mathbf{R} \to \mathbf{R} \\ (R_1 \sqcup R_2)\ x & = & (T_1 \dot{\sqcup} T_2, min(L_1, L_2) \Leftrightarrow max(H_1, H_2)) \end{array}$$

We also define the notation R^I and $R^{\setminus I}$ to respectively yield the sub-requirement which only pertains to I and the sub-requirement which does *not* mention I. They can be defined as:

$$\begin{array}{rcl} \_^{\frown} & :: & \mathbf{R} \times D \to \mathbf{R} \\ R^I & = & I \lhd R \\ \_^{\setminus\frown} & :: & \mathbf{R} \times D \to \mathbf{R} \\ R^{\setminus I} & = & I \not\Leftrightarrow R \end{array}$$

We shall sometimes need to treat a requirement as a constraint or a logic formula and thus we define a translation:

$$\begin{array}{rcl} \ulcorner \_ \urcorner & :: & \mathbf{R} \nrightarrow \mathcal{L} \\ \ulcorner R \urcorner & \triangleq & \bigwedge \{ t(x) \wedge \bigvee \{ x_s \mid s_1 \leq s \leq s_2 \} \mid x \mapsto (t, s_1 \Leftrightarrow s_2) \in R \} \end{array}$$

Finally, in one of the rules we shall need to introduce an operation which takes two requirements and returns the parts of the first requirement which are not subsumed by the second requirement. This is used in the disjunction rule where the requirements of each subformula which are not subsumed by the combined requirement $(R_1 \sqcup R_2)$ are added to the subformula. The operator $(\boxminus)$ is always applied in the context $R_1 \boxminus (R_1 \sqcup R_2)$. Let $R_1\ x = (T_1, L_1 \Leftrightarrow H_1)$ and $(R_1 \sqcup R_2)\ x = (T_2, L_2 \Leftrightarrow H_2)$, then:

$$\begin{array}{rcl} \_ \boxminus \_ & :: & \mathbf{R} \times \mathbf{R} \to \mathbf{R} \\ (R_1 \boxminus (R_1 \sqcup R_2))\ x & = & (T_1 \boxminus_T T_2, L_1 \Leftrightarrow H_1 \boxminus_S L_2 \Leftrightarrow H_2) \\ \_ \boxminus_T \_ & :: & \mathsf{subtype} \times \mathsf{subtype} \to \mathsf{subtype} \\ \_ \boxminus_S \_ & :: & \mathbf{S} \times \mathbf{S} \to \mathbf{S} \\ (L_1 \Leftrightarrow H_1) \boxminus_S (L_2 \Leftrightarrow H_2) & = & L_1 \Leftrightarrow H_1, \text{if } L_1 \Leftrightarrow H_1 < L_2 \Leftrightarrow H_2 \\ (L_1 \Leftrightarrow H_1) \boxminus_S (L_2 \Leftrightarrow H_2) & = & 0 \Leftrightarrow \infty, \text{otherwise} \end{array}$$

[7] We complete each requirement by adding in the variables which are present in the other requirement and mapping them to $(reln, 0 - \infty)$.

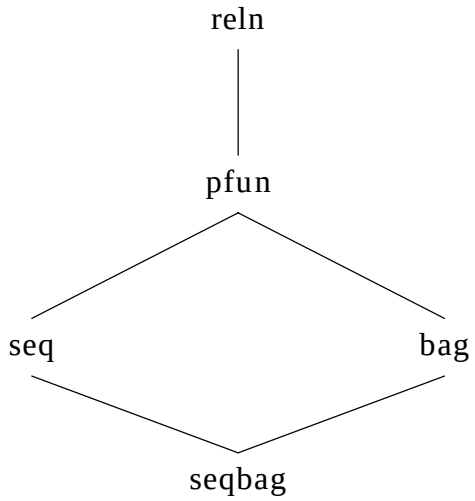
The definitions of the operations on requirements have made use of operations on sizes and on subtypes which we shall now define. The operations which we need to define are:

$$
\begin{aligned}
\_max\_ &:: \mathsf{size} \times \mathsf{size} \to \mathsf{size} \\
\_min\_ &:: \mathsf{size} \times \mathsf{size} \to \mathsf{size} \\
\_ \leq \_ &:: \mathbf{S} \times \mathbf{S} \to \mathit{Boolean} \\
\_ \dot{\sqcap} \_ &:: \mathsf{subtype} \times \mathsf{subtype} \to \mathsf{subtype} \\
\_ \dot{\sqcup} \_ &:: \mathsf{subtype} \times \mathsf{subtype} \to \mathsf{subtype} \\
\_ \boxminus_T \_ &:: \mathsf{subtype} \times \mathsf{subtype} \to \mathsf{subtype}
\end{aligned}
$$

The definitions of *max* and *min* are standard and follow from the ordering $0 < 1 < \mathbb{F} < \infty$. We can extend this ordering to size ranges (ie. pairs of sizes) as follows:

$$L_1 \Leftrightarrow H_1 \leq L_2 \Leftrightarrow H_2 \quad \Leftrightarrow \quad L_1 \geq L_2 \wedge H_1 \leq H_2$$

We can define the remaining operations by viewing the subtypes as a lattice and considering the natural operations (glb and lub) on the lattice. We define an ordering on subtypes as follows:



The operation $\dot{\sqcap}$ corresponds to the notion of "and" – ie. the greatest lower bound (glb).

$\dot{\sqcap}$	reln	pfun	seq	bag	seqbag
reln	reln	pfun	seq	bag	seqbag
pfun	pfun	pfun	seq	bag	seqbag
seq	seq	seq	seq	seqbag	seqbag
bag	bag	bag	seqbag	bag	seqbag
seqbag	seqbag	seqbag	seqbag	seqbag	seqbag

The operation $\dot{\sqcup}$ corresponds to the notion of "or" – ie. the least upper bound (lub).

$\dot{\sqcup}$	reln	pfun	seq	bag	seqbag
reln	reln	reln	reln	reln	reln
pfun	reln	pfun	pfun	pfun	pfun
seq	reln	pfun	seq	pfun	seq
bag	reln	pfun	pfun	bag	bag
seqbag	reln	pfun	seq	bag	seqbag

We can similarly define $\dot{\sqcap}_T$.

$\dot{\sqcap}_T$	reln	pfun	seq	bag	seqbag
reln	reln	reln	reln	reln	reln
pfun	pfun	reln	reln	reln	reln
seq	seq	seq	reln	seq	reln
bag	bag	bag	bag	reln	reln
seqbag	seqbag	seqbag	bag	seq	reln

We now define notions of conjunction, disjunction and negation over size ranges. We define $\otimes$, $\oplus$ and $\widetilde{\Leftrightarrow}$in terms of auxiliary functions[8] which operate on the end points of a range:

$$
\begin{array}{rcl}
\_ \otimes \_ & :: & \mathbf{S} \times \mathbf{S} \to \mathbf{S} \\
(L_1 \Leftrightarrow H_1) \otimes (L_2 \Leftrightarrow H_2) & = & (L_1 \otimes L_2) \Leftrightarrow (H_1 \otimes H_2) \\
\_ \oplus \_ & :: & \mathbf{S} \times \mathbf{S} \to \mathbf{S} \\
(L_1 \Leftrightarrow H_1) \oplus (L_2 \Leftrightarrow H_2) & = & (L_1 \oplus L_2) \Leftrightarrow (H_1 \oplus H_2) \\
\widetilde{\_} & :: & \mathbf{S} \to \mathbf{S} \\
\widetilde{L \Leftrightarrow H} & = & \widetilde{H} \Leftrightarrow \widetilde{L}
\end{array}
$$

Note that $\widetilde{X \Leftrightarrow Y} = \widetilde{Y} \Leftrightarrow \widetilde{X}$ since the operation is anti-monotonic.

$\otimes$	0	1	$\mathbb{F}$	∞
0	0	0	0	0
1	0	1	$\mathbb{F}$	∞
$\mathbb{F}$	0	$\mathbb{F}$	$\mathbb{F}$	∞
∞	0	∞	∞	∞

$\oplus$	0	1	$\mathbb{F}$	∞
0	0	1	$\mathbb{F}$	∞
1	1	$\mathbb{F}$	$\mathbb{F}$	∞
$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$	∞
∞	∞	∞	∞	∞

S	$\widetilde{S}$
0	1
1	0
$\mathbb{F}$	0
∞	0

[8] We use the same name for both the auxiliary and primary functions.

4.3 The algorithm

We now have the building blocks we need to define the subtype analysis.

A primary intuition in the analysis is that a formula $F :: I \Rightarrow O$ with requirement R is treated as $\ulcorner R^I \urcorner \wedge F \wedge \ulcorner R^O \urcorner$. Actually, the requirement is not inserted – we prefer to return the requirement for insertion earlier. A requirement is only inserted when it can not be pushed earlier. Note that a requirement that has been inserted (ie. a subtype check) must signal an error if the condition being tested doesn't hold, it can not simply fail. The reason for this is that we need to be able to push checks up past negations so that $\neg\ (\ulcorner R^I \urcorner \wedge F \wedge \ulcorner R^O \urcorner)$ is operationally equivalent to $\ulcorner R^I \urcorner \wedge \neg\ F \wedge \ulcorner R^O \urcorner$.

The subtype analysis is defined as a function from a moded formula to a four-tuple consisting of:

1. The requirements of the formula;
2. The constraints which hold after the formula has been successfully executed;
3. The possible number of solutions to the formula; and
4. The modified formula.

Its definition can be found on page 27. We now briefly explain each case.

The handling of a disjunction ($F \vee G$) is based on the idea of finding the common requirement ($R_1 \sqcup R_2$), this is the requirement of the disjunction. If the requirement of F (R_1) is stronger than the common requirement then F is modified to include the additional requirements.

For example, suppose that

$$\begin{aligned} R_1 &= \{x \mapsto (\text{pfun}, 1 \Leftrightarrow \infty)\} \\ R_1 &= \{x \mapsto (\text{pfun}, 1 \Leftrightarrow \mathbb{F})\} \end{aligned}$$

then

$$\begin{aligned} R_1 \sqcup R_2 &= \{x \mapsto (\text{pfun} \dot{\sqcup} \text{seq}, \mathit{min}(1,0) \Leftrightarrow \mathit{max}(\infty, \mathbb{F}))\} \\ &= \{x \mapsto (\text{pfun}, 0 \Leftrightarrow \infty)\} \end{aligned}$$

the first disjunct has excess R_F and the second disjunct has excess requirements R_G.

$$\begin{aligned} R_F &= R_1 \boxminus (R_1 \sqcup R_2) \\ &= \{x \mapsto (\text{pfun}, 1 \Leftrightarrow \infty)\} \boxminus \{x \mapsto (\text{pfun}, 0 \Leftrightarrow \infty)\} \\ &= \{x \mapsto (\text{pfun} \boxminus_T \text{pfun}, 1 \Leftrightarrow \infty \boxminus_S 0 \Leftrightarrow \infty)\} \\ &= \{x \mapsto (\text{reln}, 1 \Leftrightarrow \infty)\} \\ R_G &= R_2 \boxminus (R_1 \sqcup R_2) \\ &= \{x \mapsto (\text{seq}, 0 \Leftrightarrow \mathbb{F})\} \boxminus \{x \mapsto (\text{pfun}, 0 \Leftrightarrow \infty)\} \\ &= \{x \mapsto (\text{seq} \boxminus_T \text{pfun}, 0 \Leftrightarrow \mathbb{F} \boxminus_S 0 \Leftrightarrow \infty)\} \\ &= \{x \mapsto (\text{seq}, 0 \Leftrightarrow \mathbb{F})\} \end{aligned}$$

F and G are modified with their respective excess requirements to yield the modified formula:

$$(\ulcorner R_F \urcorner \wedge F) \vee (\ulcorner R_G \urcorner \wedge G)$$

The number of solutions and the constraint are straightforward.

We now consider negation. The only interesting aspect to this case is that the constraint is discarded—since a successful execution of $\neg F$ corresponds to a *failed* execution of F, F's constraint is irrelevant.

When analysing an atom the requirement is the relevant Z subtype declaration. The formula returned is $A \wedge \ulcorner R_2^O \urcorner$ where R_2 is the requirements applicable to the output variables which are not consequences of the constraint and the requirements on input variables.

The case for existential quantification is straightforward. We simply project over the variable(s) being quantified.

A set comprehension $x = \{D \mid F\}$ needs F's requirements in order to execute F. There is always exactly one solution and we (in general) know nothing about the value of x except that its size relates directly to the possible number of solutions of F.

The final and most complex case is conjunction. Consider $F_1 \wedge \ldots \wedge F_n$. Assume that each of the F_i has been analysed to yield R_i, C_i, S_i and F_i'. We then define the desired values accordingly:

$$\begin{aligned} C &= \bigwedge C_i \\ K_1 &= \varnothing \\ K_i &= K_{i-1} \wedge C_{i-1} \wedge R_{i-1} \\ R_i^* &= \{t \in R_i \mid K_i \not\models t\} \\ R &= \bigsqcap R_i^* \\ S &= \bigotimes S_i \end{aligned}$$

The algorithm analyses each F_i in turn. We maintain the evolving knowledge (K), requirements (R), number of possible solutions (S) and constraint (C). The second clause of *staconj* analyses an F_i and (i) adds the requirements and constraints to the knowledge that we have (which is relevant after the successful execution of the F_i), (ii) combines the constraints and requirements[9], and (iii) updates the number of solutions. Once all conjuncts have been processed we return the accumulated requirement, number of solutions and constraint.

An interesting aspect of the subtype analysis of conjunctions is the modification of the formula. The intuition mentioned earlier suggests that we view

$$\ulcorner R_1^I \urcorner \wedge F_1 \wedge \ulcorner R_1^O \urcorner \wedge \ldots \wedge \ulcorner R_n^I \urcorner \wedge F_n \wedge \ulcorner R_n^O \urcorner$$

as:

$$\ulcorner R_1^I \urcorner \wedge \ldots \wedge \ulcorner R_n^I \urcorner \wedge F_1 \wedge \ldots \wedge F_n \wedge \ulcorner R_1^O \urcorner \wedge \ldots \wedge \ulcorner R_n^O \urcorner$$

[9] Actually, we filter out requirements which are subsumed by the knowledge.

While these are logically equivalent, they are not operationally equivalent. A key point is that the input mode of a conjunction is *not* the union of the inputs of it's constituents. Requirements which pertain to inputs of the conjunction can be commuted to its front (and hence returned as requirements). Requirements which pertain to other variables need to be added after the point where the value for the variable is generated.

For example, consider $p(x,y) :: x \Rightarrow y$ and $q(y,z) :: y \Rightarrow z$ where q requires that y be finite. The mode of $p(x,y) \wedge q(y,z)$ is $x \Rightarrow y,z$. The requirement on y cannot be returned since y is not an input to the conjunction. Instead, we need to insert a test for y after the call to p yielding $p(x,y) \wedge y_{0-\mathbb{F}} \wedge q(y,z)$.

$$
\begin{array}{l}
sta(F_1 \wedge \ldots \wedge F_n)_{m \equiv I \Rightarrow O} \;=\; (R^I, S, C, \bigwedge(add\ R^{\setminus I}\ F')) \\
\quad \text{Where } (R,S,C,F') \;=\; staconj(F_1 \wedge \ldots \wedge F_n, \\
\qquad\qquad\qquad\qquad true, [], \varnothing, 1 \Leftrightarrow 1, true) \\
sta(F \vee G)_{m \equiv I \Rightarrow O} \;= \\
\quad (R_1 \sqcup R_2, S_1 \oplus S_2, C_1 \vee C_2, \\
\qquad\quad (\ulcorner R_F \urcorner \wedge F') \vee (\ulcorner R_G \urcorner \wedge G')) \\
\quad Where \\
\qquad R_F = R_1 \boxminus (R_1 \sqcup R_2) \\
\qquad R_G = R_2 \boxminus (R_1 \sqcup R_2) \\
\qquad (R_1, S_1, C_1, F') = sta(F)_{m_1 \equiv I_1 \Rightarrow O_1} \\
\qquad (R_2, S_2, C_2, G') = sta(G)_{m_2 \equiv I_2 \Rightarrow O_2} \\
sta(\neg F)_{m \equiv I \Rightarrow \varnothing} \;=\; (R_1, \widetilde{S_1}, true, \neg F') \\
\quad Where\ (R_1, S_1, C_1, F') = sta(F)_{m_1 \equiv I_1 \Rightarrow O_1} \\
sta(A)_{m \equiv I \Rightarrow O} \;=\; (R^I_A, S_A, C_A, A \wedge \ulcorner R^O_2 \urcorner) \\
\quad \text{Where } R_2 = \{t \mid t \in R^O_A \wedge (\ulcorner R^I_A \urcorner, C_A \not\models t)\} \\
sta(\exists D \bullet F)_{m \equiv I \Rightarrow O} = \\
\quad (R_1^{\setminus D}, S_1, C_1 \setminus D, \exists D \bullet F') \\
\quad Where\ (R_1, S_1, C_1, F') = sta(F)_{m_1 \equiv I_1 \Rightarrow O_1} \\
sta(x = \{D \mid F\})_{m \equiv I \Rightarrow O} = \\
\quad (R_1, 1 \Leftrightarrow 1, \ulcorner x \mapsto (true, S_1) \urcorner, x = \{D \mid F'\}) \\
\quad Where\ (R_1, S_1, C_1, F') = sta(F)_{m_1 \equiv I_1 \Rightarrow O_1}
\end{array}
$$

$$
\begin{array}{l}
staconj(true, K, Gs, R, S, C) \;=\; (R, S, C, reverse(Gs)) \\
staconj(F_i \wedge F_{i+1} \wedge \ldots \wedge F_n, K, Gs, R, S, C) \;= \\
\quad staconj(F_{i+1} \wedge \ldots \wedge F_n, C_i \wedge \ulcorner R_i \urcorner \wedge K, \\
\qquad\qquad [F'_i \mid Gs], R^*_i \sqcap R, S_i \otimes S, C_i \wedge C) \\
\quad \text{Where} \\
\qquad (R_i, S_i, C_i, F'_i) \;=\; sta(F_i) \\
\qquad R^*_i \;=\; \{t \mid t \in R_i \wedge K \not\models t\} \\
add\ \varnothing\ F \;=\; F \\
add\ (\{x \mapsto r\} \cup R)\ F \;=\; add\ R\ (add'\ x\ \ulcorner x \mapsto r \urcorner\ F) \\
add'\ x\ G\ [\,] = \textbf{error} \\
add'\ x\ G\ [F :: I \Rightarrow O \mid Fs] = \\
\quad \textbf{if}\ x \in O\ \textbf{then}\ [F, G \mid Fs]\ \textbf{else}\ [F \mid add'\ x\ G\ Fs]
\end{array}
$$

The derived connectives are:

$$
\begin{array}{rcl}
\forall D \bullet F & \equiv & \neg\ \exists D \bullet \neg F \\
F \Rightarrow G & \equiv & (\neg F) \vee G \\
F \Leftrightarrow G & \equiv & (F \wedge G) \vee ((\neg F) \wedge (\neg G)) \\
if\ F\ G\ H & \equiv & (F \wedge G) \vee H
\end{array}
$$

The rules for derived connectives can be derived from these rules. The derivation of the rule for $\Leftrightarrow$ can be found in appendix A.

$$
\begin{array}{l}
sta(F \Leftrightarrow G)_{m \equiv I \Rightarrow \varnothing} \;= \\
\quad (R_1 \sqcap R_2, (S_1 \otimes S_2) \oplus (\widetilde{S_1} \otimes \widetilde{S_2}), true, F' \Leftrightarrow G') \\
\quad \text{Where} \\
\qquad (R_1, S_1, C_1, F') = sta(F)_{m_1 \equiv I_1 \Rightarrow \varnothing} \\
\qquad (R_2, S_2, C_2, G') = sta(G)_{m_2 \equiv I_2 \Rightarrow \varnothing} \\
sta(\forall D \bullet F)_{m \equiv I \Rightarrow \varnothing} = (R_1, \widetilde{\widetilde{S_1}}, true, \forall D \bullet F') \\
\quad \text{Where} (R_1, S_1, C_1, F') = sta(F)_{m_1 \equiv I_1 \Rightarrow \varnothing} \\
sta(F \Rightarrow G)_m = sta(G \vee \neg F)_m
\end{array}
$$

4.4 Representing and manipulating constraints

Working with sets of general logic formulae is inefficient. We avoid this by representing constraints as sets of *clauses*. We have the following collected definitions from [3, 4]:

$C ::= \mathit{true} \mid \mathit{false} \mid C \wedge C \mid C \vee C \mid A$

clause $::= C \rightarrow A$ Where C can not contain A.

We can combine sets of clauses using the following definitions:

$$
\begin{array}{rcl}
F \dot{\wedge} F' & = & \bigwedge_{i:g_i \in F \cap F'} (C_i \vee C'_i \rightarrow g_i) \wedge \bigwedge_{i:g_i \in F \setminus F'} (C_i \rightarrow g_i) \wedge \bigwedge_{i:g_i \in F' \setminus F} (C'_i \rightarrow g_i) \\
F \dot{\vee} F' & = & \{(C \wedge C') \rightarrow g \mid C \rightarrow g \in F, C' \rightarrow g \in F'\} \\
F \setminus g_k & = & \bigwedge_{i:g_i \in F \setminus \{g_k\}} (C_i[g_k/C_k][g_i/\mathit{false}] \rightarrow g_i)
\end{array}
$$

As the following properties show, these are safe approximations to the standard logical connectives:

$$
\begin{array}{rcl}
F \dot{\wedge} F' & \Leftrightarrow & F \wedge F' \\
F \vee F' & \Rightarrow & F \dot{\vee} F' \\
F & \Rightarrow & F \setminus g_k
\end{array}
$$

Once we have transformed a general constraint into a set of clauses we can use standard results [6] to replace a general theorem prover with a top down resolution search a la Prolog without losing completeness. This is significantly more efficient.

4.5 Example

We now consider applying the above algorithm to the previously presented informal example.

Consider $\exists x, r \bullet x = y \frown z \wedge x = q \frown r$ where the overall mode is $y, z \Rightarrow q$ and the first $\frown$ is in mode $y, z \Rightarrow x$ and the second in mode $x \Rightarrow q, r$.

$$
\begin{array}{l}
\mathit{sta}(\exists x, r \bullet x = y \frown z \wedge x = q \frown r)_{y,z \Rightarrow q} = (\\
\quad R = R_0^{\setminus x,r} = R_1 = \{y \mapsto (\mathrm{seq}, 0 \Leftrightarrow \mathbb{F}), z \mapsto (\mathrm{seq}, 0 \Leftrightarrow \mathbb{F})\} \\
\quad S = S_0 = 1 \Leftrightarrow \mathbb{F} \\
\quad C = C_0 \setminus x, r = \mathrm{seq}\, y \wedge \mathrm{seq}\, z \Rightarrow \mathrm{seq}\, q \\
\quad F = \exists D \bullet F'_0 = F)
\end{array}
$$

$$
\begin{array}{l}
\mathit{sta}(x = y \frown z \wedge x = q \frown r)_{y,z \Rightarrow x,q,r} = (\\
\quad R_0 = R_1^{y,z} = R_1 \\
\quad S_0 = S_1 \\
\quad C_0 = C_1 \\
\quad F'_0 = \bigwedge(\mathit{add}\ R_0^{\setminus y,z}\ F_1) = \bigwedge F_1 = F_4 \wedge F_5
\end{array}
$$

$$staconj(x = y \frown z \land x = q \frown r, true, [], \varnothing, 1 \Leftrightarrow 1, true) = ($$
$$R_1 = R_2 = R_4$$
$$S_1 = S_2 = S_4 \otimes S_5 = S_5$$
$$C_1 = C_2 = C_4 \land C_5$$
$$F_1 = F_2 = [F_4, F_5]$$
$$where\ R_4^* = \{t \mid t \in R_4 \land true \not\models t\} = R_4$$

$$staconj(x = q \frown r, C_4 \land \ulcorner R_4 \urcorner, [F_4], R_4^*, S_4, C_4) = ($$
$$R_2 = R_3$$
$$S_2 = S_3$$
$$C_2 = C_3$$
$$F_2 = F_3$$
$$where\ R_5^* = \{t \mid t \in R_5 \land C_4 \land \ulcorner R_4 \urcorner \not\models t\} = \varnothing$$

$$staconj(true, C_5 \land \ulcorner R_5 \urcorner \land C_4 \land \ulcorner R_4 \urcorner,$$
$$[F_5, F_4], R_5^* \sqcap R_4^*, S_5 \otimes S_4, C_5 \land C_4) = ($$
$$R_3 = R_4^* \sqcap R_5^* = R_4 \sqcap \varnothing = R_4$$
$$S_3 = S_5 \otimes S_4$$
$$C_3 = C_5 \land C_4$$
$$F_3 = reverse[F_5, F_4] = [F_4, F_5]$$

$$sta(x = y \frown z)_{y,z \Rightarrow x} = ($$
$$R_4 = \{y \mapsto (\mathrm{seq}, 0 \Leftrightarrow \mathbb{F}), z \mapsto (\mathrm{seq}, 0 \Leftrightarrow \mathbb{F})\}$$
$$S_4 = 1 \Leftrightarrow 1$$
$$C_4 = \mathrm{seq}\, x \Leftrightarrow \mathrm{seq}\, y \land \mathrm{seq}\, z$$
$$F_4 = x = y \frown z)$$

Note that the requirements on the output variable follow from the constraint.

$$sta(x = q \frown r)_{x \Rightarrow q,r} = ($$
$$R_5 = \{x \mapsto (\mathrm{seq}, 0 \Leftrightarrow \mathbb{F})\}$$
$$S_5 = 1 \Leftrightarrow \mathbb{F}$$
$$C_5 = \mathrm{seq}\, x \Leftrightarrow \mathrm{seq}\, q \land \mathrm{seq}\, r$$
$$F_5 = x = q \frown r)$$

Note that the requirements on the output variables follow from the constraint.

5 Complete Example

We now trace through a complete example. A typical pattern in a state based Z specification with a state invariant is the following:

$$invariant[s] \land s' = \ldots s \ldots \land invariant[s']$$

for example

$$\#s < k \wedge s' = s \cup \{v\} \wedge \#s' < k$$

This is translated to the following logic formula:

$$\begin{array}{l} \mathit{card}(s, cs) \wedge \mathit{lt}(cs, k) \wedge \\ \mathit{makeset}(v, sv) \wedge \mathit{union}(s, sv, un) \wedge \mathit{eq}(s', un) \wedge \\ \mathit{card}(s', cs') \wedge \mathit{lt}(cs', k) \end{array}$$

Let us assume that the following modes are known for primitive operations:

$$\begin{array}{l} \mathit{card} :: 1 \Rightarrow 2 \\ \mathit{lt} :: 1, 2 \Rightarrow \\ \mathit{makeset} :: 1 \Rightarrow 2 \\ \mathit{union} :: 1, 2 \Rightarrow 3 \\ \qquad :: 1, 3 \Rightarrow 2 \\ \qquad :: 2, 3 \Rightarrow 1 \\ \mathit{eq} :: 1 \Rightarrow 2 \\ \quad :: 2 \Rightarrow 1 \\ \quad :: 1, 2 \Rightarrow \end{array}$$

There are nine possible mode combinations for the formula. However, since *makeset* binds *sv*, union cannot also bind *sv* and so we have six remaining modes to consider:

1. $\mathit{union} :: s, sv \Rightarrow un$, $\mathit{eq} :: s' \Rightarrow un$:
   eq and *union* cannot both bind *un*. No mode.

2. $\mathit{union} :: s, sv \Rightarrow un$, $\mathit{eq} :: un \Rightarrow s'$:
   The resulting mode is: $s, k, v \Rightarrow cs, sv, un, s', cs'$. One possible moded formula is:

   $$\begin{array}{l} \mathit{makeset}_{io}(v, sv) \wedge \mathit{union}_{iio}(s, sv, un) \wedge \mathit{eq}_{oi}(s', un) \wedge \\ \mathit{card}_{io}(s', cs') \wedge \mathit{lt}_{ii}(cs', k) \wedge \\ \mathit{card}_{io}(s, cs) \wedge \mathit{lt}_{ii}(cs, k) \end{array}$$

3. $\mathit{union} :: s, sv \Rightarrow un$, $\mathit{eq} :: s', un \Rightarrow$:
   The resulting mode is: $s, k, v, s' \Rightarrow cs, sv, un, cs'$. One possible moded formula is:

   $$\begin{array}{l} \mathit{card}_{io}(s, cs) \wedge \mathit{lt}_{ii}(cs, k) \wedge \\ \mathit{makeset}_{io}(v, sv) \wedge \mathit{union}_{iio}(s, sv, un) \wedge \mathit{eq}_{ii}(s', un) \wedge \\ \mathit{card}_{io}(s', cs') \wedge \mathit{lt}_{ii}(cs', k) \end{array}$$

4. $\mathit{union} :: sv, un \Rightarrow s$, $\mathit{eq} :: s' \Rightarrow un$:
   The resulting mode is: $k, v, s' \Rightarrow un, sv, s, cs, cs'$. One possible moded formula is:

   $$\begin{array}{l} \mathit{card}_{io}(s', cs') \wedge \mathit{lt}_{ii}(cs', k) \wedge \\ \mathit{eq}_{io}(s', un) \wedge \mathit{makeset}_{io}(v, sv) \wedge \mathit{union}_{oii}(s, sv, un) \wedge \\ \mathit{card}_{io}(s, cs) \wedge \mathit{lt}_{ii}(cs, k) \end{array}$$

5. $union :: sv, un \Rightarrow s$, $eq :: un \Rightarrow s'$:
   The resulting mode is: $k, v, un \Rightarrow s', sv, s, cs, cs'$. One possible moded formula is:

$$eq_{oi}(s', un) \wedge$$
$$card_{io}(s', cs') \wedge lt_{ii}(cs', k) \wedge$$
$$makeset_{io}(v, sv) \wedge union_{oii}(s, sv, un) \wedge$$
$$card_{io}(s, cs) \wedge lt_{ii}(cs, k)$$

6. $union :: sv, un \Rightarrow s$, $eq :: s', un \Rightarrow$:
   The resulting mode is: $k, v, un, s' \Rightarrow sv, s, cs, cs'$. One possible moded formula is:

$$card_{io}(s', cs') \wedge lt_{ii}(cs', k) \wedge$$
$$eq_{ii}(s', un) \wedge makeset_{io}(v, sv) \wedge union_{oii}(s, sv, un) \wedge$$
$$card_{io}(s, cs) \wedge lt_{ii}(cs, k)$$

We assume that the primitives have the following requirements, constraints and number of solutions[10]:

Operation	Mode	Requirements	Solutions	Constraints
card	$1 \Rightarrow 2$	$1_{0-\mathbb{F}}$	$1 \Leftrightarrow 1$	2_1
lt	$1, 2 \Rightarrow$	$1_{0-1}, 2_{0-1}$	$0 \Leftrightarrow 1$	
makeset	$1 \Rightarrow 2$		$1 \Leftrightarrow 1$	$1_{0-\mathbb{F}} \Rightarrow 2_{0-\mathbb{F}}$
union	$1, 2 \Rightarrow 3$	$1_{0-\mathbb{F}}, 2_{0-\mathbb{F}}$	$1 \Leftrightarrow 1$	$3_{0-\mathbb{F}}$
	$1, 3 \Rightarrow 2$	$1_{0-\mathbb{F}}, 3_{0-\mathbb{F}}$	$0 \Leftrightarrow \mathbb{F}$	$2_{0-\mathbb{F}}$
	$2, 3 \Rightarrow 1$	$2_{0-\mathbb{F}}, 3_{0-\mathbb{F}}$	$0 \Leftrightarrow \mathbb{F}$	$1_{0-\mathbb{F}}$
eq	$1 \Rightarrow 2$		$1 \Leftrightarrow 1$	F
	$2 \Rightarrow 1$		$1 \Leftrightarrow 1$	F
	$1, 2 \Rightarrow$	$1_{0-\mathbb{F}}, 2_{0-\mathbb{F}}$	$0 \Leftrightarrow 1$	F

where F is the formula:

$$(1_0 \Leftrightarrow 2_0) \wedge (1_1 \Leftrightarrow 2_1) \wedge (1_{\mathbb{F}} \Leftrightarrow 2_{\mathbb{F}}) \wedge (1_\infty \Leftrightarrow 2_\infty)$$

Let us now consider subtype analysis. Observe that none of the primitive operations used have any requirements on output. Thus, the analysis of the primitive operations simply returns the relevant information from the table above. This being the case, *staconj* will not modify the formula and so we elide the relevant arguments in the presentation below.

The following table summarises the analysis results for the atomic subformulae.

[10] We write $1_{0-\mathbb{F}}$ to indicate that the first variable is required to be non-infinite.

i	F_i	R_i	S_i	C_i
1	$makeset_{io}(v, sv)$	$\Leftrightarrow$	$1 \Leftrightarrow 1$	$v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}}$
2	$union_{iio}(s, sv, un)$	$s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}$	$1 \Leftrightarrow 1$	$un_{0-\mathbb{F}}$
3	$eq_{oi}(s', un)$	$\Leftrightarrow$	$1 \Leftrightarrow 1$	F
4	$card_{io}(s', cs')$	$s'_{o-\mathbb{F}}$	$1 \Leftrightarrow 1$	cs'_1
5	$lt_{ii}(cs', k)$	cs'_{0-1}, k_{0-1}	$0 \Leftrightarrow 1$	$\Leftrightarrow$
6	$card_{io}(s, cs)$	$s_{o-\mathbb{F}}$	$1 \Leftrightarrow 1$	cs_1
7	$lt_{ii}(cs, k)$	cs_{0-1}, k_{0-1}	$0 \Leftrightarrow 1$	$\Leftrightarrow$

$sta(F_1 \wedge \ldots \wedge F_7) = (R^I, S, C, add\ R^{\setminus I}\ [F_1, \ldots, F_7])$
$\quad$ *where* $(R, S, C) = staconj(F_1 \wedge \ldots \wedge F_7, true, \varnothing, 1 \Leftrightarrow 1, true)$

$staconj(F_1 \wedge \ldots \wedge F_7, true, \varnothing, 1 \Leftrightarrow 1, true)$
$= staconj(F_2 \wedge \ldots \wedge F_7, C_1 \wedge \ulcorner R_1 \urcorner \wedge true, R_1^* \sqcap \varnothing, S_1 \otimes 1 \Leftrightarrow 1, C_1 \wedge true)$
$[R_1 = \varnothing \Rightarrow R_1^* = \varnothing]$
$= staconj(F_2 \wedge \ldots \wedge F_7, v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}}, \varnothing, 1 \Leftrightarrow 1, v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}})$

$= staconj(F_3 \wedge \ldots \wedge F_7, C_2 \wedge \ulcorner R_2 \urcorner \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}}, R_2^* \sqcap \varnothing, 1 \Leftrightarrow 1 \otimes 1 \Leftrightarrow 1,$
$\quad un_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}})$
$[R_2^* = R_2]$
$= staconj(F_3 \wedge \ldots \wedge F_7, K_2, \{s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}, 1 \Leftrightarrow 1, un_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}})$
$\quad$ *where* $K_2 = un_{0-\mathbb{F}} \wedge s_{0-\mathbb{F}} \wedge sv_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}}$

$= staconj(F_4 \wedge \ldots \wedge F_7, C_3 \wedge \ulcorner R_3 \urcorner \wedge K_2, R_3^* \sqcap \{s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}, 1 \Leftrightarrow 1 \otimes 1 \Leftrightarrow 1,$
$\quad C_3 \wedge un_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}})$
$[R_3 = \varnothing \Rightarrow R_3^* = \varnothing]$
$= staconj(F_4 \wedge \ldots \wedge F_7, K_3, \{s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}, 1 \Leftrightarrow 1, F \wedge un_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}})$
$\quad$ *where* $K_3 = F \wedge un_{0-\mathbb{F}} \wedge s_{0-\mathbb{F}} \wedge sv_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}}$

$= staconj(F_5 \wedge F_6 \wedge F_7, C_4 \wedge \ulcorner R_4 \urcorner \wedge K_3, R_4^* \sqcap \{s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}, 1 \Leftrightarrow 1 \otimes 1 \Leftrightarrow 1,$
$\quad C_4 \wedge F \wedge un_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}})$
[Since $K_3 \models s'_{0-\mathbb{F}}$ we have that $R_4^* = \varnothing$]
$= staconj(F_5 \wedge F_6 \wedge F_7, cs'_1 \wedge s'_{0-\mathbb{F}} \wedge K_3, \{s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}, 1 \Leftrightarrow 1, cs'_1 \wedge F \wedge un_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}})$

$= staconj(F_6 \wedge F_7, C_5 \wedge \ulcorner R_5 \urcorner \wedge cs'_1 \wedge s'_{0-\mathbb{F}} \wedge K_3, R_5^* \sqcap \{s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}, 0 \Leftrightarrow 1 \otimes 1 \Leftrightarrow 1,$
$\quad C_5 \wedge cs'_1 \wedge F \wedge un_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}})$
[Since $cs'_1 \in K$ we have $R_5^* = \{k_{0-1}\}$]
$= staconj(F_6 \wedge F_7, K_5, \{k_{0-1}, s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}, 0 \Leftrightarrow 1, cs'_1 \wedge F \wedge un_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}})$
$\quad$ *where* $K_5 = k_{0-1} \wedge cs'_1 \wedge s'_{0-\mathbb{F}} \wedge F \wedge un_{0-\mathbb{F}} \wedge s_{0-\mathbb{F}} \wedge sv_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}}$

$= staconj(F_7, C_6 \wedge \ulcorner R_6 \urcorner \wedge K_5, R_6^* \sqcap \{k_{0-1}, s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}, 1 \Leftrightarrow 1 \otimes 0 \Leftrightarrow 1,$
$\quad C_6 \wedge cs_1' \wedge F \wedge un_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}})$
$[R_6^* = \varnothing]$
$= staconj(F_7, cs_1 \wedge K_5, \{k_{0-1}, s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}, 0 \Leftrightarrow 1, cs_1 \wedge cs_1' \wedge F \wedge un_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}})$

$= staconj(true, C_7 \wedge \ulcorner R_7 \urcorner \wedge cs_1 \wedge K_5, R_7^* \sqcap \{k_{0-1}, s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}, 0 \Leftrightarrow 1 \otimes 0 \Leftrightarrow 1,$
$\quad C_7 \wedge cs_1 \wedge cs_1' \wedge F \wedge un_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}})$
[Since $\{cs_1, k_{0-1}\} \subset K$ we have $R_7^* = \varnothing$]
$= staconj(true, cs_{0-1} \wedge k_{0-1} \wedge cs_1 \wedge K_5, \{k_{0-1}, s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}, 0 \Leftrightarrow 1,$
$\quad cs_1 \wedge cs_1' \wedge F \wedge un_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}})$
$= (\{k_{0-1}, s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}, 0 \Leftrightarrow 1, cs_1 \wedge cs_1' \wedge F \wedge un_{0-\mathbb{F}} \wedge v_{0-\mathbb{F}} \Rightarrow sv_{0-\mathbb{F}})$

$R^I = \{k_{0-1}, s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}^{s,k,v} = \{k_{0-1}, s_{0-\mathbb{F}}\}$
$R^{\setminus I} = \{k_{0-1}, s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}^{\setminus s,k,v} = \{sv_{0-\mathbb{F}}\}$

$add\ \{sv_{0-\mathbb{F}}\}\ [F_1, \ldots, F_7]$
$= add\ \varnothing (add'\ sv\ \ulcorner sv_{0-\mathbb{F}} \urcorner\ [F_1, \ldots, F_7])$
$= add'\ sv\ \ulcorner sv_{0-\mathbb{F}} \urcorner\ [F_1, \ldots, F_7]$
[Since $F_1 :: v \Rightarrow sv$]
$= [F_1, \ulcorner sv_{0-\mathbb{F}} \urcorner, F_2, \ldots, F_7]$

Hence the formula is:

$makeset_{io}(v, sv) \wedge \ulcorner sv_{0-\mathbb{F}} \urcorner \wedge union_{iio}(s, sv, un) \wedge eq_{oi}(s', un) \wedge$
$card_{io}(s', cs') \wedge lt_{ii}(cs', k) \wedge card_{io}(s, cs) \wedge lt_{ii}(cs, k)$

It has either one or zero solutions and requires that k is of size $0 \Leftrightarrow 1$ (ie. that it is an atom– in this case a number), and that s is finite.

The table below summarises the results of the analysis for the other modes. Below we summarise the inserted tests for each formula.

No.	Mode	Requirements	Solutions
1	N / A	N / A	N / A
2	$s, k, v \Rightarrow cs, sv, un, s', cs'$	$\{k_{0-1}, s_{0-\mathbb{F}}, sv_{0-\mathbb{F}}\}$	$0 \Leftrightarrow 1$
3	$s, k, v, s' \Rightarrow cs, sv, un, cs'$	$\{s'_{0-\mathbb{F}}, k_{0-1}, s_{0-\mathbb{F}}\}$	$0 \Leftrightarrow 1$
4	$k, v, s' \Rightarrow un, sv, s, cs, cs'$	$\{s'_{0-\mathbb{F}}, k_{0-1}\}$	$0 \Leftrightarrow \mathbb{F}$
5	$k, v, un \Rightarrow s', sv, s, cs, cs'$	$\{un_{0-\mathbb{F}}, k_{0-1}\}$	$0 \Leftrightarrow \mathbb{F}$
6	$k, v, un, s' \Rightarrow sv, s, cs, cs'$	$\{s'_{0-\mathbb{F}}, k_{0-1}, un_{0-\mathbb{F}}\}$	$0 \Leftrightarrow \mathbb{F}$

3. $card_{io}(s', cs') \wedge lt_{ii}(cs', k) \wedge makeset_{io}(v, sv) \wedge \ulcorner sv_{0-\mathbb{F}} \urcorner \ldots$
4. $card_{io}(s', cs') \wedge lt_{ii}(cs', k) \wedge eq_{io}(s', un) \wedge makeset_{io}(v, sv) \wedge \ulcorner sv_{0-\mathbb{F}} \urcorner \ldots$
5. $eq_{oi}(s', un) \wedge card_{io}(s', cs') \wedge lt_{ii}(cs', k) \wedge makeset_{io}(v, sv) \wedge \ulcorner sv_{0-\mathbb{F}} \urcorner \ldots$
6. $card_{io}(s', cs') \wedge lt_{ii}(cs', k) \wedge eq_{ii}(s', un) \wedge makeset_{io}(v, sv) \wedge \ulcorner sv_{0-\mathbb{F}} \urcorner \ldots$

6 Conclusions and Further Work

We have presented two analyses which are used to determine information needed to render a Z specification executable. The analyses operate on a Horn clause translation of a Z specification. The first analysis determines possible modes for formulae. The second analysis eliminates (some) redundant subtype checks.

The analyses have been implemented and the author and others are in the process of integrating the analyses with the (separate) translator from Z to Horn clauses and evaluating the animation on various Z specifications.

The mode analysis has scope for improvement. Two ideas that deserve investigation are *aspect based modes* and *preferential modes*.

Standard mode systems focus on the bindings of variables or parts of variables (for example, if X denotes a pair then a mode might indicate that the first element of the pair is bound but that the second isn't). It would be useful to generalise this notion of a part of a variable to include *aspects* which are not strictly a part of a binding. For example, we may know that a variable X representing a sequence, contains a certain set of elements but may not know the ordering of the elements. In this case, no *part* of X is determined, but an *aspect* of X is known and it is possible to automatically generate possible bindings for X by considering permutations of its elements.

When considering possible modes for atoms we would like to be able to execute operations in as many modes as possible. On the other hand, some of these modes will be more efficient than others. For example, given the formula $X = Y \land Y \in Z$ in mode $X, Z \Rightarrow Y$ we could either use the moding $X = Y :: X \Rightarrow Y \land Y \in Z :: Y, Z \Rightarrow$ or the moding $Y \in Z :: Z \Rightarrow Y \land X = Y :: X, Y \Rightarrow$. Clearly, the former moding is preferable since it generates Y directly and then tests for membership in Z, whereas the latter moding generates multiple bindings for Y and tests each one. In certain situations (for example, if Z is defined by a set comprehension yielding an infinite set) the former may terminate whereas the latter may not. Ideally, we would like to be able to specify that the mode $X = Y :: X \Rightarrow Y$ is "better than", or *preferable* to the mode $X = Y :: X, Y \Rightarrow$.

An important piece of further work is to formally prove that the analyses presented are correct. This is an important part of the overall proof that the animation tool is correct.

References

[1] S. M. Brien and J. E. Nicholls. Z base standard. Technical Monograph PRG-107, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, November 1992. Accepted for standardization under ISO / IEC JTC1 / SC22.

[2] Krasner H. Curtis, B. and Neil Iscoe. A field study of the software design process for large systems. *Communications of The ACM*, 31(11):1268–1287, 1988.

[3] Philip W. Dart. On derived dependencies and connected databases. *Journal of Logic Programming*, 11:163–188, 1991.

[4] Philip W. Dart and Justin Zobel. Efficient run-time checking of typed logic programs. *Journal of Logic Programming*, 14(1&2):31–69, 1992.

[5] Robyn R. Lutz. Targeting safet-related errors during software requirements analysis. In *Proceedings of SIGSOFT '93, Foundations of Software Engineering*, pages 99–105, 1993.

[6] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[7] Lee Naish. A declarative view of modes. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 185–199. MIT Press, September 1996.

[8] Philip Sallis, Graham Tate, and Stephen McDonell. *Software Engineering*. Addison Wesley Publishing Co., 1995.

[9] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, February 1995.

[10] Zoltan Somogyi, Fergus Henderson, Thomas Conway, Andrew Bromage, Tyson Dowd, David Jeffery, Peter Ross, Peter Schachte, and Simon Taylor. Status of the Mercury system. In *Proceedings of the JICSLP'96 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pages 207–218, 1996.

[11] Zoltan Somogyi, Fergus Henderson, Thomas Conway, and Richard O'Keefe. Logic programming for the real world. In Donald A. Smith, editor, *Proceedings of the ILPS'95 Postconference Workshop on Visions for the Future of Logic Programming*, pages 83–94, Portland, Oregon, 1995.

[12] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

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

[14] Michael Winikoff and Philip Dart. Translating Z to logic. Technical Report 97/14, Department of Computer Science, Melbourne University, 1997.

A Derivation of the subtype analysis rule for $\Leftrightarrow$

We expand $F \Leftrightarrow G$ as $(F \wedge G) \vee (\neg F \wedge \neg G)$. We associate subscripts to the results of analysing subformulae as follows:

- R_1 . . . - F
- R_2 . . . - G
- R_3 . . . - $\neg F$
- R_4 . . . - $\neg G$
- R_5 . . . - $F \wedge G$
- R_6 . . . - $\neg F \wedge \neg G$
- R . . . - $(F \wedge G) \vee (\neg F \wedge \neg G)$

Using the rule for $\neg$ we have that

$$R_3 = R_1, C_3 = \mathit{true}, S_3 = \widetilde{S_1}, F_3 = \neg F_1$$

$$R_4 = R_2, C_4 = \mathit{true}, S_4 = \widetilde{S_2}, F_4 = \neg F_2$$

Using the rule for $\wedge$ we have that

$$S_5 = S_1 \otimes S_2, C_5 = C_1 \wedge C_2$$

$$R_5 = R_1 \sqcap T_1^I$$

$$F_5 = \bigwedge(\mathit{add}\ R_5^{\setminus I}\ [F_1, F_2])$$

Since neither F_1 nor F_2 can bind variables we have that $R_5^{\setminus I} = \varnothing$ and hence $F_5 = F_1 \wedge F_2$.

Using the rule for $\wedge$ again we can derive that

$$S_6 = S_3 \otimes S_4 = \widetilde{S_1} \otimes \widetilde{S_2}$$

$$C_6 = C_3 \wedge C_4 = \mathit{true} \wedge \mathit{true} = \mathit{true}$$

$$R_6 = R_3 \sqcap T_2^I = R_1 \sqcap T_2^I$$

$$F_6 = F_3 \wedge F_4 = \neg F_1 \wedge \neg F_2$$

$$\begin{aligned} T_1 &= \{t \in R_2 \mid (C_1, \ulcorner R_1 \urcorner \nVdash t)\} \\ T_2 &= \{t \in R_4 \mid (C_3, \ulcorner R_3 \urcorner \nVdash t)\} \\ &= \{t \in R_2 \mid (\mathit{true}, \ulcorner R_1 \urcorner \nVdash t)\} \\ &= \{t \in R_2 \mid \ulcorner R_1 \urcorner \nVdash t\} \end{aligned}$$

T_1 is the requirements in R_2 which do not follow from C_1, R_1 and T_2 is the requirements in R_2 which do not follow from R_1. Since it is safe to increment the requirements of a formula we can increase T_1 to T_2. If this is done then $R_6 = R_1 \sqcap T_2^I = R_5$ and hence, in the $\vee$ rule,

$$R_5 \sqcup R_6 \;=\; R_6 \;=\; R_5$$

and furthermore,

$$R_5 \boxminus R_5 \;=\; R_6 \boxminus R_6 \;= \mathit{true}.$$

Finally, we can use the rule for $\vee$ to construct the final result:

$$S = S_6 \oplus S_5 = (S_1 \otimes S_2) \oplus (\widetilde{S_1} \otimes \widetilde{S_2})$$

$$C = C_5 \vee C_6 = C_5 \vee \mathit{true} = \mathit{true}$$

$$F = F_5 \vee F_6 = \ldots = F_1 \Leftrightarrow F_2$$

$$R = R_5 \sqcup R_6 = R_5 = R_1 \sqcap T_2^I$$

Observe that $R_1 \sqcap T_2^I \;=\; R_1 \sqcap R_2$ since T_2^I is just the parts of R_2 which can not be derived from R_1. Hence

$$R = \ldots = R_1 \sqcap R_2$$

Collecting, we have that the result of analysing $F \Leftrightarrow G$ is

$$S = S_6 \oplus S_5 = (S_1 \otimes S_2) \oplus (\widetilde{S_1} \otimes \widetilde{S_2})$$

$$C = \mathit{true}, F' = F_1 \Leftrightarrow F_2, R = R_1 \sqcap R_2$$

B A brief overview of lattice theory

A binary relation $\mathcal{R}$ is

- *Reflexive* iff $\forall\, x\ x\mathcal{R}x$
- *Transitive* iff $\forall\, x \forall\, y \forall\, z\ ((x\mathcal{R}y) \wedge (y\mathcal{R}z)) \Rightarrow (x\mathcal{R}z)$
- *Anti-Symmetric* iff $\forall\, x \forall\, y\ ((x\mathcal{R}y) \wedge (y\mathcal{R}x)) \Rightarrow (x = y)$

A *preorder* is a binary relation which is reflexive and transitive.
A *partial order* is a preorder which is also antisymmetric.
Given a domain X and a subset of the domain $Y \subseteq X$.

- $x \in X$ is an *upper bound* for Y iff $\forall\, y \in Y\ (y\mathcal{R}x)$
- $x \in X$ is a *lower bound* for Y iff $\forall\, y \in Y\ (x\mathcal{R}y)$

$x \in X$ is a *least* upper bound for Y iff x is an upper bound and all upper bounds are greater or equal to x.
Notation:

$$x = lub\ Y \text{ or } x = \bigsqcup Y$$

We also use a binary version of this operator written

$$x \vee y \text{ or } x\ join\ y$$

Dually x is a *greatest* lower bound iff it is a lower bound for Y and all lower bounds are less than or equal to it.
Notation:

$$x = glb\ Y \text{ or } x = \bigsqcap Y$$

We also use a binary version of this operator written

$$x \wedge y \text{ or } x\ meet\ y$$

Y is a *chain* iff $\forall\, y, y' \in Y\ (y\mathcal{R}y') \vee (y'\mathcal{R}y)$.
A set X with a partial order defined on it (a poset) is a *lattice* if $\bigsqcup Y$ and $\bigsqcap Y$ exist for all *finite* subsets $Y \subseteq X$.
A set X with a partial order defined on it (a poset) is a *complete* lattice if $\bigsqcup Y$ and $\bigsqcap Y$ exist for *all* subsets $Y \subseteq X$.
We define

$$\top = \bigsqcup X = \bigsqcap \varnothing \text{ and } \bot = \bigsqcap X = \bigsqcup \varnothing$$

Let X and Y be complete lattices. Let $f :: X \to Y$ and $g :: X \to X$ be functions.
Then f is *monotonic* iff

$$\forall\, x, x'\ (x\mathcal{R}x') \Rightarrow (f(x)\ \mathcal{R}\ f(x'))$$

f is *continuous* iff

$$f(\bigsqcup Z) = \bigsqcup\{f(z) \mid z \in Z\}$$

for all non-empty chains Z.
If f is continuous then it is also monotonic.
An element $x \in X$ is a *fixpoint* of g iff $g(x) = x$.
An element $x \in X$ is the *least* fixpoint of g if it is a fixpoint of g and all fixpoints are greater than or equal to it.
If g is a function operating over complete lattices and g is monotonic then the *least fixpoint* of g is well defined.
If $g :: X \to X$ is a *continuous* function over a complete lattice then its least fixpoint is well defined and is the limit of the sequence

$$\bot,\ (g\bot),\ (g^2\bot),\ (g^3\bot)\ldots$$