The Development of a Tableau Machine

Tony Clark
Department of Computing
University of Bradford
March 4, 1999
Incomplete Draft

1 Introduction

Software development usually involves describing what a program will do in terms of a *specification* and then describing how a program will achieve the specified behaviour in terms of a *design* and ultimately an *implementation*. An aim of Software Engineering is to develop techniques and tools which support the process of software construction from its specification to its implementation.

There are a number of competing strategies for achieving this aim, one of which is to view the specification as an *infeasible* program which must be incrementally modified in order to produce the required software. An infeasible program typically differs from an acceptable or *feasible* program in that it consumes too many resources, but often has the advantage that it is easier to understand.

This paper makes a contribution to the development of software from infeasible programs. In particular it identifies a novel technique which can be used to analyse an infeasible program in order to translate it into a more acceptable program. The technique is explained by means of an example from the area of mathematical logic: the construction of a *tableau* for a first order predicate calculus formula.

The construction of a tableau can be defined using an infeasible program. A feasible program for constructing a tableau is defined in terms of an abstract machine. The machine is defined by analysing the calculations which are performed by the infeasible program and using the analysis to define a collection of equivalent machine calculations. It is then possible to define a machine by "reading off" the necessary transitions from the required machine calculations. The approach is described by the following diagram:

$$\begin{tikzcd}
\begin{array}{c}\textit{infeasible}\\ \textit{program}\end{array} \arrow[r] \arrow[d] & \begin{array}{c}\textit{feasible}\\ \textit{program}\end{array} \\
\begin{array}{c}\textit{infeasible}\\ \textit{calculations}\end{array} \arrow[r] & \begin{array}{c}\textit{feasible}\\ \textit{calculations}\end{array} \arrow[u]
\end{tikzcd}$$

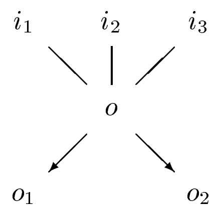
where the translation of an infeasible program to a feasible program is induced by a corresponding translation of the infeasible calculations.

The structure of this paper is as follows. Section 2 describes the technique which is used in the rest of the paper. Section 3 gives a brief overview of the tableau method from mathematical logic. Section 4 describes a simple infeasible program which constructs a tableau for a first order predicate calculus formula. Section **??** defines the calculations which are performed by the infeasible program. Section **??** analyses the infeasible calculations in order to identify the features which will be required of a consistent machine calculation. Section 6 associates each infeasible calculation with an equivalent machine calculation. Section 7 defines a machine by analysing the necessary steps in the machine calculations. Section 8 argues that the machine is correct with respect to the infeasible program and section 9 analyses the work.

2 Calculations

A *calculation*[1] is a representation of the execution, or part-execution, of a program. A calculation is a graph where the nodes are labelled with data items and the edges are labelled with operations between data items. Each edge may have zero, one or many source nodes corresponding to the number of inputs to the operation. Each edge may have zero, one or more target nodes corresponding to the number of distinct outputs produced by the operation. A *data item* is to be interpreted in the usual sense with respect to programs. An *operation* is to be interpreted as either a built-in programming language feature such as + or *log*, or as a calculation.

A calculation is conveniently drawn on a sheet of paper where the inputs to the calculation are at the top and the outputs are at the bottom. A single edge may be drawn as a "squashed spider" where the legs lead from and to the inputs and outputs and where the body is the operation. The following diagram shows an operation o which has three inputs and two outputs:



We are interested in two specific types of calculations which will be referred to as *binary* and *unary* calculations respectively. A binary calculation contains edges with at most two source or destination nodes. A unary calculation contains

[1]The term *calculation* and description given here is due to Peter Landin.

edges with at most one source or destination node. A special type of unary calculation where the edges are all labelled with the same operation will be called a *machine calculation.*

A graph which is a unary calculation will be represented as a term:

$$unary(x_1, o, x_2)$$

consisting of two nodes labelled with data items x_1 and x_2 and an edge between them labelled with operation o. The input to the unary calculation is x_1 and the output from the unary calculation is x_2.

A graph which is a binary calculation will be represented as a term:

$$binary(x_1, o_1, b_1, b_2, o_2, x_2)$$

which is to be viewed as a graph of the following general form:

$$\begin{tikzcd}
 & x_1 \arrow[d, dash] & \\
 & o_1 \arrow[dl] \arrow[dr] & \\
b_1 \arrow[dr, dash] & & b_2 \arrow[dl, dash] \\
 & o_2 \arrow[d] & \\
 & x_2 &
\end{tikzcd}$$

The binary calculation consists of an input x_1, an output x_2, an operation o_1 which splits the input into two data items, an operation which combines two data items to produce a single data item, and two sub-calculations b_1 and b_2. The input to calculation b_1 is one of the outputs from o_1 and the input to calculation b_2 is the other output from o_1. Correspondingly, the inputs to operation o_2 are provided by the outputs from the two sub-calculations.

Each calculation has a collection of inputs and outputs. Most of the calculations which we will analyse will have a single input and a single output. The inputs of a calculation b are the labels on nodes of b which are the target of no edges: $\downarrow b$. Similarly, the outputs of a calculation b are the labels on nodes of b which are the source of no edges: $\uparrow b$. When either the inputs or outputs are singleton sets we will identify the set with its element by an abuse of notation.

Calculations represent specific examples of program execution and as such can be used as a semantics for a language. This paper applies calculations to the area of program behavioural equivalence and in particular to the issue of translating a non-tail recursive program into one that is tail recursive. In order to motivate the issues raised in the rest of the paper, this section concludes with a simple example.

Consider a language N containing phrases which are either an integer n or the addition of two phrases $e_1 + e_2$. A phrase from this language may be evaluated using the non-tail recursive program *eval*:

$$\begin{aligned}&\mathit{eval}(n) = n\\ &\mathit{eval}(e_1 + e_2) = \mathit{eval}(e_1) + \mathit{eval}(e_2)\end{aligned}$$

In order to translate an N-phrase into an integer, the program *eval* performs a calculation. These calculations are represented as follows:

- $\mathit{const}(n) = \mathit{unary}(n, \mathit{const}, n)$ which is a calculation with two nodes and a single edge. The source of the edge is a node labelled with the program phrase n. The target of the edge is a node labelled with the integer n. The edge is labelled with a "constant" operation.

- $\mathit{add}(b_1, b_2) = \mathit{binary}(\downarrow b_1 + \downarrow b_2, +\text{-}\mathit{split}, b_1, b_2, \uparrow b_1 + \uparrow b_2)$ is a calculation which describes the decomposition of an addition phrase, followed by the execution of both sub-phrases, and finally the addition of two integers which are the outputs from the two sub-phrases.

Consider a calculation $\mathit{add}(b_1, b_2)$ which has been performed by *eval*. No assumptions are made regarding *how* this was performed, *i.e.* the calculation strategy may have performed b_1 and then b_2, vice versa, a mixture of the two, or b_1 may have been performed concurrently with b_2.

We wish to translate the binary calculations performed by *eval* to a machine calculation. We will then use the machine calculations to define a machine which performs them. Since a machine calculation is also unary, we must place an ordering on all calculations. A machine calculation must also exhibit all variations by changing the contents of data items at the nodes in a calculation since the labels on the edges cannot change.

Each binary calculation is associated with a machine calculation in a particular context. The context describes pending outputs, inputs and operators. For example, consider the calculation $\mathit{add}(b_1, b_2)$, a machine calculation must place an ordering on the execution of the two sub-calculations:

1. while the first sub-calculation is performed, the input to the second sub-calculation must be stored in the context until it can be used.

2. while the second sub-calculation is performed, the output from the first sub-calculation must be stored in the context until it can be used.

3. while both sub-calculations are performed, the operator which combines the two outputs must be stored in the context.

Binary calculations nest properly and so the context of each sub-calculation will nest within the context of its super-calculation. This allows contexts to be implemented as stacks. A machine calculation for the example language will have two stacks: s, an *output stack*, will be used for outputs and c, a *control stack*, will be used for both inputs and operators. A machine state is a pair (s, c) and a machine calculation is a sequence of machine states.

Each binary calculation b is associated with an equivalent machine calculation using a relation:

$$b, s, c \Rightarrow m, s'$$

where b is a binary calculation, s and s' are output stacks, c is a control stack, and m is a machine calculation. The intended interpretation is that b and m are equivalent calculations in the initial context containing s and c and producing a final context s' and c. The relation is defined using an axiom and a rule.

$$\mathit{const}(n), s, c \Rightarrow [(s, n : c), (n : s, c)], n : s \tag{1}$$

Axiom 1 defines that a constant calculation in a context s and c produces a constant machine calculation which contains the context information.

$$\frac{\begin{array}{c} b_1, s, \downarrow b_2 : (+) : c \Rightarrow m_1, s' \\ b_2, s', (+) : c \Rightarrow m_2, \_ : \_ : s'' \end{array}}{\mathit{add}(b_1, b_2), s, c \Rightarrow [(s, \downarrow b_1 + \downarrow b_2)] +\!\!+\, m_1 +\!\!+\, m_2 +\!\!+ [(\uparrow b_1 + \uparrow b_2 : s'', c)]} \tag{2}$$

Rule 2 defines that an addition calculation on a machine performs one sub-calculation and then the other. Notice that the input and output of the machine calculation are taken from the binary calculation and are therefore consistent with it.

It is now possible to define an addition machine by “reading off” the necessary transitions. The machine is defined as follows:

$$(s, n : c) \mapsto (n : s, c) \tag{3}$$

$$(s, e_1 + e_2 : c) \mapsto (s, e_1 : e_2 : (+) : c) \tag{4}$$

$$(n_2 : n_1 : s, (+) : c) \mapsto (n_1 + n_2 : s, c) \tag{5}$$

Transition 3 produces the calculations which are required by axiom 1. Transitions 4 and 5 produce the “book-end” calculations which are required by the rule 2. Transition 4 sets up the correct context by placing the addition operator onto the control stack.

The machine calculations which are performed by $\mapsto$ are simply the sequence of machine states. If e is an N-phrase and m is the corresponding machine calculation then $e \mapsto^* m$. In order that the machine is correct with respect to the evaluation of N-phrases as defined by the program *eval* the following condition must hold:

$$e \mapsto^* m \Leftrightarrow \mathit{eval}(e), [], [] \Rightarrow m, s$$

The proof is by rigorous argument as follows:

1. If the expression e is a single integer n then the machine calculations defined by 3 and 1 are trivially equivalent in any given context s and c.

2. If the expression e is an addition $e_1 + e_2$ then the machine calculation defined by 4 and 5 in a context s and c is:

   $$[(s, e_1 + e_2 : c)] \mathbin{+\!\!+} m_1 \mathbin{+\!\!+} m_2 \mathbin{+\!\!+} [(n_1 + n_2 : s, c)]$$

   By induction we assume that m_1 and m_2 are equivalent to the calculations produced by 1 and 2. Therefore, by rule 2, the following is true:

   $$eval(e), [], [] \Rightarrow m, s \rightarrow e \mapsto^* m$$

   The proof in the opposite direction is achieved by similar argument.

Therefore we conclude that the machine defines the same semantics for arithmetic expressions as the program *eval*.

3 Semantic tableaux

This section describes a technique which constructs a semantic tableau for first order predicate calculus formulas. The process of tableau construction will be used in the rest of the paper as an example of how an infeasible program can be transformed into a feasible program by analysing its calculations.

A *first order predicate calculus formula*, or just a *formula*, is one of the following: *true*, *false*, $p(\tilde{v})$, $\neg q$, $q_1 \wedge q_2$, $q_1 \vee q_2$, $\forall x \bullet q$, and $\exists x \bullet q$ where x is an arbitrary variable, $\tilde{v}$ is an ordered collection of variables and constants, q, q_1 and q_2 are arbitrary formulas, and p is a predicate name. An ordered collection of constants, or *individuals*, $\tilde{v}$ will be viewed as an unordered set of individuals where appropriate. In order to simplify the analysis of formulas, we make use of the standard equivalences and deal only with negated formulas of the form $\neg p(\tilde{v})$.

An *atomic formula* is $p(\tilde{v})$ or $\neg p(\tilde{v})$. Two atomic formulas are *contradictory* when they are the same except one is negated ($\neg$) and the other is not. An atomic formula is *ground* when it contains no variables.

An *interpretation* for a closed set of formulas is a mapping between predicate names, p, and sets of ordered collections of individuals. The standard model based semantics for a first order formula uses an interpretation to associate the formula with one of two values denoting truth and falsity.

A formula is *valid* if it is true in every interpretation and it is *unsatisfiable* if it is false in every interpretation. The tableau method can be used to determine whether a group of sentences are unsatisfiable. The method takes a formula and develops a disjunctive set of conjunctive sets of ground atomic formulas. The sets interpret each formula in all possible ways so that it is true. Each of the sets is inspected to see if it contains contradictory atomic formulas. If all of the sets contain contradictory atomic formulas then the original formula was unsatisfiable. The tableau method is often used to show the validity of a formula $q_1 \Rightarrow q_2$ by showing the unsatisfiability of the formula $q_1 \wedge \neg q_2$.

A tableau for a formula can be constructed by a human using a step-by-step process. We start with a binary tree containing formulas at the nodes. Each node may be marked or unmarked. When a node is marked, the formula at that node is no longer available. Initially all the nodes are unmarked. An unmarked node is selected at random and the formula at the node is inspected. The action which is performed depends upon the type of the formula:

1. if the formula is *true* then the node is marked.
2. if the formula is marked then the node is marked.
3. if the formula is $p(\tilde{v})$ then the node is marked.
4. if the formula is $\neg p(\tilde{v})$ then the node is marked.
5. if the formula is $q_1 \wedge q_2$ then the node is marked. All tree branches below the node are extended by adding a new branch ending in an unmarked node containing q_1. Then all branches are further extended by a new branch ending in an unmarked node containing q_2.
6. if the formula is $q_1 \vee q_2$ then the node is marked. All branches below the node are extended by adding a pair of new branches. The first branch ends in an unmarked node containing q_1 and the second branch ends in an unmarked node containing q_2.
7. if the formula is $\forall x \bullet q$ then the tree is inspected for all the individuals which have been encountered so far. For each individual v_k encountered so far, a new formula q_k is constructed by replacing x in q by v_k. All branches of the tree below the node are extended with a new branch ending in an unmarked node containing the formula $q_1 \wedge q_2 \wedge \ldots \wedge q_n$.
8. if the formula is $\exists x \bullet q$ then the node is marked. All branches below the node are extended with a new branch which ends with an unmarked node containing the formula q', where q' is the same as q except that x has been replaced by some arbitrary new individual.

The process repeats until there are no unmarked nodes except for universal formulas each of which has been instantiated with respect to all the explicitly referenced individuals in the tree.

Each path from the root of the tree to a leaf is a set of atomic formulas. If all sets contain contradictory formulas then we conclude that no model exists which makes the initial collection of formulas true. This method is often used in the following manner. Suppose that we wish to show that $q_1 \Rightarrow q_2$. We construct an initial tree which contains $q_1 \wedge \neg q_2$. If, after applying the tableau process, we find that there is no possible interpretation for $q_1 \wedge \neg q_2$ then we are forced to conclude that the opposite must be true, *i.e.* $\neg q_1 \vee q_2$ which is equivalent to $q_1 \Rightarrow q_2$.

4 Tableaux construction

A tableau is represented as follows:

1. a ground atomic formula p is contained in the tableau $<p>$.
2. two tableaux t_1 and t_2 can be joined together to form the conjunction $t_1 \,\&\, t_2$.
3. two tableaux t_1 and t_2 can be joined together to form the disjunction $t_1 \mid t_2$.

A tableau which is constructed using $<\_>$, $\&$ and $\mid$ is a tree of nodes labelled with formulas as described in the previous section. Given an expression in terms of these operators it is straightforward to draw the corresponding tree.

In order to define an infeasible program which constructs a tableau for a given formula we define the following:

- $\backslash(\oplus)(f)(x)(\{v_1, v_2, \ldots, v_n\}) = x \oplus f(v_1) \oplus f(v_2) \oplus \ldots \oplus f(v_n)$ where $\oplus$ is a binary operator, f is a unary operator, x and v_i are all data items of appropriate types.
- it is possible to substitute a value for a variable in a formula and in a calculation. This is represented as $[v_1/x]q$ and $[v_2/x]c$ where q is a formula and c is a calculation and v_i are data items of the appropriate types. Substitution into a formula has the usual meaning and substitution into a calculation is defined in terms of substitution into a formula. The expression $[\_/x]y$ is a function which substitutes its argument into the formula/calculation y.

Suppose that, given a formula q we know the set of individuals which will be used in constructing a tableau for q. Let this set of individuals be V. The program *tableau* defines how a tableau is constructed:

$$
\begin{array}{l}
tableau(true) = <true> \\
tableau(false) = <false> \\
tableau(p(\tilde{v})) = <p(\tilde{v})> \\
tableau(\neg p(\tilde{v})) = <\neg p(\tilde{v})> \\
tableau(p_1 \wedge p_2) = (tableau(p_1)) \,\&\, (tableau(p_2)) \\
tableau(p_1 \vee p_2) = (tableau(p_1)) \mid (tableau(p_2)) \\
tableau(\forall x \bullet p) = (<\forall x \bullet p>) \,\&\, (tableau(\backslash(\wedge)([\_/x]p)(true)(V))) \\
tableau(\exists x \bullet p) = (<\exists x \bullet p>) \,\&\, (tableau([v/x]p)) \text{ where } v \text{ is new}
\end{array}
$$

In this context an individual v is *new* when it is not referenced explicitly in the rest of the formula and is not used in constructing the tableau for any other existential formula.

5 Specification

A binary calculation type $\mathcal{B}$ is a pair: $< S_1, S_2 >$ where S_1 is a collection of data item sorts and S_2 is a set of edge labels. The type $\mathcal{B}$ is viewed as a set containing all the possible calculations freely constructed from the data item sorts and the edge labels, with the proviso that the source and target nodes of edges must be labeled consistently with respect to the edge label.

The sorts in S_1 describe the data items which occur as node labels in $\mathcal{B}$-calculations. The labels in S_2 are one of the following: operations whose signature is constructed from S_1; calculations whose input and output data types are all in S_1; tuples containing labels.

The binary calculations which are performed by *tableau* are not in a suitable format for flattening into machine calculations. This is because we require the unary calculation to start with no knowledge of individuals whereas *tableau* starts with coplete knowledge of all individuals.

The translation of *tableau* calculations into machine calculations can be done in two stages. The first stage is to translate the infeasible binary calculations performed by *tableau* into feasible binary calculations. The second stage is to flatten the feasible binary calculations into a machine calculations.

This section specifies the first stage of *tableau* calculation translation. To do this, we set up a binary calculation type and restrict it so that it describes just the calculations performed by *tableau*. This calculation type is used as the basis of modifications and restrictions leading to a semantically equivalent feasible calculation type. The result of flattening these feasible calculations will produce machine calculations which are equivalent to the binary calculations performed by *tableau*.

The starting point is to define a binary calculation type for *tableau*:

$$\mathcal{Q}_1 =< \{Q, T\}, \{<\_>, \wedge\text{-}split, \&\text{-}join, \vee\text{-}split, |\text{-}join\} + Q_1 >$$

The type Q_1 is restricted so that the operations label edges whose source and target are labelled with appropriate data items. For example, $\wedge$-*split* may only label an edge whose source is a formula $q_1 \wedge q_2$ and which has exactly two targets labeled with q_1 and q_2 respectively.

Q_1 is also restricted so that the only edges labelled with calculations are those whose source node is labelled with a quantified formula. The label must be the calculation which is performed by *tableau* when supplied with the quantified formula.

A tableau transformation calculation is binary and is an element of the following type:

$$\mathcal{T}_1 = < \{T\}, \{\&\text{-}split, \&\text{-}join, |\text{-}split, |\text{-}join, id\} >$$

The type $\mathcal{T}$ is restricted so that the operations used as edge labels are used appropriately. The only operation which may be applied to atomic tableaux of the form $<q>$ is *id*; consequently, the type $\mathcal{T}$ defines identity transformation calculations.

A list transformation calculation is binary and is an element of the following type:

$$\mathcal{L}(\alpha) = < \{[\alpha], \{:\text{-}split, :\text{-}join, [], id\} >$$

The type $\mathcal{L}(\alpha)$ is parameterised with respect to the type of the components of the list α. It is respected so that the operations are used as labels on appropriate edges and therefore $\mathcal{L}$ denotes identity calculations.

The type $\mathcal{Q}_1$ is extended with an extra label component on each edge. The new component is a list transformation calculation where the list element type is $\mathcal{T}_1$, *i.e.* transformations of lists of tableau transformation calculations. This extension is required because some machine operations will force us to backtrack and modify tableaux are intermediate outcomes. The extended type is defined below:

$$\mathcal{Q}_2 = < \{Q, T\}, (\{<\_>, \wedge\text{-}split, \&\text{-}join, \vee\text{-}split, |\text{-}join\} + \mathcal{Q}_2) \times \mathcal{L}(\mathcal{T}_1) >$$

Each label on a $\mathcal{Q}_2$ edge is a pair whose second component is a calculation which produces a sequence of tableaux. The extra tableaux are combined with the original output from the first component using &.

6 Machine calculations

This section associates a machine calculation with each binary calculation performed by the program *tableau*. In doing so, we will translate infeasible non-tail recursive calculations into feasible tail recursive calculations with the intention of then "reading off" the machine which performs them.

Each binary calculation is associated with a machine calculation using three relations which have the following general form:

$$x, s, c, v \Rightarrow m, s', v'$$

where the type of x defines which of the three relations we are dealing with. The calculation x may be: binary; stack; or tableau. In each case s and s' are stack calculations, c is a control item stack, v and v' are sets of individuals and m is a machine calculation. The intended interpretation is that the calculation x is equivalent to the calculation m in the context s, c and v producing a context s', c and v'.

The stack calculation s serves two related purposes:

1. the input to the calculation s is the stack component for the current machine context.
2. s is the current rescan calculation.

In each case c and v are the control item and individual contexts for the machine calculation respectively. The rest of this section defines the three relations. Each relation is defined using a collection of axioms and rules with respect to the inductive structure of the calculations.

6.1 Binary calculations

A binary calculation b is associated with a machine calculation m in the context s, c and v producing a new context s', c and v' when the following relationship holds:

$$b, s, c, v \Rightarrow m, s', v'$$

the rest of this section defines the relationship between binary calculations and machine calculations.

$$\mathit{true}, s, c, v \Rightarrow [(s, \mathit{true} : c, v), (<\mathit{true}> : s, c, v)], \mathit{cons}(\mathit{just}(\mathit{true}), s), v \tag{6}$$

$$\mathit{false}, s, c, v \Rightarrow [(s, \mathit{false} : c, v), (<\mathit{false}> : s, c, v)], \mathit{cons}(\mathit{just}(\mathit{false}), s), v \tag{7}$$

Axioms 6 and 7 associate constant binary calculations which construct tableau for *true* and *false* with the equivalent machine calculations. Note that as the constant k is pushed on to the machine stack there is an equivalent constant calculation pushed on to the stack calculation:

$$\mathit{cons}(\mathit{just}(k), s)$$

which means that when the stack calculation is performed, the stack will be decomposed, calculations $\mathit{just}(k)$ and s will be performed, and then the stack will be re-composed (possibly having performed some outstanding calculations).

$$\frac{\begin{array}{c} b_1, s, \downarrow b_2 : (\wedge) : c, v \Rightarrow m_1, s', v' \\ b_2, s', (\wedge) : c, v' \Rightarrow m_2, \mathit{cons}(t_1, \mathit{cons}(t_2, s'')), v'' \end{array}}{\begin{array}{l} \mathit{and}(b_1, b_2), s, c, v \Rightarrow \\ \quad [(\downarrow s, \downarrow b_1 \wedge \downarrow b_2 : c, v)] +\!\!+ \\ \qquad m_1 +\!\!+ m_2 +\!\!+ \\ \qquad\quad [(\downarrow t_1 \;\&\; \downarrow t_2 : \downarrow s'', c, v'')], \\ \qquad\qquad \mathit{cons}(\mathit{both}(t_1, t_2), s''), v'' \end{array}} \tag{8}$$

$$\frac{\begin{array}{c} b_1, s, \downarrow b_2 : (\vee) : c, v \Rightarrow m_1, s', v' \\ b_2, s', (\vee) : c, v' \Rightarrow m_2, cons(t_1, cons(t_2, s'')), v'' \end{array}}{\begin{array}{l} or(b_1, b_2), s, c, v \Rightarrow \\ \quad [(\downarrow s, \downarrow b_1 \vee \downarrow b_2 : c, v)] +\!\!+ \\ \quad\quad m_1 +\!\!+ m_2 +\!\!+ \\ \quad\quad\quad [(\downarrow t_1 \mid \downarrow t_2 :\downarrow s'', c, v'')], \\ \quad\quad\quad\quad cons(either(t_1, t_2), s''), v'' \end{array}} \tag{9}$$

Rules 8 and 9 associate machine calculations with binary calculations arising from conjunction and disjunction formulas. Essentially these two rules are the same except for the various constructors used to produce formulas and calculations. Consider a calculation $and(b_1, b_2)$. We perform b_1 first which produces an equivalent machine calculation m_1. The control context for m_1 includes the input to b_2 and the combination operator $\wedge$. The context which is current after m_1 is performed is s' and v' which are used as the initial context for the calculation m_2 which is equivalent to the second binary calculation b_2. The final context for m_2 is described by a stack calculation:

$$cons(t_1, cons(t_2, s''))$$

and a set of individuals v''. The stack calculation contains two tableau calculations t_1 and t_2 whose inputs are the tableaux produced by calculations m_1 and m_2 respectively. To produce the complete machine calculation we add the "book-end" machine states:

$$[(s, \downarrow c1 \wedge \downarrow b_2 : c, v)]$$

which is the starting state constructed from the context and the inputs to the two binary calculations; and:

$$[\downarrow t_1 \,\&\, \downarrow t_2 :\downarrow s'', c, v'')]$$

where the data item stack is constructed from the final context of machine calculation m_2. Together, these calculations produce the machine calculation which is equivalent to $and(b_1, b_2)$:

$$[(s, \downarrow b_1 \wedge \downarrow b_2 : c, v)] +\!\!+ m_1 +\!\!+ m_2 +\!\!+ [(\downarrow t_1 \,\&\, \downarrow t_2 :\downarrow s'', c, v'')]$$

$$\frac{\backslash(and)([\_/x]q)(true)(v), s, c, v \Rightarrow m, cons(t, s'), v'}{\begin{array}{l} forall(x, q, \backslash(and)([\_/x]q)(true)(V)), s, c, v \Rightarrow \\ \quad [(\downarrow s, \forall x \bullet q : c, v)] +\!\!+ \\ \quad\quad m +\!\!+ [((<\forall x \bullet q>, v') \,\&\, \downarrow t :\downarrow s', c, v')], \\ \quad\quad\quad cons(both(forall(x, q, \backslash(and)([\_/x]q)(true)(V - v')), t), s'), v' \end{array}} \tag{10}$$

Rule 10 defines how a binary calculation $forall(x, q, b)$ is associated with an equivalent machine calculation. The binary calculation b will have been constructed using the full set of individuals required for the tableau. At this point in the machine calculation we know from the context information that only the set v of individuals is known. If we represent the calculation b as:

$$\backslash(\mathit{and})([\_/x]q)(\mathit{true})(V)$$

then we may only produce a machine calculation which is equivalent to:

$$\backslash(\mathit{and})([\_/x]q)(\mathit{true})(v)$$

The rule defines that this is the case and shows how the stack calculation which is produced contains the outstanding binary calculation:

$$\backslash(\mathit{and})([\_/x]q)(\mathit{true})(V-v)$$

This is the key reason why we cannot represent the data item stack in the context as a value. We must represent it as a calculation since it contains outstanding calculations which must be performed before the tableau is completed.

$$\frac{\begin{array}{c} s, \mathit{nil}, (\exists x \bullet q, i) : c, v \cup \{i\} \Rightarrow m_1, \mathit{cons}(s', \mathit{nil}), v' \\ b, \mathit{cons}(\mathit{just}(\exists x \bullet q), s'), (\wedge) : c, v' \Rightarrow m_2, \mathit{cons}(t, s''), v'' \end{array}}{\begin{array}{l} \mathit{exists}(x, q, i, b), s, c, v \Rightarrow \\ \quad [(\downarrow\!s, \exists x \bullet q : c, v)] +\!\!+ \\ \quad\quad m_1 +\!\!+ m_2 +\!\!+ \\ \quad\quad\quad [((<\exists x \bullet q> \,\&\, \downarrow\!t) :\downarrow\!s'', c, v'')], \\ \quad\quad\quad\quad \mathit{cons}(\mathit{both}(\mathit{just}(\exists x \bullet q), t), s''), v'' \end{array}} \tag{11}$$

Rule 11 defines how a binary calculation $\mathit{exists}(x, q, i, b)$ is associated with a machine calculation. The calculation introduces a new individual i which requires the machine calculation to do a re-scan which is m_1. Note that during the re-scan, the formula is retained at the head of the control stack as $(\exists x \bullet q, i)$. On completing the re-scan, the calculation b is scanned to produce the equivalent machine calculation m_2. The machine calculation is completed by concatenating the two sub-calculations and placing the "book-end" machine states: $[(\downarrow\!s, \exists x \bullet q : c, v)]$ and $[((<\exists x \bullet q> \,\&\, \downarrow\!t) :\downarrow\!s'', c, v'')]$.

$$\frac{s, \mathit{nil}, <p(\tilde{v})> : c, v \cup \tilde{v} \Rightarrow m, \mathit{cons}(s', \mathit{nil}), v'}{\begin{array}{l} \mathit{pred}(p, \tilde{v}), s, c, v \Rightarrow \\ \quad [(\downarrow\!s, p(\tilde{v}) : c, v)] +\!\!+ \\ \quad\quad m +\!\!+ [<p(\tilde{v})> :\downarrow\!s', c, v')], \\ \quad\quad\quad \mathit{cons}(\mathit{just}(p(\tilde{v})), s'), v' \end{array}} \tag{12}$$

$$\frac{s, \mathit{nil}, <\neg p(\tilde{v})> : c, v \cup \tilde{v} \Rightarrow m, \mathit{cons}(s', \mathit{nil}), v'}{\begin{array}{l} \mathit{not}(p, \tilde{v}), s, c, v \Rightarrow \\ \quad [(\downarrow\!s, \neg p(\tilde{v}) : c, v)] +\!\!+ \\ \quad\quad m +\!\!+ [(<\neg p(\tilde{v})> :\downarrow\!s', c, v')], \\ \quad\quad\quad \mathit{cons}(\mathit{just}(\neg p(\tilde{v})), s'), v' \end{array}} \tag{13}$$

Rule 12 defines how a machine calculation is associated with a binary calculation $\mathit{pred}(p, \tilde{v})$. Such a calculation is important because it may introduce new individuals in $\tilde{v}$ and initiates a re-scan of the binary calculation in order to perform any outstanding calculations. The re-scan is performed by associating a machine calculation with the current stack calculation s; this is defined in section 6.2. Rule 13 differs from rule 12 only in its use of $\neg$.

6.2 Stack calculations

A stack calculation s is associated with a machine calculation m in a context s', c and v producing a new context s'', c, v' when the following relationship holds:

$$s, s', c, v \Rightarrow m, s'', v'$$

Note the occurrence of three stack calculations s, s' and s''. The two stack calculations s' and s'' represent re-scan calculations occurring before and after the machine calculation m is performed. The rest of this section defines the relationship between stack calculations and machine calculations.

$$\mathit{nil}, s, c, v \Rightarrow, [(\downarrow s, [] : c, v), ([] : s, c, v)], \mathit{cons}(\mathit{nil}, s), v \tag{14}$$

Axiom 14 defines how a *nil* stack calculation is performed on a machine. The machine simply transfers the empty sequence from the control to the data item stack.

$$\frac{\begin{array}{c} x, s, \downarrow s_1 : (:) : c, v \Rightarrow m_1, s', v' \\ s_1, s', (:) : c, v' \Rightarrow m_2, \mathit{cons}(\mathit{cons}(x', s'_1), s''), v'' \end{array}}{\begin{array}{l} \mathit{cons}(x, s_1), s, c, v \Rightarrow \\ \quad [(\downarrow s, (\downarrow x :\downarrow s_1) : c, v)] +\!\!+ \\ \qquad m_1 +\!\!+ m_2 +\!\!+ \\ \qquad\quad [((\downarrow x' :\downarrow s'_1) :\downarrow s'', c, v'')], \\ \qquad\qquad \mathit{cons}(\mathit{cons}(x', s'), s''), v'' \end{array}} \tag{15}$$

Rule 15 defines how a calculation $\mathit{cons}(c_1, c_2)$ is performed on a machine. The machine must perform the calculation which produced the top of the stack, then perform the machine which produced the tail of the stack and then compose the two outcomes together.

6.3 Tableau calculations

A tableau calculation t is associated with an equivalent machine calculation m in the context s, c, v producing a new context s', c, v' when the following relationship holds:

$$t, s, c, v \Rightarrow m, s', v'$$

The rest of this section defines this relationship.

$$\mathit{just}(q), s, c, v \Rightarrow [(\downarrow s, <q> : c, v), (<q> :\downarrow s, c, v)], \mathit{cons}(\mathit{just}(q), s), v \tag{16}$$

Axiom 16 defines how the calculation which constructs a tableau containing an atomic formula q is performed on a machine.

$$\frac{\begin{array}{c} b_1, s, \downarrow b_2 : (\&) : c, v \Rightarrow m_1, s', v' \\ b_2, s', (\&) : c, v' \Rightarrow m_2, \mathit{cons}(t_2, \mathit{cons}(t_1, s'')), v'' \end{array}}{\begin{array}{l} \mathit{both}(b_1, b_2), s, c, v \Rightarrow \\ \quad [(\downarrow s, \downarrow b_1 \,\&\, \downarrow b_2 : c, v)] +\!\!+ \\ \qquad m_1 +\!\!+ m_2 +\!\!+ \\ \qquad\quad [(\downarrow t_1 \,\&\, \downarrow t_2 : \downarrow s'', c, v'')], \\ \qquad\qquad \mathit{cons}(\mathit{both}(t_1, t_2), s''), v'' \end{array}} \tag{17}$$

$$\frac{\begin{array}{c} b_1, s, \downarrow b_2 : (|) : c, v \Rightarrow m_1, s', v' \\ b_2, s', (|) : c, v' \Rightarrow m_2, \mathit{cons}(t_2, \mathit{cons}(t_1, s'')), v'' \end{array}}{\begin{array}{l} \mathit{either}(b_1, b_2), s, c, v \Rightarrow \\ \quad [(\downarrow s, \downarrow b_1 \mid \downarrow b_2 : c, v)] +\!\!+ \\ \qquad m_1 +\!\!+ m_2 +\!\!+ \\ \qquad\quad [(\downarrow t_1 \mid \downarrow t_2 : \downarrow s'', c, v'')], \\ \qquad\qquad \mathit{cons}(\mathit{either}(t_1, t_2), s''), v'' \end{array}} \tag{18}$$

Rules 17 and 18 define how tableau calculations which involve & and | are associated with equivalent machine calculations.

7 A tableaux machine

A tableau machine can be defined by "reading off" the transitions which are necessary to produce a machine calculation which is equivalent to the binary calculations which are produced by the program *tableau*.

The transitions are determined by the axioms and rules which are defined in the previous section. In general, an axiom will produce a single transition and a rule will produce a number of transitions. Often, a rule produces two transitions, the first of which decomposes a composite formula and the second composes two tableaux.

The rest of this section defines a tableau machine. A collection of related transitions is defined and then briefly analysed.

$$(s, \mathit{true} : c, v) \mapsto (<\!\mathit{true}\!> : s, c, v) \tag{19}$$

$$(s, \mathit{false} : c, v) \mapsto (<\!\mathit{false}\!> : s, c, v) \tag{20}$$

Transitions 19 and 20 produce the calculations which are defined in axioms 6 and 7.

$$(s, p(\tilde{v}) : c, v) \mapsto ([], s : \mathit{pop} : <\!p(\tilde{v})\!> : c, v \cup \tilde{v}) \tag{21}$$

$$(s, \neg p(\tilde{v}) : c, v) \mapsto ([], s : \mathit{pop} : <\!\neg p(\tilde{v})\!> : c, v \cup \tilde{v}) \tag{22}$$

$$([s], \mathit{pop} : c, v) \mapsto (s, c, v) \tag{23}$$

$$(s, <q> : c, v) \mapsto (<q> : s, c, v) \tag{24}$$

Transitions 21 – 24 produce calculations which are defined in rules 12 and 13. Transitions 21 and 22 define the initial steps in calculations which start with formulas $p(\tilde{v})$ and $\neg p(\tilde{v})$ respectively. In both cases, the stack is pushed onto the control so that a re-scan will occur. Transition 23 defines the step necessary when the re-scan is complete. Finally, transition 24 is necessary to produce a tableau containing the formula $p(\tilde{v})$ or $\neg p(\tilde{v})$.

$$(s, p \wedge q : c, v) \mapsto (s, p : q : (\wedge) : c, v) \tag{25}$$

$$(t_2 : t_1 : s, (\wedge) : c, v) \mapsto (t_1 \,\&\, t_2 : s, c, v) \tag{26}$$

$$(s, p \vee q : c, v) \mapsto (s, p : q : (\vee) : c, v) \tag{27}$$

$$(t_2 : t_1 : s, (\vee) : c, v) \mapsto (t_1 \,|\, t_2 : s, c, v) \tag{28}$$

Transitions 25 – 28 define the steps necessary to perform the calculations defined in rules 8 and 9. Transitions 25 and 27 define the steps which decompose a formula at the head of the control stack. The formulas will be translated to tableaux at the head of the data item stack. Transitions 26 and 28 define the steps which occur when the tableaux are combined.

$$(s, \forall x \bullet q : c, v) \mapsto (s, \backslash(\wedge)([\_/x]q)(<\forall x \bullet q>, v)(v) : c, v) \tag{29}$$

$$(s, (<\forall x \bullet q>, v') : c, v) \mapsto \begin{cases} ((<\forall x \bullet q>, v') : s, c, v) \text{ when } v = v' \\ (s, \backslash(\wedge)([\_/x]q)(<\forall x \bullet q>, v)(v - v') : c, v) \end{cases} \tag{30}$$

Transitions 29 and 30 define the steps which are necessary to perform the calculations defined in rule 10. Notice that transition 29 constructs a data item $(<\forall x \bullet q>, v)$ which is equivalent to an outstanding calculation, it records that the universal formula has been instantiated with respect to the individuals v.

$$(s, \exists x \bullet q : c, v) \mapsto ([], s : pop : (\exists x \bullet q, i) : c, v \cup \{i\}) \tag{31}$$

$$(s, (\exists x \bullet q, i) : c, v) \mapsto (<\exists x \bullet q> : s, [i/x]q : (\wedge) : c, v) \tag{32}$$

Transitions 31 and 32 define the steps necessary to perform the calculations defined in 11.

$$(s, [] : c, v) \mapsto ([] : s, c, v) \tag{33}$$

$$(s, (x : s') : c, v) \mapsto (s, x : s' : (:) : c, v) \tag{34}$$

$$(s' : x : s, (:) : c, v) \mapsto ((x : s') : s, c, v) \tag{35}$$

$$(s, t_1 \,\&\, t_2 : c, v) \mapsto (s, t_1 : t_2 : (\&) : c, v) \tag{36}$$

$$(t_2 : t_1 : s, (\&) : c, v) \mapsto (t_1 \,\&\, t_2 : s, c, v) \tag{37}$$

$$(s, t_1 \,|\, t_2 : c, v) \mapsto (s, t_1 : t_2 : (|) : c, v) \tag{38}$$

$$(t_2 : t_1 : s, (|) : c, v) \mapsto (t_1 \,|\, t_2 : c, v) \tag{39}$$

Transitions 33 – 39 define the steps which are necessary to perform re-scan calculations. These correspond to the rules 14 – 18.

8 Correctness

This section is to be updated with a proof of the correctness of the tableau machine by a rigorous argument.

The tableau machine described in this paper has been implemented in the programming language Scheme. The rules associating binary calculations with machine calculations have been implemented in the programming language Prolog.

9 Analysis of method

There are number of distinct approaches to the construction of programs. For example, the refinement technique initially specifies a program in terms of its preconditions and postconditions and then successively transforms the specification by weakening the preconditions and strengthening the postconditions. An alternative approach starts with a complete but infeasible program and applies successive semantics preserving transformations which increase the efficiency of the program.

The technique which is described in this paper views programs in terms of the required calculations. A specification is a description of the required calculations and a constraint describing the calculations which are possible. The required calculations are then associated with feasible calculations which are used to define a feasible program.

This approach is novel and has the following advantages over other approaches:

1. no particular execution strategy is enforced by the approach. By describing the ideal calculations which are required, no assumptions are being made with respect to the particular implementation strategy which will eventually be used.

2. the approach does not rely upon a collection of laws which are provided for program transformation. Laws may be defined which prove the correctness of a program with respect to a particular collection of calculations.

3. the approach exhibits the intuitive appeal of allowing specifications to be constructed by analysing fragments of desired calculations.

4. the approach allows the freedom to define program development strategies which are entirely semantics based, *i.e.* without the need for a syntax in order to describe what a program does.

In particular, the application described in this paper benefits from the approach since it provides a consistent mechanism for representing all aspects of *tableau* calculations, including complex issues such as nested re-scans. Although it has not been attempted here, the approach allows different methods of tableau construction to be developed from the program *tableau*; all methods will be consistent, however some may be more desirable than others. The approach allows the efficiency of programs to be analysed since it deals explicitly with abstract calculations.

The approach is universally applicable and may be used in conjunction with other approaches to software specification and development. For example, the operations which are used as node labels can be defined in terms of logical formulas.