Constructive interval temporal logic in Alf

Simon Thompson

Computing Laboratory
University of Kent at Canterbury
S.J.Thompson@ukc.ac.uk

Abstract. This paper gives an implementation of an interval temporal logic in a constructive type theory, using the Alf proof system. After explaining the constructive approach, its relevance to interval temporal logic and potential applications of our work, we explain the fundamentals of the Alf system. We then present the implementation of the logic and give a number of examples of its use. We conclude by exploring how the work can be extended in the future.

1 Introduction

The traditional approach to executing temporal logics is to execute the formulas of the logic; this is in accord with the logic programming paradigm. The implementation can be deterministic for particular subclasses of formulas, as in the approach taken by Moszkowski and others [8, 6, 4]. On the other hand, all the formulas of a logic can be executed using a backtracking mechanism; this is shown by Gabbay's normal form result in [5] and is implemented in the various MetateM systems [1] amongst others.

There is another paradigm for implementing logics, based on a *constructive* philosophy [7, 11]. Instead of formulas being seen as true or false on the basis of truth tables or model theory, a constructive approach takes proof as the means of exhibiting validity; as a slogan, one might say that constructive logic is 'proof functional' rather than 'truth functional'. A constructive proof of a proposition contains more information than a classical version, so that from a proof we can derive all the evidence for the proposition being valid. For instance, a proof of an existential statement will contain a *witness* which is an object for which the statement holds.

Under the constructive approach, then, we execute not the formulas of the logic but their *proofs*, which we can see from the discussion above contain sufficient witnessing information to be executable. Further details of the basics of constructive logics, and their interpretation as programming languages can be found in Section 2.

The system in which we make our implementation is Alf, which comes from the programming logics group at Chalmers University of Technology in Göteborg, Sweden. We give a short introduction to Alf in Section 3.

The logic we look at here is an interval temporal logic, which describes finite intervals. Because of this, besides containing the familiar temporal operators $\Box$,

$\Diamond$ and so on, interval temporal logics also contain predicates which can only apply to finite intervals, such as those which measure length or which compose two propositions in sequence (or 'chop'). Introductions to the logic are to be found in [8, 4, 2] and we refer readers there for further details. One distinctive aspect of our logic is that it involves atomic *actions* which happen at the instants of an interval.

The approach we examine here can equally well be used to give a constructive account of an infinitary linear (or branching time) logic. Details of an implementation in the Coq system are to be found elsewhere [10].

We see three strengths of the work reported here.

- We provide a single system in which we can model both specifications and their implementations. Specifications can be related by logical inference, and are shown to be consistent by exhibiting an implementation; inconsistent specifications will simply have no implementation.
- We maintain two levels of abstraction in our system. In specifications we can use operators such as $\Diamond$ and 'chop' which can be realised in many ways; we can think of them as non-deterministic. In our implementations or proofs we have determinism. For example, a constructive proof of a formula $\Diamond P$ will show not only that P holds at some point in the future but also will state at precisely which point in the future P holds. This distinction is entirely appropriate to the modelling applications of interval temporal logics.
- An implementation of a logic such as this forces an implementer to check the coherence of his or her definition of the logic. In our work here we see a distinction between the notions of interval and interval proposition which in an informal account may be elided. We also have to maintain a distinction between an action A, say, and the proposition that 'A happens (now)'.
  In our related work on linear-time temporal logics, [10], the issue of whether the logic is anchored or not has to be addressed at an early stage in writing the implementation.

I am grateful to Erik Poll both for supplying an implementation of basic logic for modification and for making a number of useful comments on drafts of the paper. I would also like to thank Howard Bowman, Helen Cameron and Peter King for their collaboration in the *Mexitl* [2] work. It was this which stimulated the investigation reported here.

2 Constructive logic

What counts as a constructive proof of a formula? An informal explanation is given in Figure 1.

That this gives the logic a distinctive character should not be in question; while truth functionally one would accept $A \vee \neg A$ for any A, it is by no means clear that for an arbitrary formula one can find either a proof of A or a proof that A is contradictory. On the other hand we can see that proofs are much more informative than in the classical case. A proof of a disjunction must be a proof

$A \wedge B$	A proof of $A \wedge B$ consists of a proof of A and a proof of B.
$A \vee B$	A proof of $A \vee B$ consists either of a proof of A or of a proof of B.
$A \rightarrow B$	A proof of $A \rightarrow B$ is a method (or function) taking proofs of A into proofs of B.
$(\exists x \in A)B(x)$	A proof of $(\exists x \in A)B(x)$ consists of an element a of A together with a proof of $B(a)$.
$(\forall x \in A)B(x)$	A proof of $(\forall x \in A)B(x)$, which we also write $(x \in A) \rightarrow B$, consists of a function taking x in A to a proof of $B(x)$.

Fig. 1. Proof in constructive logic

of one of the disjuncts, and a proof of an existential statement must provide a witness which is a point where the statement holds, together with a proof that it does indeed hold at that point.

How does a constructive implementation work? We take the formulas of a logic as specifications of behaviour; it is then the *proofs* of these formulas that are implemented. Underlying this is an important correspondence, attributed to Curry and Howard and illustrated in Figure 2, which identifies a constructive logic and a (functional) programming language.

Constructive Logic		**Programming Language**
Formula		Type
Proof		Value
Conjunction	$\wedge$	Product or record type
Disjunction	$\vee$	Sum or union type
Implication	$\rightarrow$	Function space
Existential quantification	$\exists$	'Dependent' record type
Universal quantification	$\forall$	'Dependent' function type

Fig. 2. The Curry-Howard correspondence

Under the Curry-Howard correspondence the formulas of a logic are seen as the types of an expressive type system which includes not only record, union and function types but also dependent function types

$$(x \in A) \rightarrow B$$

in which B can depend upon x, so that the type of a function application can depend upon the *value* to which the function is applied. These types correspond to universally quantified formulas, while a dependent record type represents an existentially quantified statement — we shall see this in Section 3.1.

Given this explanation we can now see how our implementation is built. The formulas of our interval logic become the types of functions which implement the specifications that the formulas embody.

3 Introducing Alf

The logic used here is a standard formulation of a constructive logic in Alf. As we explained in Section 2 we can view Alf as a functional programming language with a strong type system. It is for this reason that we chose to use Alf here rather than, say, Coq; in Coq the proof terms are implicit rather than explicit, and we wanted to emphasise these functions in our account.

We explain the basics of the system by means of a sequence of examples

3.1 Basic constructive logic in Alf

Built into the system is the type

$$Set \in Type$$

which is the type of sets or alternatively the type of propositions. Types are defined in Alf by inductive definitions, these are a strengthening of the algebraic data types of standard functional languages such as Haskell [9]. We first define a trivially true proposition *True* by giving it a single element, **trivial**. Constructors of types are given in **boldface**; here we see that **trivial** is a constant, that is a 0-ary constructor.

$$\left[\begin{array}{l} True \in Set \\ \quad = \mathbf{data}\ \{\mathbf{trivial}\ (\ )\} \end{array}\right.$$

Thinking set theoretically, *True* is a one element set. A *False* proposition is a proposition with no proof, or an empty type, which has no constructors:

$$\left[\begin{array}{l} False \in Set \\ \quad = \mathbf{data}\ <\ > \end{array}\right.$$

The angled brackets here indicate that the type has *no* constructors, and so is indeed empty. Next, we have a definition of conjunction:

$$\left[\begin{array}{l} And(P, Q \in Set) \in Set \\ \quad = \mathbf{data}\ \left\{\mathbf{conj}\ \begin{pmatrix} p \in P \\ q \in Q \end{pmatrix}\right\} \end{array}\right.$$

This definition of a **data** type states that to construct an element of $And\ P\ Q$ it is necessary to use the single constructor **conj**. This requires two arguments to construct an element of the conjunction, namely elements p of P and q of Q.

In other words, it is necessary to supply proofs of both conjuncts to give a proof of the conjunction. We also have a definition of a constructive disjunction

$$\left[\begin{array}{l} Or(P, Q \in Set) \in Set \\ \quad = \textbf{data} \left\{ \begin{array}{l} \textbf{inl}\ (p \in P) \\ \textbf{inr}\ (q \in Q) \end{array} \right\} \end{array}\right.$$

To supply an element of $Or\ P\ Q$ we need either to give an element p of P, making $\textbf{inl}\ p \in (Or\ P\ Q)$, or to give an element q of Q, so that $\textbf{inr}\ q \in (Or\ P\ Q)$. This is evidently a constructive disjunction, since a proof of $Or\ P\ Q$ is a proof of one of the disjuncts; the first disjunct if it is of the form $\textbf{inl}\ p$ and the second disjunct otherwise. As we implied earlier, this explanation is quite different from a classical interpretation, and so the law of the excluded middle, $(Or\ A\ (Not\ A))$ is not valid in general.

The existential quantifier is also constructive:

$$\left[\begin{array}{l} Exists(A \in Set,\ \ P \in (x \in A) \rightarrow Set) \in Set \\ \quad = \textbf{sig} \left\{ \begin{array}{l} witness \in A \\ proof \in P\ witness \end{array} \right\} \end{array}\right.$$

We can think of this type as giving a **sig**nature; the elements of the type are **struct**ures taking the form

$$\textbf{struct} \left\{ \begin{array}{l} witness = \ldots \\ proof = \ldots \end{array} \right\}$$

thus containing a *witness* of the point at which the predicate P holds together with a *proof* that the predicate holds at the *witness*, that is an element of $P\ witness$. Note that we use a *dependent* type here: the type of the second element: $P\ witness$ depends on the first element, *witness*.

The syntax of Alf allows quantifiers to be written in a more readable form, with

$$Exists\ x \in A\ .\ ....x....$$

replacing

$$Exists\ A\ (\lambda\ x \rightarrow ....x....)$$

where we use $....x....$ for an expression involving x. We use this form in the remainder of the paper.

3.2 Data types

The natural numbers are given by the declaration

$$\left[\begin{array}{l} Nat \in Set \\ \quad = \textbf{data} \left\{ \begin{array}{l} \mathbf{0}\ (\,) \\ \mathbf{S}\ (n \in Nat) \end{array} \right\} \end{array}\right.$$

and the constants *zero*, *one*, *two* and so on have the obvious meaning.

In our implementation of interval temporal logic we represent intervals by non-empty lists of propositions. In order to do this we have to define a type constructor for non-empty lists, and this constructor needs to be of the appropriate *kind*: since it is used to build lists of *Set* it needs to take a *Type* to a *Type*. The constructor is called *list*, and takes a *Type* argument, making it polymorphic:

$$\left[\begin{array}{l} list \in (T \in Type) \rightarrow Type \\ \quad = \lambda\ T \rightarrow \mathbf{data} \left\{ \begin{array}{l} \mathbf{sing}\ (x \in T) \\ \mathbf{cons} \begin{pmatrix} x \in T \\ xs \in list\ T \end{pmatrix} \end{array} \right\} \end{array}\right.$$

Because the lists are non-empty, they all have a *first* and a *last* element. Here we use the **case** construction which gives case analysis (and indeed primitive recursion) over a **data** type, by means of pattern matching.

$$\left[\begin{array}{l} first \in (T \in Type,\ b \in list\ T) \rightarrow T \\ \quad = \lambda\ T\ b \rightarrow \mathbf{case}\ b\ \mathbf{of} \left\{ \begin{array}{lcl} \mathbf{sing}\ x & \rightarrow & x \\ \mathbf{cons}\ x\ xs & \rightarrow & x \end{array} \right\} \end{array}\right.$$

An arbitrary element of the type *list* T will either have the form **sing** x or **cons** $x\ xs$; the **case** construct requires us to give the value of $first$ in both these cases. We can use the variables in the particular pattern in the corresponding part of the definition.

$$\left[\begin{array}{l} last \in (T \in Type,\ b \in list\ T) \rightarrow T \\ \quad = \lambda\ T\ b \rightarrow \mathbf{case}\ b\ \mathbf{of} \left\{ \begin{array}{lcl} \mathbf{sing}\ x & \rightarrow & x \\ \mathbf{cons}\ x\ xs & \rightarrow & last^{\circ}\ xs \end{array} \right\} \end{array}\right.$$

Although the function *last* takes two arguments we suppress the first (type) argument, since it is invariably obvious from the context. The absence of one or more parameters is indicated by the superscript in $last^{\circ}$.

Before we proceed, note that in this presentation of lists we take the length of a list to be the number of elements it contains *minus one*. In particular therefore a single element list has length zero in this formulation.

The functions *take* and *drop* are used to select portions of a list. The natural number argument supplied gives an indication of the number of elements taken or dropped from the front of the list. Specifically

$$take^{\circ}\ n\ l$$

gives a list of length n (that is comprising *(n+1)* elements) from the front of l, whilst

$$drop^{\circ}\ n\ l$$

removes n elements from the front of l. The effect of this choice is that $take^{\circ}\ n\ l$ and $drop^{\circ}\ n\ l$ overlap by one element.

$$\left[\begin{array}{l} take \in (T \in Type,\ b \in Nat,\ c \in list\ T) \rightarrow list\ T \\ \quad = \lambda\ T\ b\ c \rightarrow \mathbf{case}\ b\ \mathbf{of} \left\{ \begin{array}{lcl} \mathbf{0} & \rightarrow & \mathbf{case}\ c\ \mathbf{of} \left\{ \begin{array}{lcl} \mathbf{sing}\ x & \rightarrow & \mathbf{sing}\ x \\ \mathbf{cons}\ x\ xs & \rightarrow & \mathbf{sing}\ x \end{array} \right\} \\ \mathbf{S}n & \rightarrow & \mathbf{case}\ c\ \mathbf{of} \left\{ \begin{array}{l} \mathbf{sing}\ x \ \rightarrow\ \mathbf{sing}\ x \\ \mathbf{cons}\ x\ xs\ \rightarrow \\ \qquad \mathbf{cons}\ x\ (take^{\circ}\ n\ xs) \end{array} \right\} \end{array} \right\} \end{array}\right.$$

$$\left[\begin{array}{l} drop \in (T \in Type,\ b \in Nat,\ c \in list\ T) \rightarrow list\ T \\ \quad = \lambda\ T\ b\ c \rightarrow \mathbf{case}\ b\ \mathbf{of} \left\{ \begin{array}{l} \mathbf{0}\ \rightarrow\ c \\ \mathbf{S}n\ \rightarrow\ \mathbf{case}\ c\ \mathbf{of} \left\{ \begin{array}{lll} \mathbf{sing}\ x & \rightarrow & \mathbf{sing}\ x \\ \mathbf{cons}\ x\ xs & \rightarrow & drop^{\circ}\ n\ xs \end{array} \right\} \end{array} \right\} \end{array}\right.$$

The function *index* selects an element of a list, numbering the elements from zero. If the index exceeds the number of elements in the list, the last element is returned.

$$\left[\begin{array}{l} index \in (T \in Type,\ b \in Nat,\ c \in list\ T) \rightarrow T \\ \quad = \lambda\ T\ b\ c \rightarrow \mathbf{case}\ b\ \mathbf{of} \left\{ \begin{array}{l} \mathbf{0}\ \rightarrow\ first^{\circ}\ c \\ \mathbf{S}n\ \rightarrow\ \mathbf{case}\ c\ \mathbf{of} \left\{ \begin{array}{lll} \mathbf{sing}\ x & \rightarrow & x \\ \mathbf{cons}\ x\ xs & \rightarrow & index^{\circ}\ n\ xs \end{array} \right\} \end{array} \right\} \end{array}\right.$$

3.3 Using Alf

We have used the experimental **Alfa** version of Alf, which is implemented using Haskell and the Fudgets library [3] by Thomas Hallgren. The system contains an interactive graphical editor which allows a user to build complex definitions by point and click. A particularly valuable feature is a menu of options giving possible constructions which it would be type safe to use at any point in an expression; by means of this one constructs type correct programs.

This concludes our introduction to the aspects of Alf used here; more details can be found at `http://www.cs.chalmers.se/ hallgren/Alfa/`

4 Interval Temporal Logic

In this section we begin by giving in Section 4.1 our definition of the fundamentals of the implementation, namely definitions of what it is to be an interval, an action and an interval proposition.

Central to interval logic are various connectives or combinators which allow us to combine interval propositions together to give more complex propositions. Apart from the obvious lifting of the propositional connectives and quantifiers of predicate logic, which we look at in Section 4.8, and the standard temporal operators defined in Section 4.7, we introduce two operators characteristic of an interval logic.

The first is *chop* P Q, in Section 4.3, which holds of an interval when the interval can be split into two halves satisfying P and Q separately. Secondly we introduce *proj* P Q which projects one interval, by means of P, onto another which should meet Q; projection is introduced in Section 4.5.

4.1 Actions and Intervals

We take the type of actions as given; for the sake of exposition here we assume it is a data type of constants (or 0-ary constructors):

$$\left[\begin{array}{l} Act \in Set \\ \quad = \mathbf{data} \left\{ \begin{array}{l} \mathbf{A}\ (\ ) \\ \dots \end{array} \right\} \end{array}\right.$$

There are various means of representing sets in constructive logic; here we choose to model sets of actions by 'characteristic' functions from Act to Set:

$$\left[\begin{array}{l} ActSet \in Type \\ \quad = (A \;\in\; Act) \;\rightarrow\; Set \end{array}\right.$$

An interval is a *list* of action sets.

$$\left[\begin{array}{l} Interval \in Type \\ \quad = list\ ActSet \end{array}\right.$$

and an interval proposition or $IntProp$ is a function which takes an interval to a proposition (that is a Set).

$$\left[\begin{array}{l} IntProp \in Type \\ \quad = (I \in Interval) \rightarrow Set \end{array}\right.$$

An interval is said to be empty if it contains a single point.

$$\left[\begin{array}{l} Empty \in IntProp \\ \quad = \lambda\ I \rightarrow \mathbf{case}\ I\ \mathbf{of} \left\{\begin{array}{lll} \mathbf{sing}\ x & \rightarrow & True \\ \mathbf{cons}\ x\ xs & \rightarrow & False \end{array}\right\} \end{array}\right.$$

Generalising this is a proposition $Length\ n$ expressing that the length of an interval is n: $Empty$ is then given by $Length\ 0$.

$$\left[\begin{array}{l} Length \in (a \in Nat) \rightarrow IntProp \\ \quad = \lambda\ a\ b \rightarrow \mathbf{case}\ a\ \mathbf{of} \left\{\begin{array}{lll} \mathbf{0} & \rightarrow & \mathbf{case}\ b\ \mathbf{of} \left\{\begin{array}{lll} \mathbf{sing}\ x & \rightarrow & True \\ \mathbf{cons}\ x\ xs & \rightarrow & False \end{array}\right\} \\ \mathbf{S}\ n & \rightarrow & \mathbf{case}\ b\ \mathbf{of} \left\{\begin{array}{lll} \mathbf{sing}\ x & \rightarrow & False \\ \mathbf{cons}\ x\ xs & \rightarrow & Length\ n\ xs \end{array}\right\} \end{array}\right\} \end{array}\right.$$

Our final example of an atomic proposition is 'A happens now', that is at the $first$ point of an interval

$$\left[\begin{array}{l} happensNow \in (A \in Act) \rightarrow IntProp \\ \quad = \lambda\ A\ I \rightarrow first^{\circ}\ I\ A \end{array}\right.$$

The expression $first^{\circ}\ I$ is an action set, and so the proposition that A holds is given by applying the action set to the action, giving the proposition $(first^{\circ}\ I\ A)$.

4.2 Specifications

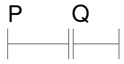
A *specification* of an interval can now be seen as an interval property, that is a member P of $IntProp$. An *implementation* of such a specification will be an interval I for which we can find a proof

$$p \;\in\; P\ I$$

The proof p contains information about exactly *how* the interval I meets the specification P. It will, for instance, state a point in an interval at which a $\Diamond$ property holds, and state which of a pair of disjuncts is valid. Examples are given in Sections 4.4 and 4.6 below.

4.3 Chop

An interval I satisfies $chop\ P\ Q$, where P and Q are interval propositions, if it can be split (or 'chopped') into two halves with the first satisfying P and the second Q.



In our logic a proof of a chop formula specifies the point at which the split is made, together with proofs that the resulting halves have the requisite properties themselves. It is formalised thus:

$$\left[\begin{array}{l} chop \in (P,\ Q \in IntProp) \rightarrow IntProp \\ \quad = \lambda\ P\ Q\ I \rightarrow Exists\ n \in Nat\ .\ And\ (P\ (take^{\circ}\ n\ I))\ (Q\ (drop^{\circ}\ n\ I)) \end{array}\right.$$

Note that because of the way that *take* and *drop* are defined the two halves of the interval, namely $(take^{\circ}\ n\ I)$ and $(drop^{\circ}\ n\ I)$, share an end point.

4.4 Examples of proof – I

To make some intervals over which to give examples we define the function *makeInt*. When applied to a natural number it makes an interval of that length, at each point of which all actions take place

$$\left[\begin{array}{l} allActs \in ActSet \\ \quad = \lambda\ A \rightarrow True \end{array}\right.$$

$$\left[\begin{array}{l} makeInt \in (a \in Nat) \rightarrow Interval \\ \quad = \lambda\ a \rightarrow \mathbf{case}\ a\ \mathbf{of} \left\{\begin{array}{lll} \mathbf{0} & \rightarrow & \mathbf{sing}\ allActs \\ \mathbf{S}\ n & \rightarrow & \mathbf{cons}\ allActs\ (makeInt\ n) \end{array}\right\} \end{array}\right.$$

Our first example shows that $makeInt\ two$ has length two:

$$\left[\begin{array}{l} len2 \in Length\ two\ (makeInt\ two) \\ \quad = \mathbf{trivial} \end{array}\right.$$

The proof here is **trivial**. Building on this we can give an example of a proof involving *chop*. Our goal is to prove that

$$chop\ (Length\ two)\ (Length\ one)$$

holds of the interval $makeInt\ three$. So, we define

$$\left[\begin{array}{l} egChop \in Set \\ \quad = chop\ (Length\ two)\ (Length\ one)\ (makeInt\ three) \end{array}\right.$$

What is a proof of this? Proofs of existential statements are **struct**s consisting of two things: a *witness* saying at which point the chop is made and a *proof* that the halves have the appropriate properties. This latter proof will itself be a conjunction of proofs that each half has the requisite property. Our proof is

$$\left[\begin{array}{l} chopPrf \in egChop \\ \quad = \textbf{struct} \left\{\begin{array}{l} witness = two \\ proof = \textbf{conj trivial trivial} \end{array}\right\} \end{array}\right.$$

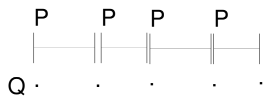
The chop at position *two* ensures that the first interval has length *two* and the second length *one*. In both cases the proofs of these facts are **trivial**.

4.5 Projection

A more powerful interval operation is that of projection. The proposition *proj P Q* holds for an interval if

- the interval can be split into a sequence of sub-intervals each of which satisfies P and also
- a new interval made up from the end points of the original interval satisfies the interval proposition Q.

In pictures we require that



Note that as was the case for *chop*, the end point of the first sub-interval is the first element of the second and so on.

In order to define *proj* we have to define two subsidiary operations, *allSat* and *select* which embody the two cases enumerated above. These in turn take a parameter of type *List Nat* — the type of (empty and non-empty) lists of natural numbers — given by the following declaration. The points in the list indicate the lengths of the sub-sequences of the list which form the projection; we therefore have no case in which the *List* is empty.[1]

$$\left[\begin{array}{l} List(A \in Set) \in Set \\ \quad = \textbf{data} \left\{\begin{array}{l} \textbf{Nil}\ (\,) \\ \textbf{Cons} \begin{pmatrix} head \in A \\ tail \in List\ A \end{pmatrix} \end{array}\right\} \end{array}\right.$$

To define *allSat* we perform a primitive recursion over the list of lengths

$$\left[\begin{array}{l} allSat \in (P \in IntProp,\ nums \in List\ Nat,\ I \in Interval) \to Set \\ \quad = \lambda\ P\ nums\ I \to \textbf{case}\ nums\ \textbf{of} \left\{\begin{array}{l} \textbf{Nil} \ \to\ \textbf{True} \\ \textbf{Cons}\ head\ tail\ \to \\ \quad And\ \ (P\ (take^{\circ}\ head\ I)) \\ \qquad\quad (allSat\ P\ tail\ (drop^{\circ}\ head\ I)) \end{array}\right\} \end{array}\right.$$

[1] Why then use *List* rather than *list* here? The answer is that the former gives a *Set* over which we can existentially quantify, whilst the latter only gives a type, and so we cannot use *Exists* to perform quantification over it.

In the interesting — **Cons** *head tail* — case *allSat* is a conjunction of two propositions. The first that the sub-interval of length *head* has property P; the second that all the subdivisions specified by *tail* have the property.

The function *select* is defined to pick out the end points specified by the subdivisions in the *List* of natural numbers.

$$\left[\begin{array}{l} select \in (T \in Type,\ b \in list\ T,\ c \in List\ Nat) \rightarrow list\ T \\ \quad = \lambda\ T\ b\ c \rightarrow \mathbf{case}\ c\ \mathbf{of} \left\{\begin{array}{l} \mathbf{Nil}\ \rightarrow\ \mathbf{sing}\ (last^{\circ}\ b) \\ \mathbf{Cons}\ head\ tail\ \rightarrow \\ \qquad \mathbf{cons}\ (first^{\circ}\ b)\ (select^{\circ}\ (drop^{\circ}\ head\ b)\ tail) \end{array}\right\} \end{array}\right.$$

Now we can put the two components together. To prove *proj* P Q I we need to provide a list of lengths into which I is split. Given this, we then have to prove the two obligations enumerated above.

$$\left[\begin{array}{l} proj \in (P, Q \in IntProp) \rightarrow IntProp \\ \quad = \lambda\ P\ Q\ I \rightarrow Exists\ x \in (List\ Nat)\ .\ And\ (Q\ (select^{\circ}\ I\ x))\ (allSat\ P\ x\ I) \end{array}\right.$$

4.6 Examples of proof – II

Here we give a simple projection proof using the *Length* predicate.

$$\left[\begin{array}{l} egProj \in Set \\ \quad = proj\ (Length\ two)\ (Length\ three)\ (makeInt\ six) \end{array}\right.$$

The proof is

$$\left[\begin{array}{l} projPrf \in egProj \\ \quad = \mathbf{struct} \left\{\begin{array}{l} witness = \mathbf{Cons}\ three\ (\mathbf{Cons}\ three\ \mathbf{Nil}) \\ proof = \mathbf{conj\ trivial}\ (\mathbf{conj\ trivial}\ (\mathbf{conj\ trivial\ trivial})) \end{array}\right\} \end{array}\right.$$

In this **struct** the *witness* gives the lengths into which the interval is cut, namely two intervals of length three.

In the *proof* part, the outer **conj** separates to give two proofs. The first is

$$\mathbf{trivial} \in Length\ two\ (select^{\circ}\ (makeInt\ six)\ (\mathbf{Cons}\ three\ (\mathbf{Cons}\ three\ \mathbf{Nil})))$$

which states that if we split the interval into sub-intervals of length three, the interval made up of the end points will have length two. (Recall that in this exposition the length of an interval is taken to be the number of points in the interval *minus one*.)

The second conjunct is

$$\begin{array}{l} \mathbf{conj\ trivial}\ (\mathbf{conj\ trivial\ trivial}) \in \\ \ allSat\ (Length\ three)\ (\mathbf{Cons}\ three\ (\mathbf{Cons}\ three\ \mathbf{Nil}))\ (makeInt\ six) \end{array}$$

which states that the sub-intervals all satisfy (*allSat*) the property that they have length three. These two conjuncts together show that the projection holds.

4.7 Temporal Operators

The traditional temporal operators have straightforward definitions in Alf. Given an interval proposition P, $next\ P$ should be valid over an interval if P holds over the tail, that is the interval starting at the *next* instant.

$$\left[\begin{array}{l} next \in (P \in IntProp) \rightarrow IntProp \\ \quad = \lambda\ P\ I \rightarrow \mathbf{case}\ I\ \mathbf{of} \left\{ \begin{array}{lcl} \mathbf{sing}\ x & \rightarrow & False \\ \mathbf{cons}\ x\ xs & \rightarrow & P\ xs \end{array} \right\} \end{array}\right.$$

We might, for instance, state that an action takes place at the next instant in time by

$$next\ (happensNow\ \mathbf{A})$$

The proposition $sometime\ P$ should be valid over an interval if P holds over some final segment of the interval (including possibly the whole interval itself). A recursive definition is

$$\left[\begin{array}{l} sometime \in (P \in IntProp) \rightarrow IntProp \\ \quad = \lambda\ P\ I \rightarrow \mathbf{case}\ I\ \mathbf{of} \left\{ \begin{array}{lcl} \mathbf{sing}\ x & \rightarrow & P\ I \\ \mathbf{cons}\ x\ xs & \rightarrow & Or\ (P\ I)\ (sometime\ P\ xs) \end{array} \right\} \end{array}\right.$$

Because of the constructive nature of our definition of Or it is possible to read off from a proof *which* segment it is over which the interval proposition holds. An explicit definition of 'sometime' states

$$\left[\begin{array}{l} sometimeAlt \in (P \in IntProp) \rightarrow IntProp \\ \quad = \lambda\ P\ I \rightarrow Exists\ n \in Nat\ .\ P(drop^{\circ}\ n\ I) \end{array}\right.$$

Again, a proof of this contains a witness n to the particular segment over which P holds.

The proposition $always\ P$ should be valid over an interval if P holds over every final segment of the interval. A recursive definition is

$$\left[\begin{array}{l} always \in (P \in IntProp) \rightarrow IntProp \\ \quad = \lambda\ P\ I \rightarrow \mathbf{case}\ I\ \mathbf{of} \left\{ \begin{array}{lcl} \mathbf{sing}\ x & \rightarrow & P\ I \\ \mathbf{cons}\ x\ xs & \rightarrow & And\ (P\ I)\ (always\ P\ xs) \end{array} \right\} \end{array}\right.$$

Since a proof of $And\ P\ Q$ must contain component proofs of P and Q, in the case of a non-empty interval it is necessary for both $(P\ I)$ and $(always\ P\ xs)$ to be provable, with the consequence that from a proof of $(always\ P\ I)$ it is possible to extract proofs of (P y) for every final segment y. An explicit definition of 'always' states

$$\left[\begin{array}{l} alwaysAlt \in (P \in IntProp) \rightarrow IntProp \\ \quad = \lambda\ P\ I \rightarrow (n \in Nat) \rightarrow P\ (drop^{\circ}\ n\ I) \end{array}\right.$$

It is clear here that the proof of $(alwaysAlt\ \ P\ \ I)$ gives a proof for all final segments, since the proof is a function taking a natural number n to a proof of $P\ (drop^{\circ}\ n\ I)$.

4.8 Lifting Propositional connectives

We can lift the standard propositional connectives in a uniform way – for instance to lift binary connectives we can define

$$\left[\begin{array}{l} liftC \in (conn \in (a, b \in Set) \rightarrow Set,\ \ P, Q \in IntProp) \rightarrow IntProp \\ \quad = \lambda\ conn\ P\ Q\ I \rightarrow conn\ (P\ I)\ (Q\ I) \end{array}\right.$$

and as examples we define the infix functions

$$\left[\begin{array}{l} andT \in (P, Q \in IntProp) \rightarrow IntProp \\ \quad = liftC\ And \end{array}\right.$$

$$\left[\begin{array}{l} impliesT \in (P, Q \in IntProp) \rightarrow IntProp \\ \quad = liftC\ (\lambda\ a\ b \rightarrow (c \in a) \rightarrow b) \end{array}\right.$$

We can then lift the theorem $implies\ (And\ A\ B)\ A$ and supply a proof:

$$\left[\begin{array}{l} liftThm \in (P, Q \in IntProp,\ I \in Interval) \rightarrow ((P\ andT\ Q)\ impliesT\ P)\ I \\ \quad = \lambda\ P\ Q\ I\ c \rightarrow \mathbf{case}\ c\ \mathbf{of}\ \{\mathbf{conj}\ p\ q \rightarrow p\} \end{array}\right.$$

The method we use here is uniform: any result that we have for our standard logic can be lifted to the Interval Logic in a similar way.

4.9 Discussion

The implementation discussed here can be modified in a number of ways. We might choose to represent intervals by finite functions rather than by finite lists; such a representation is more reminiscent of an implementation of a traditional (linear-time) temporal logic by means of functions over Nat; an implementation of this in Coq is reported in [10].

We reiterate that the interpretations of the (temporal) connectives $\vee$, $\diamond$, '$chop$' and '$proj$' are constructive. This means that from proofs of statements using these we can derive witnessing information about how exactly the result is proved; in other words each proof resolves the implicit non-determinism or under-specification in a particular way.

5 Conclusions and Future Work

This paper reports an implementation of an interval temporal logic using the Alf proof system. We have shown how proofs of temporal formulas can be executed, the proofs corresponding to (functional) programs and the formulas to the types of a programming language.

We argued in the Introduction that this separation between specification and implementation is valuable: specifications can be at a higher level of abstraction than their implementations. In particular, specifications can be non-deterministic whilst their implementations are deterministic.

In the future we intend to investigate a variety of topics.

- Type abstraction and (relative) completeness issues: our reasoning here is in terms of the representation of the logic. We seek an abstraction result which allows us to reason directly at the temporal level; from another point of view we look for a (relatively) complete temporal formal system for the primitives of our system.
- Refinement: we can model the refinement of one specification, S_1, by another, S_2, by giving the implication $S_2 \rightarrow S_1$. A proof of this implication in a constructive system provides a transformation from implementations of S_2 to implementations of S_1.
- Efficiency of implementation: we would like to investigate the efficiency of this style of implementation. We also envisage links with tableau methods of proof search, which can automate the construction of implementations in some cases.
- Actions and framing: as a matter of urgency we aim to investigate the constructive perspective on framing.
- Higher-level language: as a part of the *Mexitl* programme we will investigate the possibility of defining a higher-level language for multimedia description; the constructive approach can have a role to play in this definition and its implementation.

References

1. H. Barringer et al. MetateM: An imperative approach to temporal logic programming. *Formal Aspects of Computing*, 7(E), 1995.
2. Howard Bowman, Helen Cameron, Peter King, and Simon Thompson. Specification and prototyping of structured multimedia documents using interval temporal logic. Technical Report 3-97, Computing Laboratory, University of Kent at Canterbury, U.K., 1997. (Submitted for publication to ICTL97).
3. M. Carlsson and T. Hallgren. Fudgets - a graphical user interface in a lazy functional language. In *Functional Programming & Computer Architecture*, 93.
4. Z. H. Duan. *An Extended Interval Temporal Logic and A Framing Technique for Temporal Logic Programming.* PhD thesis, University of Newcastle Upon Tyne, May 1996.
5. D. Gabbay. Declarative past and imperative future: Executable temporal logic for interactive systems. In *Proceedings of the 7th ACM Symposium on the Principles of Programming Languages, Lecture Notes in Computer Science, vol. 398*, pages 402–450. Springer–Verlag, 1987.
6. R. Hale. Using temporal logic for prototyping: the design of a lift controller. In *Lecture Notes in Computer Science, vol. 379*, pages 375–408. Springer–Verlag, 1989.
7. Per Martin-Löf. Constructive mathematics and computer programming. In C. A. R. Hoare, editor, *Mathematical Logic and Programming Languages.* Prentice-Hall, 1985.
8. Ben Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.
9. John Peterson and Kevin Hammond, editors. *Report on the Programming Language Haskell, Version 1.3.* `http://haskell.cs.yale.edu/haskell-report/haskell-report.html`, 1996.

10. Simon Thompson. Temporal logic using the Coq proof assistant. Technical report, Computing Laboratory, University of Kent at Canterbury, U.K., 1997. (In preparation).
11. Simon J. Thompson. *Type Theory and Functional Programming.* Addison Wesley, 1991.