Automatic Generation of Theories

A G Hamilton
Department of Computing Science
University of Stirling
Stirling FK9 4LA
e-mail: agh@uk.ac.stir.cs

17th August 1994

Abstract

Constructive type theory (CTT) has potential as a framework for program synthesis. But the basic theory provides only a minimal collection of mathematical structures which can be used to represent data. Because of the uniform structure and the extensibility of CTT, it is proposed that new structures can best be introduced directly, via additional rules, rather than by representation in terms of basic structures. For arbitrary freely generated structures, the necessary rules can be constructed, starting from a simple form of specification. Further, it is shown how certain definitions can be made, and how theorems can be generated and proved by uniform methods. In particular, closure, uniqueness, cancellation and equality-decidability theorems are dealt with. All of the methods described have been fully implemented, using PICT, a proof assistant for CTT which has been developed by the author. Additional rules are generated in PICT's format, so that PICT can use them without modification, and new theorems are proved by PICT automatically. Examples are given.

Keywords: Constructive type theory, data structures, mechanized proof.

1 Introduction

The constructive type theory (CTT) of Martin-Löf [M-L84], now described excellently in the books by Nordström, Petersson and Smith [NPS90] and by Thompson [T91], provides a basis for the specification and development of programs. It has a rich language, in which some expressions (types) can represent specifications of programming tasks, some expressions (objects) can represent data, and some expressions (functions) can represent programs. The rules of the logic allow reasoning about these things, for example whether two objects are equal, whether two specifications are equivalent, or whether a given function meets a given specification. Martin-Löf's set of rules, however, is, in a sense, minimal. They deal directly with the standard logical connectives and quantifiers. The constructive interpretation of the connectives and quantifiers allows them to be used to represent data (e.g. conjunction for pairing, disjunction for disjoint union). But where data structures are concerned, the facilities are limited. There are rules for pairs, disjoint unions, finite types, natural numbers and the W type (Martin-Löf [M-L84], and that is all. It is a system for *constructive mathematics.*

The needs of program specification and development are different from those of mathematics. In particular, there is a need to design and use data structures which are appropriate to each situation. Now it is always the case that a programmer or specifier will be restricted in this task by the language which is used, and must choose representations for data according to the structures which are available. In the case of CTT, the structures available are basic and mathematical, and considerable work is necessary if appropriate representations of data are to be constructed.

There are two different approaches which might be taken over this in the context of CTT. One is to extend the facilities provided by building new structures in terms of existing ones. Indeed Martin-Löf's W type is designed with just this in mind - it is a general type construction which can be particularized to represent most recursive data structures, such as lists and trees. Also, in a more elementary way, *tuples* of arbitrary length can be constructed from a *pair* former. This approach was taken by Khamiss ([K86]), in relation to trees. It has also been taken by Stevens in work related to NuPRL [S89]. The other approach is to extend the system with new type formers and corresponding rules. Chisholm ([Ch87], for his derivation of a parsing algorithm, introduced new structures and rules which were appropriate for the task. Petersson [Pe82] also adopted this approach in his implementation of CTT, in providing, over and above the basic types of CTT, a list type former and a set of rules for lists. Experience with this implementation (and with subsequent ones developed by the author [H85, H91, H92]) has led to the belief that the second approach is more convenient in practice.

It is not only experience which has led us to this conclusion, however. One of the great virtues of CTT is its uniform structure, both in the language and in the logic, but particularly in the logic. This makes it possible to envisage the application of uniform methods when considering proof construction. This is discussed by Backhouse ([B86]) and by Ireland [I89, I91]. And when it comes to providing extensions to the basic theory the real benefits of uniformity are found. Not only can new rules for new data structures be produced in standard form, but standard existing methods of proof construction can then use these new rules without modification. In this paper are described the development of such methods, their implementation, and their use. The vehicle for implementation is PICT, a proof assistant for CTT, developed by the author [H91].

2 The Structure of CTT Language and Rules

The language of CTT is extremely simple. Its expressions are constants, abstractions and applications. These are governed by basic λ-conversion rules. Among the constants are types and type constructors, and objects and object constructors.

The rules of CTT are in natural deduction style. There are particular rules for special purposes (e.g. variable introduction, substitution, equality). Apart from these, every type constructor is dealt with in the same way:

1. A type constructor may have type arguments (or, in the case of the quantifiers, arguments which are type abstractions). In certain cases there may be no arguments, as with the type of Boolean values and the type of natural numbers.

2. Associated with a type constructor is a number of canonical object constructors. These may be nullary, as are $true$ and 0, or they may have arguments. If they have arguments, these may be recursive arguments, as is x in $succ(x)$, or non-recursive, as is h in $cons(h, t)$.

3. Associated with a type constructor also is a single non-canonical object constructor. Examples are rec for natural numbers, $cond$ for Booleans, and, less familiarly, $split$ for conjunctions (i.e. types of pairs).

4. A set of rules (in Martin-Löf's terms) expresses the meanings of the type constructor and the corresponding object expressions. Another way to regard this is that the set of rules governs the behaviour of the expressions in question. These rules have uniform structure:

   two formation rules,
   introduction rules (two for each canonical object constructor),
   two elimination rules,
   computation rules (one for each canonical object constructor).

The formation, introduction and elimination rules come in pairs, one of the pair dealing with equality judgements.

This uniform structure has been remarked on by others. In particular, Backhouse [B86] has presented a formalism for generating elimination and computation rules for a new type constructor from given formation and introduction rules. A slightly different approach is proposed here, namely to take a step further back, and to use as a starting point a specification of a data type in a form which is completely independent of CTT. From such a specification, appropriate rules and mechanisms for the type will be generated. The detail will be expanded below, but what is meant here by 'rules and mechanisms' are the following:

- Formation, introduction, elimination and computation rules for the new type.
- Two additional universe-introduction rules, so that the new type can be proved to belong to the universe.
- Definitions of useful terms.
- Basic 'structural' lemmas and theorems.

Now what must be contained in the specification which is to be the starting point? We certainly need the following:

1. The name of the new type constructor, and its arity.
2. Details of the intended structures of canonical objects.
3. The name of the non-canonical object constructor.

In Section 7 more detail will be given about what exactly is required and how it is to be represented.

These methods will deal uniformly with *freely generated* structures. They do not cope with equational constraints. The structures may be recursive, however. This does allow quite a broad range of structures, including, for example:

- Tuples (cases where there is only one canonical constructor, with an argument for each type parameter).
- Disjoint unions (where there is one canonical constructor for each type parameter, each having a single argument).
- Enumerated types (where the list of type parameters is empty, and all canonical constructors as nullary).
- Recursive structures such as lists and trees.

3 Rules

For illustration, here are two cases. Below are rules for two type constructors which are not included in the basic CTT. For reasons of space, the equality versions of the formation, introduction and elimination rules have been omitted. The presentation here is informal: the formalism for assumptions is not precise, and inessential assumptions have been omitted. When the details are discussed below, the syntax of PICT will be used, with greater precision.

3.1 Three-way Conjunction

Here the name of the type constructor is *And*3, and it has three type parameters, (here A, B and C). There is one canonical object constructor, *triple*, and the non-canonical object constructor is *split*3. Note that the object s which appears in the rules as one of the arguments to *split*3 is an abstraction.

$$\frac{A\ type \quad B\ type \quad C\ type}{And3(A,B,C)\ type}\ And3\text{-}form$$

$$\frac{a : A \quad b : B \quad c : C}{triple(a,b,c) : And3(A,B,C)}\ And3\text{-}intr$$

$$\frac{t : And3(A,B,C) \qquad \begin{array}{c}[q : And3(A,B,C)]\\ \vdash G(q)\ type\end{array} \qquad \begin{array}{c}[x : A, y : B, z : C]\\ \vdash s(x,y,z) : G(triple(x,y,z))\end{array}}{split3(t,s) : G(t)}\ And3\text{-}elim$$

$$\frac{a : A \quad b : B \quad c : C \qquad \begin{array}{c}[q : And3(A,B,C)]\\ \vdash G(q)\ type\end{array} \qquad \begin{array}{c}[x : A, y : B, z : C]\\ \vdash s(x,y,z) : G(triple(x,y,z))\end{array}}{split3(triple(a,b,c),s) = s(a,b,c) : G(triple(a,b,c))}\ And3\text{-}comp$$

3.2 Lists

Here the name of the type constructor is *List*, and it has one type parameter, (here A). There are two canonical object constructors, *nil* and *cons* (which is here represented by the infix '::'), and the non-canonical object constructor is *listrec*. The object *ind* which appears as one of the arguments to *listrec* is an abstraction.

$$\frac{A\ type}{List(A)\ type}\ List\text{-}form$$

$$\frac{A\ type}{nil : List(A)}\ List\text{-}intrnil \qquad\qquad \frac{a : A \quad l : List(A)}{a :: l : List(A)}\ List\text{-}intrcons$$

$$\frac{l : \mathit{List}(A) \quad \begin{array}{c}[q : \mathit{List}(A)]\\ \vdash G(q)\ \mathit{type}\end{array} \quad b : G(\mathit{nil}) \quad \begin{array}{c}[x : A, y : \mathit{List}(A), z : G(y)]\\ \vdash \mathit{ind}(x,y,z) : G(x :: y)\end{array}}{\mathit{listrec}(l, b, \mathit{ind}) : G(l)}\ \mathit{List\text{-}elim}$$

$$\frac{A\ \mathit{type} \quad \begin{array}{c}[q : \mathit{List}(A)]\\ \vdash G(q)\ \mathit{type}\end{array} \quad b : G(\mathit{nil}) \quad \begin{array}{c}[x : A, y : \mathit{List}(A), z : G(y)]\\ \vdash \mathit{ind}(x,y,z) : G(x :: y)\end{array}}{\mathit{listrec}(\mathit{nil}, b, \mathit{ind}) = b : G(\mathit{nil})}\ \mathit{List\text{-}compnil}$$

$$\frac{a : A \quad l : \mathit{List}(A) \quad \begin{array}{c}[q : \mathit{List}(A)]\\ \vdash G(q)\ \mathit{type}\end{array} \quad b : G(\mathit{nil}) \quad \begin{array}{c}[x : A, y : \mathit{List}(A), z : G(y)]\\ \vdash \mathit{ind}(x,y,z) : G(x :: y)\end{array}}{\mathit{listrec}(a :: l, b, \mathit{ind}) = \mathit{ind}(a, l, \mathit{listrec}(l, b, \mathit{ind})) : G(a :: l)}\ \mathit{List\text{-}compcons}$$

A uniform method for constructing such sets of rules has been implemented. Care has been taken to ensure that the structure of the rules is correct. Backhouse, Dyckhoff and others have noted omissions from the lists of premisses of Martin-Löf's rules (and from Petersson's implementations of them). Dyckhoff (personal communication) has shown exactly what is required, and his conclusions have been the starting point for this work.

4 Defined Terms

Because of the diversity of data structures which can be treated by these methods, the range of generally useful defined terms is somewhat limited. But there are some. Three groups are identified and treated here: *recognisers*, *projectors* and *selectors*. In all cases the terms being defined are *abstractions*. In CTT there is an important distinction between abstractions and functions. The former are a feature of the underlying theory of expressions, the latter are a feature of the logic.

1. Recognisers recognise the nature of an object — for example, to recognise whether a natural number is zero or is a successor. In the context of CTT such an operation may be defined in two ways. First, following 'propositions-as-types', it may be a type operation, evaluating to *Null* (the empty type) in one case and *Unit* (a type with one member) in the other. And second, it may be a Boolean operation. Specifically, for numbers, we could use either

   $(n)\mathit{rec}(n, \mathit{Unit}, (x,y)\mathit{Null})$ or
   $(n)\mathit{rec}(n, \mathit{true}, (x,y)\mathit{false})$

   There is no difficulty in general in introducing these as defined terms. Of course there is a need for them only when there is more than one object constructor.

2. Where there is only one object constructor (as with *And*3 above), projectors which evaluate to the various arguments are easily defined. For example

   $\mathit{second} = (t)\mathit{split3}(t, (x,y,z)y)$

3. Where there are two or more object constructors, CTT has a serious problem with selector operations. In the case of *List*, for example, it is the problem of defining an operation which returns the head of a list. Specifically, what to do if the list is empty? The formalism of CTT does not allow for *partial* operations (at least not without incurring considerable inconvenience). Operations on a type must be applicable to every member of the type. Where there is more than one object constructor, there is no simple way to restrict a selector to apply only to objects of the appropriate form. Here one way is described in which this problem can be partially circumvented.

   A type expression can be constructed which embodies the statement 'the head of list l is equal to a':

   $$\mathit{Head} = (A, l, a)\mathit{listrec}(l, \mathit{Null}, (x, y, z)\mathit{Eq}(A, x, a))$$

   (Note that $Eq(A, x, a)$ is a type which embodies the statement 'x is equal to a in type A'. Because A occurs in this expression, it must be abstracted (in effect, a parameter to *Head*). For particular values of A, l and a, $\mathit{Head}(A, l, a)$ is a type. This type will be *un*inhabited (regarded as a proposition it will be false) in two situations: when l is *nil*, and when l is non-empty but a is not equal to the head of l. Use of this type provides a means of selecting which is unencumbered by the need explicitly to distinguish the empty and non-empty cases.

   In general such terms can be constructed and given names automatically, for each argument of each object constructor in the specification of a new type.

5 Theorems

The term 'structural theorem' was used above to describe the kind of theorems which concern us. The point is that we wish to prove, by uniform methods, analogous theorems for every theory. This certainly constrains the range of theorems it is possible to consider. Four kinds are dealt with here. These are listed below with examples, and details of the generation of the proofs are given later.

1. **Closure Theorem**. Informally, every object in a type is equal to one whose outer structure is canonical. For example:

   (a) (for lists)
   $Eq(\mathit{List}(A), l, \mathit{nil}) \vee (\Sigma a : A)(\Sigma t : \mathit{List}(A))Eq(\mathit{List}(A), l, a :: t)$
   (under an assumption $l : \mathit{List}(A)$).

   (b) (for three-way conjunction)
   $(\Sigma a : A, b : B, c : C)Eq(\mathit{And}3(A, B, C), t, \mathit{triple}(a, b, c))$
   (under an assumption that $t : \mathit{And}3(A, B, C)$).

   Such closure properties may be expressed, in general, as disjunctions, where each disjunct corresponds to a distinct canonical object constructor. Generality is obtained by covering an apparently free variable by an assumption, rather than by use of an explicit universal quantifier. This makes the theorems easier to apply in practice. Implicitly, the types A, B and C will also be covered by assumptions (that they belong to the universe).

2. **Uniqueness Theorems**. These state that distinct canonical object constructors build distinct object expressions. For example:

   (a) (for lists)
   $\neg Eq(List(A), nil, a :: l)$

   (b) (for disjoint union, i.e. constructive disjunction)
   $\neg Eq(A \vee B, inl(x), inr(y))$

   Again, a, l, x and y (and A and B) are to be covered by assumptions. In general, there is a uniqueness theorem for each pair of canonical object constructors for the type. In cases where there is only one canonical constructor, there will be no uniqueness theorems. Such is the case with three-way conjunction, for example.

3. **Cancellation Theorems**. These express the fact that if two objects formed by application of the same canonical constructor are equal then their component parts must (separately) be equal. For example:

   (a) (for lists)
   $Eq(List(A), h1 :: t1, h2 :: t2) \Rightarrow Eq(A, h1, h2)$

   (b) (for three-way conjunction)
   $Eq(And3(A, B, C), triple(a1, b1, c2), triple(a2, b2, c2)) \Rightarrow Eq(B, b1, b2)$

   (with the appropriate assumptions). In general, there is a cancellation theorem for each argument of each canonical object constructor.

4. **Decidability of Equality**. For us, equality is Martin-Löf's extensional equality. Of course, this is not decidable in general. And, of course, this undecidability stems from the way that function types are dealt with. Some types (for example, the type of natural numbers) do have decidable equalities, and some type constructors have the property that if their argument types have decidable equalities, then so do the constructed types. Examples of this are conjunction, disjunction and list formation. It is this conditional decidability that can be proved in general for new type constructors (provided that they may be specified as described above).

   The decidability of equality for type A is expressed in CTT by:

   $$(\Pi x : A)(\Pi y : A)(Eq(A, x, y) \vee \neg Eq(A, x, y))$$

   If this is abbreviated to $Eqdec(A)$, then we may express the conditional decidability from above, for example, as:

   (a) (for lists)
   $Eqdec(A) \vdash Eqdec(List(A))$

   (b) (for three-way conjunction)
   $[Eqdec(A), Eqdec(B), Eqdec(C)] \vdash Eqdec(And3(A, B, C))$

6 PICT

Only a brief summary of PICT and its facilities can be given here. For further details, see the PICT Guide [H92]. The name PICT is an acronym, abbreviating 'Programs and Proofs in Constructive Type Theory'. It is the name of a program, written by the author, the purpose of which is to assist with the process of program synthesis within the formalism of CTT. More specifically, its purpose is to facilitate the construction of a proof object (i.e. program) `P`, given a type (i.e. specification) `S`, at the same time generating a proof of the CTT judgement that `P` belongs to (i.e. meets) `S`. The logic is that of Martin-Löf [M-L82, M-L84], following the implementation by Petersson [Pe82], with modifications suggested by Dyckhoff. All four judgement forms of Martin-Löf's formalism are included. The rules include versions of all of Petersson's rules except those for finite sets (and consequently includes additionally rules for the types *Null*, *Unit* and *Boolean*). They do not include rules for subset types or for quotient types. The rules, naturally, are grouped into *formation*, *introduction*, *elimination* and *computation* rules. Only small types (i.e. members of the first universe) are included, on the grounds that a great deal can be done at this level without incurring the extra complication that a hierarchy of universes brings. The equality type embodies *extensional* equality. The actual proof system comes from the same mould which produced Edinburgh LCF [GMW79], NuPRL [Co86] and Isabelle [Pa86]. But PICT has been designed specifically for CTT, and, further, specifically with the process of program synthesis in mind. It is not a general (far less generic) theorem prover.

6.1 Mode of Operation of PICT

The following list illustrates both the nature of PICT and the criteria which guided its design.

- The system is portable and does not require excessive time or space. It is written in Prolog, and has been successfully run on a number of different machines.
- It is an interactive system, and is simple to use. Input from from the user, when necessary, should be through use of a mouse or a minimum of keystrokes.
- Proofs are constructed in a goal-directed manner. The system maintains a proof tree and all the associated information. Housekeeping is all automatic and invisible, and information about the current state of a proof is readily obtainable.
- A toolkit is provided which allows the user to work above the level of the logical rules, and to build a proof tree in large chunks.
- The system carries out trivial steps by itself, and at appropriate points asks the user for guidance. The meaning of 'appropriate' is quite subtle. Ideally, user input should be restricted to those points in the proof where critical decisions have to be made. Sometimes this means *not* carrying out automatically an apparently obvious trivial step.
- Changes of mind or recovery from errors are as easy as possible, and involve only minimal disruption of the proof.
- There are facilities to extend the system, through making and saving definitions and through saving and applying theorems.

Details of the operation of PICT are given below, as necessary for the later descriptions of the mechanisms for generating new theories.

Extension of the proof tree (strictly, until it is complete it is merely a goal tree) is by matching a goal with the conclusion of a rule, and inserting the premisses (appropriately instantiated) into the tree as subgoals. Here a 'goal' is a CTT judgement, not merely a type. Goals may contain *scheme variables*, i.e. metavariables which stand for CTT expressions and which will subsequently become instantiated. Indeed the program synthesis process consists of instantiating an initial scheme variable to a value which is a member of the initially given type. The system manages these instantiations, most of which occur as a result of the tree extension process, though they can also be explicitly entered by the user. The matching process used is based on Prolog's unification. There is no higher-order unification.

6.2 Language of PICT

The use of Prolog as the implementation language has impinged on the design. First, as noted above, use is made of Prolog's unification process. Second, use is made of Prolog's lexical analyser. Input of CTT expressions is as Prolog terms, not as text. This is for efficiency reasons. And not much needs to be compromised as a result. But the following must be noted in order to understand the syntax used below.

- CTT syntax includes constants, abstractions and applications. Constants can be Prolog atoms. Less obviously, the other two must have an explicit Prolog operator as functor, so `x@expr` represents an abstraction, and `a^arg` represents an application. The functor in an abstraction is `@`, and the functor in an application is `^`.

- Atoms in Prolog must have an initial lower-case letter. It is customary (and useful) to distinguish the object level from the type level by use of lower- and upper-case initial letters, but that is disallowed by Prolog. Consequently, PICT uses names like `nAT`, `eQ`, `pI` and `lIS` for types and type constructors. And, for example, the defined term which was referred to above as *Eqdec*, in PICT is called `eQdec`.

This syntax is PICT's *external* syntax. PICT has also an internal syntax which reflects more effectively the actual language structure, the details of which need not be mentioned here.

To illustrate both the syntax and the structure of the rules, the rules for `aND` are shown in an appendix at the end of the paper. They are shown in external syntax, in a readable format, although they are actually stored in internal syntax. The rules contain Prolog variables (upper-case letters), enabling unification with the conclusion to instantiate the premisses. The appendix also contains the same rules in standard notation, to assist comprehension.

6.3 PICT Facilities

In PICT, the mechanism for a single proof tree extension step is the *tactic*. A tactic applies a single rule, in reverse, to generate subgoals. But a particular tactic may itself select which rule to use. For example, there are `form`, `intr`, `elim` and `comp` tactics, which will select from the appropriate category of rules, according to the form of the goal. These can cope without modification with additional rules for new data types.

PICT also contains the means to build *strategies* from individual tactics. These use standard strategy formers, like `or`, `thenl`, `repeat`, etc. Several strategies are built-in, to enable the user to work, much of the time, above the level of the basic rules and tactics. Because of the uniform way that the tactics work, the built-in strategies can also make use of additional rules for new data types.

In PICT, the user can make and store definitions, and there are mechanisms for unfolding and folding, with reference to the collection of stored definitions. Again, several are built-in.

Proved theorems can be saved. In order to be generally applicable, theorems are converted before being stored, this process substituting Prolog variables consistently for parameters which appear in the theorem. (This is not a generalization process, merely a renaming.)

The primary intention for PICT is program synthesis, in which case the initial input is an assumption list (in effect a list of declarations of parameters and their types) and a goal type. The task is then to synthesize a member of this goal type. But PICT can also be used as a *verifier*, i.e. to construct a proof of an explicit CTT judgement. In this case, no instantiations are found at the top level (but instantiations may still have to be managed during the construction of the proof).

At the heart of PICT is its `autostrat`. This is a strategy which carries out simple or obvious proof steps automatically. It consists of ten strategies, which are tried as alternatives, in a set order. The choice of the component strategies, and the order in which they are tried critically affects the performance of the system, and the present `autostrat` is the result of much experience. Automation is desirable, but the system should not be too eager, or it will too often take wrong turnings. Also the user must be able to control the essential structure of the proof, so that the outcome (the synthesized program) is a sensible and useful one. PICT works by applying `autostrat` if it can, then applying `autostrat` to the subgoals (depth-first, right-to-left). Whenever `autostrat` cannot be successfully applied, PICT asks the user for guidance. After, this guidance has been acted on, `autostrat` is again applied in the manner described above. On average, the system carries out roughly nine out of every ten proof extension steps automatically, and seeks guidance from the user for the tenth. This is very variable, however.

Although it is designed to be interactive, PICT can quite simply be made to work with a predetermined sequence of inputs which have been stored in a file. In this case the mode of operation is as described above, except that, when `autostrat` fails and guidance is required, that guidance is taken from the file (the next item in the file), and not from the user at the keyboard. This facility is used for the proofs described later in this paper, which are all carried out without user intervention, by working with pre-computed input scripts (called *guidance sequences*). The details are given below.

Last, PICT has mechanisms which allow recovery from error. At one extreme, a proof can be restarted from the beginning. At the other extreme, the single most recent proof extension step can be undone. And (most useful in practice) the proof can be undone as far as the point where the previous user intervention was made. There are also facilities to inspect the tree and the instantiations made so far, at any point during a proof.

7 Details

7.1 Specification of a New Type

The description given above is expanded here. The methods can deal with *free* structures only — so no equational constraints are allowed. The structures can be recursively defined, however, but mutually recursive types are beyond the scope of this work. The following are sufficient to specify a new type (examples are given for the case of the list type constructor):

1. The name of the new type constructor (*List*).
2. A list of names of (formal) type parameters ($[A]$).
3. A list of specifications of the forms of canonical objects. Each of these may be atomic (for nullary constructors), or may be a structure. In such a structure, the object constructor is the functor, and each argument is either the name of the new type (for recursive arguments) or one of the type parameters (for non-recursive arguments). At least one canonical object form must be nullary or have no recursive arguments. ($[nil, cons(A, List)]$)
4. The name of the non-canonical object constructor (*listrec*).

The four items above are represented in Prolog as arguments to a predicate `newtype`. The Prolog goal which will cause the details of the new theory to be generated and stored has the form

```
?- newtype(Former,Paramlist,Conspeclist,Rec).
```

The example given before was that of lists. In this case the Prolog goal would be

```
?- newtype(lIST,[a],[nil,cons(a,lIST)],listrec).
```

The name of the new type constructor is chosen as `lIST` (with initial lower-case letter, so that it is a Prolog atom), there is one type parameter, here represented by the atom `a` (it must be an atom, but the choice of name is not significant), `nil` is a nullary canonical object constructor, `cons` is a canonical object constructor with the specified argument types, and `listrec` is the name chosen for the non-canonical object constructor.

For the case of three-way conjunction as discussed above, the Prolog goal could be

```
?- newtype(aND3,[a,b,c],[triple(a,b,c)],split3).
```

For the case of strictly binary trees with data items held at terminal nodes, it might be

```
?- newtype(sBT,[a],[leaf(a),branch(sBT,sBT)],sbtrec).
```

For the case of an enumerated type, it might be

```
?- newtype(lEVS,[],[hi,med,lo],levcase).
```

7.2 Generating New Constants and Rules

The operation of PICT requires that the basic constants be 'declared' in Prolog facts. These simply hold basic information in a standard form, and there is no difficulty in constructing and asserting appropriate ones, given the information in a `newtype` goal. The only complication is computing the arity of the non-canonical object constructor. For example, in the case of `sBT`, the new facts would be

```
tconst(sBT,0->0).
const(leaf,0->0,can,sBT).
const(branch,0->0->0,can,sBT).
const(sbtrec,0->(0->0)->(0->0->0->0->0)->0,noncan,sBT).
```

The non-canonical constructor is an abstraction. Its first abstracted variable stands for a member of the constructed type. It has other abstracted variables, one for each canonical object constructor: if the constructor is nullary, this variable has arity `0`; if it is not nullary, this variable itself stands for an abstraction, with number of abstracted variables dependent on the number and type of the arguments to the non-canonical constructor in question. A recursive argument yields two, and a non-recursive argument yields one.

The rules for the new type must be constructed and asserted as facts in the standard form of the basic PICT rules. Besides the four categories of rules for the new type, also required (as noted above) are new universe-introduction rules. In detail, *every* PICT rule is a Prolog fact of the following form:

```
ttrule(Num,Cat,Former,Vlist,Evlist,Tlist,Conclusion,Premlist).
```

In the appendix at the end of this paper, the PICT rules for `aND` are given in full, as an illustration. Here, `Num` stands for a number, for reference purposes, `Cat` is `form`, `intr`, `elim` or `comp`, and `Former` is the name of the type constructor. Also `Conclusion` and `Premlist` are respectively the conclusion of the rule and the list of premisses of the rule. The other three arguments do not have significance in terms of the logic. Their sole purpose to assist Prolog's unification — to pass parameters in the course of operation. It is not necessary here to explain in detail what these are, but inspection of the rules in the appendix should give an indication.

The construction of the new rules required for a new data structure is complicated, but not difficult. The structures (both of PICT and of CTT) are sufficiently uniform, as observed above, to allow individual algorithms to construct arbitrary formation, introduction, elimination and computation rules. In each case the starting point is the given type specification — the elimination and computation rules are not based on the formation an introduction rules directly. Also note that what were type parameters in the specification must become Prolog variables in these rules.

7.3 Generating Closure Theorems

The type which embodies a closure property is a disjunction. There is one disjunct for each canonical object constructor. This disjunct is an equality if the constructor is nullary, and is an existentially quantified equality otherwise. Constructing the closure type is not difficult. Nor indeed is constructing an object which inhabits the closure type. In the example case of lists, they are, respectively

```
oR^(eQ^(lIST^tA,l,nil),
    sIG^(tA,(x)@sIG^(lIST^tA,(y)@eQ^(lIST^tA,l,cons^(x,y)))))

listrec^(l,inl^eq,(x,y,z)@inr^(pr^(x,pr^(y,eq))))
```

As noted in Section 5.2, the variables which are apparently free here are implicitly covered by appropriate assumptions. (Note that `eq` is the canonical object in an equality type.)

But of course it is not sufficient merely to construct these — it is necessary to prove that the object inhabits the type. PICT can do this, operating in its normal fashion, without modification and without guidance. For this proof PICT in its verification mode may be used, giving as initial input the full explicit judgement, including necessary assumptions, for example in the case of lists above that `l` belongs to `lIST^tA` and that `tA` inhabits the (first) universe. PICT's `autostrat` will complete the proof in every case. It would not be able to do this if PICT were being used in synthesis mode, because then the structure of the object would not be available to help in determining the steps in the proof. PICT generates the proof tree, and saves the theorem, with a name generated by the system (the name of the type constructor with `"cl"` appended. This proof tree can be substantial — in the case of `oR4` (four-way disjunction) it has 1018 nodes.

7.4 Generating Uniqueness Theorems

What are referred to here as uniqueness theorems are characteristic of freely generated structures. An object whose outer structure is an application of one canonical constructor cannot be equal to an object whose outer structure is an application of a different canonical constructor. The first observation that can be made is that if there is only one canonical object constructor then there are no uniqueness theorems. If there are n canonical constructors then there are $n * (n-1)/2$ uniqueness theorems.

The type which embodies a uniqueness property is a negated equality. In CTT, of course, negation is not primitive, and $\neg T$ means $T \Rightarrow \mathit{Null}$ (or, in PICT notation: `fUN^(T,nULL)`). Objects which inhabit negated types do not have useful computational content (beyond being functions), and correspondingly do not have structure which can assist a verification process, as the object in the closure type did. Consequently, for the proofs of uniqueness theorems, PICT is used in synthesis mode — PICT is given the type, and it constructs the object. This time, however, `autostrat` is not up to the task, and these proofs require guidance. Nevertheless, the proofs are carried out automatically. In order for this to happen, a script of inputs (a guidance sequence) for each proof is constructed in advance, and is fed to PICT while the proof is being carried out.

The structure of all uniqueness proofs is the same, and can be illustrated by that for natural numbers. Here the uniqueness type is `fUN^(eQ^(nAT,0,s^x),nULL)` (`x` is covered by an assumption). From the equality of `0` and `s^x` we can deduce the equality of `iSzero^0` and `iSzero^(s^x)`. Presuming that `iSzero` is the recogniser which evaluates to `uNIT` or `nULL`, we can deduce the equality of `uNIT` and `nULL`, and hence construct a member of `nULL`. Note that one of the newly-defined recognisers is used. This proof outline determines the guidance sequence which is necessary. To illustrate, here is the guidance sequence which the system constructs in order to generate this proof:

```
[i,inst,unit,eqtype,uNIT,v2,iSzero^0,iSzero^(s^x),u,u]
```

As described in Section 6, PICT's `autostrat` generates the proof tree, but sometimes pauses to ask for guidance. The members of this sequence are the individual items of guidance. It is important to realise that `autostrat` may well construct substantial portions of the proof tree in between using these guidance items. Of course this sequence is never visible to the user. Nevertheless, we should note its structure. It is not necessary here to give the details of PICT commands, but briefly: `i`, `inst` and `eqtype` are PICT tactics, `v2` is a PICT strategy, and `u` is a request to unfold defined terms. The proof tree in this case actually has 68 nodes. The essential point is that in all other cases the guidance sequences for the proofs of uniqueness theorems are exactly the same, except for the two entries which involve the expression containing the recogniser. The proofs are not large in general — for the example of `oR4`, there are 213 nodes.

The system stores these theorems after they have been proved, with names generated internally. For two constructors called `con1` and `con2`, the corresponding uniqueness theorem is called `con1_con2`. These theorems are used in the course of subsequent proofs.

7.5 Generating Cancellation Theorems

Cancellation theorems are assertions of the following form: two equal objects formed by application of the same canonical constructor have equal corresponding subobjects. Again, for each type there will be a set of cancellation theorems. For each canonical object constructor associated with the type there will be a cancellation theorem for each of its arguments. The type which embodies a cancellation property is a type of functions from one equality to another. The object which is to inhabit such a type again has neither useful computational content nor useful structure to help a proof. So here again PICT is used in synthesis mode. And here too PICT requires guidance via a prepared script. The situation is more complicated, however. There is a uniformity to the structure of all proofs of cancellation theorems, but this uniformity depends not only on definitions (as with the uniqueness proofs), but also on *lemmas*. The definitions which are needed here are the 'selectors' mentioned in Section 4. The lemmas express certain properties of these defined terms.

Let us illustrate again with the example of lists. Corresponding to the constructor `cons` there will be two cancellation theorems. There are two selectors: `sCONS1` and `sCONS2`. (The upper-case characters indicate that these are *type* abstractions.)

`sCONS1^(tA,l,h)`	means	'`h` is equal to the head of list `l` in `lIST^tA`'
`sCONS2^(tA,l,t)`	means	'`t` is equal to the tail of list `l` in `lIST^tA`'

Four lemmas are required in this case (in general, two for each defined selector, i.e. two for each argument to the canonical object constructor in question). These lemmas are (only the types involved are given):

1. `sCONS1^(tA,cons^(h,t),h)`
2. `sCONS2^(tA,cons^(h,t),t)`
3. `fUN^(sCONS1^(tA,cons^(h,t),x),eQ^(tA,x,h))`
4. `fUN^(sCONS2^(tA,cons^(h,t),x),eQ^(lIST^tA,x,t))`

These are referred to as *selection* lemmas (the first two) and *selection-uniqueness* lemmas (the last two).

These lemmas require to be proved and stored before the cancellation theorem can be proved. However, the proof structures are sufficiently uniform for the same form of guidance sequence to be used for all of them. For the selection lemmas the sequence contains only three inputs, and for the selection-uniqueness lemmas the number is four. The proof trees for these four lemmas (for lists) contain respectively 34, 38, 69 and 81 nodes. The lemmas are proved and stored, with names chosen by the system (by appending `"1_sel"`, `"2_sel"`, etc. and `"1_selu"`, `"2_selu"`, etc. to the name of the object constructor).

These lemmas are certainly very obvious and trivial, but their exact forms have been carefully chosen so that not only do their own proofs have uniform structure which enable automatic proofs, but also so that their use fits into a uniform proof structure for the cancellation theorems in all cases.

To show the form of the guidance sequences for the cancellation theorems, here is one which is generated in the case of lists, for the cancellation theorem for `cons` (for its first argument) mentioned in Section 5:

```
[i,hyp,Ityp,thm,cons1_selu,seeq,Obj,thm,cons1_sel]
```

For convenience here `Ityp` stands for `sCONS1^(tA,cons^(h1,t1),h2)` and `Obj` stands for `cons^(h1,t1)`. Note that an entry `thm` in such a sequence is a call to the theorem (lemma) whose name is the next entry in the sequence. Also, `hyp` is a PICT tactic, and `seeq` is a strategy. The proof tree constructed by this guidance sequence has 98 nodes. In the general case the sequence is the same, except that the values of `Ityp` and `Obj` will vary, and the names of the lemmas used will be different.

The cancellation theorems are proved and stored. The system constructs names for them by appending `"1_canc"`, `"2_canc"`, etc. on the name of the object constructor. They are used in the decidability proofs described next.

7.6 Generating Equality-Decidability Theorems

Equality is decidable on new types formed in the way that has been described, provided that equality is decidable on all the type parameters involved. This result again has a uniformity which can be exploited in order to build an algorithm which will prove it in all cases. This time, however, the uniformity is rather harder to find. As above, it is necessary to construct intermediate lemmas (described below), but this time the PICT guidance sequence has to be constructed by a more complicated process. The proof tree does not have the same structure in all cases. Instead, the proof trees are all built up from similarly formed branches, but arranged differently for the different proofs. To form the guidance sequence, it is necessary to form individual parts corresponding to the different branches of the proof tree, compose them into a tree structure, and then flatten them into a list.

But first let us examine the necessary lemmas, called here *constructor-decidability* lemmas. There is one of these for each canonical object constructor. Again, their exact forms have been chosen to bring out the uniformity of the proofs in which they are used. They take different forms in the cases where the constructor is nullary and when it is not. Using `lIST` again to illustrate:

```
eQdec2^(lIST^tA,nil,l)
```

```
[eQdec2^(tA,h1,h2),eQdec2^(lIST^tA,t1,t2)] /-
                 eQdec2^(lIST^tA,cons(h1,t1),cons(h2,t2))
```

Here yet another abbreviatory definition (which is known to PICT) has been used. `eQdec2` is an abstraction with three abstracted variables, and

```
eQdec2^(A,X,Y)    means    oR^(eQ^(A,X,Y),nOT^(eQ^(A,X,Y)))
```

The lemmas have names, formed by appending `"_dec"` on the name of the object constructor. The proofs of these lemmas for nullary constructors are very straightforward, and always have the same structure, appealing to an appropriate uniqueness theorem. The guidance sequence for `nil_dec` is:

```
[1,il,ir,negthm,nil_cons]
```

The number `1` is PICT's way of calling the `elim` tactic used in conjunction with the *first* assumption. The `il` and `ir` are the `intr` tactic (for `oR`, introducing into the left disjunct and right disjuncts, respectively), and `negthm` appeals to a (named) theorem which happens to be a negated equality. (The effect of `negthm` is that it can use the named theorem directly if appropriate, or can first do the necessary manipulation if the goal is `nOT^(eQ^(A,X,Y))` and the theorem is `nOT^(eQ^(A,Y,X))`.)

In the case of non-nullary constructors, the generation of the proof tree starts with as many applications of the `oR`-elimination rule as there are arguments to the constructor. The leftmost subgoal then is the only case when the two terms *are* equal, and in all the others they are unequal, so the required object is an injection into the right (negated) disjunct. These require appeals to cancellation theorems. Given below are the guidance sequences for `succ_dec`, `cons_dec` and `tree_dec`, to show the varying forms:

```
[1,il,i,ir,hsthm,num1_canc]

[2,2,il,i,ir,hsthm,list1_canc,
          ir,hsthm,list2_canc]

[3,3,3,il,i,ir,hsthm,tree1_canc,
            ir,hsthm,tree2_canc,
            ir,hsthm,tree3_canc]
```

A number which appears in these sequences indicates a call to the `elim` tactic, in conjunction with the assumption with the given position in the (then) current assumption list. Otherwise the only PICT feature appearing here which has not figured before is the strategy `hsthm`. This is for convenient use of theorems which are implications. (The prefix `"hs"` stands for 'hypothetical syllogism'). When the goal has the form (schematically) `fUN^(B,C) /- fUN^(A,C)` and the theorem has the form `/- fUN^(A,B)`, this strategy will do the work. This is applicable particularly if `C` is `nULL`, i.e. when the goal has form `nOT^B /- nOT^A`.

These lemmas can have substantial proofs (the proof tree for `tree_dec` has 1001 nodes).

Now the final proof can be carried out, that of the equality-decidability theorem. Here again, the varying structure of the proof tree must be reflected in the constructed guidance sequence. Essentially, there are two applications of the elimination tactic for the new constructor. Amongst the subgoals generated by each of these will be one subgoal for each of the canonical object constructors. These must be treated differently, depending on whether the constructor is nullary or not, and, at the second level, whether the constructor is the same one as on the same branch of the proof tree at the previous level. The decidability lemmas are used, as are the uniqueness theorems. Here are example guidance sequences, generated by the system, for natural numbers, lists and trees, again:

```
[u,i,1,i,thm,zero_dec,i,1,ir,negthm,zero_succ,thm,succ_dec,4]

[u,i,1,i,thm,nil_dec,i,1,ir,negthm,nil_cons,thm,cons_dec,5]

[u,i,1,i,thm,empty_dec,i,1,ir,negthm,empty_tree,
        thm,tree_dec,u1,eQdec2,7,u1,eQdec2,9,u1,eQdec2,pi2]
```

The numbers indicate the `elim` tactic as described before (the position of the correct assumption to use is computed in advance). The entry `u1` is a request to unfold the definition whose definiendum is the next entry in the script. And `pi2` is a PICT strategy.

Finally, perhaps it should be emphasised again that a PICT guidance sequence contains the inputs needed to guide a proof, so the sequence is a flattened tree of inputs, the tree representing the critical or non-trivial structure of the whole proof tree. In the case of binary trees above, the guidance sequence for the equality-decidability theorem contains 22 entries. These entries are guidance given to PICT at 16 of the 429 nodes which make up this particular proof tree. The proof extension steps at the other nodes are carried out by `autostrat`. The structure of the proof tree can be represented, therefore, by the tree shown in Figure 1.

The nodes omitted (403 of them) from this tree are the ones at which guidance is not necessary. Experience suggests that they are also nodes which are not significant in terms of understanding the structure of the proof. It has been argued elsewhere (in particular by Backhouse) that proofs can provide documentation for the proof process. Full proofs, with all their detail, cannot be useful except in very simple cases, because of their scale. However, a record of a sequence of inputs to PICT, together with information about the positions in the tree where they were required, and about the goals held in the tree at those positions, does have potential to be useful in this way. Indeed PICT produces as an output at the end of a proof just such a record, intended specifically as documentation.

8 Examples

8.1 Inputs

Listed below are some examples which have been successfully tried. The last one is exactly the 'parse tree' data structure which Chisholm used for his parsing algorithm derivation ([Ch87]). Note that the first seven are *basic* structures which are part of the PICT implementation of CTT in any case. That these can be treated by the system is no surprise, as it is the uniformity of the rules for the basic structures which leads to the possibility of extensions as described in this paper. But it is reassuring to know that the general process does in fact generate rules which are identical to the existing ones in such cases. Of course in practice *duplication* of rules (or of identifiers used as type or object constructors) cannot be allowed, so when the extension process is used on such basic structures, the names chosen for the constructors must be different from existing ones. This check (amongst others) is incorporated into `newtype`.

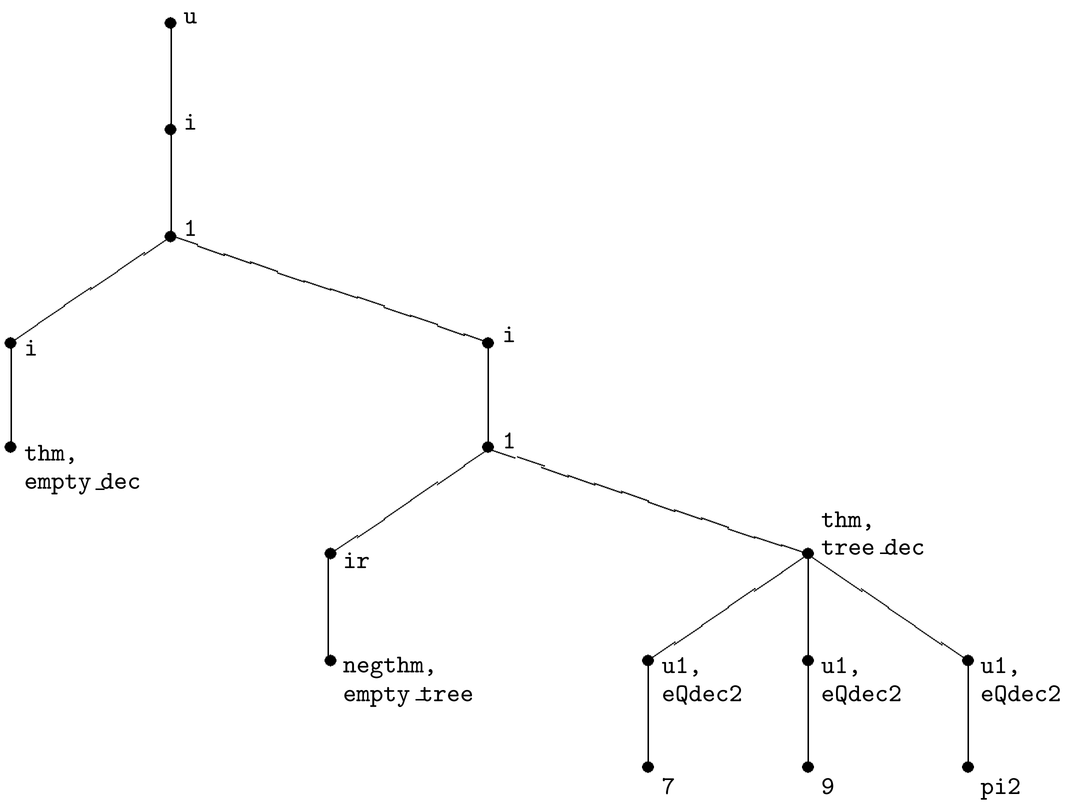

Figure 1: Skeleton proof tree

```
newtype(eMPTY,[],[],negcase).

newtype(tRUE,[],[u],tcase).

newtype(bOOLEAN,[],[true,false],if).

newtype(aND2,[a,b],[pair(a,b)],split2).

newtype(oR2,[a,b],[i(a),j(b)],wh2).

newtype(nUM,[],[zero,succ(nUM)],numrec).

newtype(lIST,[a],[nill,list(a,lIST)],listrec).

newtype(dAYS,[],[mon,tue,wed,thu,fri],daycase).

newtype(lEVS,[],[hi,mid,lo],levcase).

newtype(aND3,[a,b,c],[triple(a,b,c)],split3).

newtype(oR3,[a,b,c],[i3(a),j3(b),k3(c)],wh3).

newtype(oR4,[a,b,c,d],[i4(a),j4(b),k4(c),l4(d)],wh4).

newtype(sBT,[a],[leaf(a),branch(sBT,sBT)],sbtrec).

newtype(tREE,[a],[empty,tree(a,tREE,tREE)],treerec).

newtype(pT,[t,o1,o2],[tip(t),n1(o1,pT),n2(o2,pT,pT)],prec).
```

8.2 Outputs

Here is a list of all the data produced by `newtype` in the case of the type constructor `sBT` (which forms strictly binary trees with data items at the leaf nodes):

1. **Constants**. Facts declaring `sBT`, `leaf`, `branch` and `sbtrec`.
2. **Rules**. Two formation rules, four introduction rules, two elimination rules and two computation rules. Also two universe-introduction rules are constructed.
3. **Definitions**. These are:

   `iSleaf`, `iSbranch` (type-valued recognisers),
   `isleaf`, `isbranch` (Boolean-valued recognisers),
   `sLEAF1`, `sBRANCH1`, `sBRANCH2` (selectors).
4. **Theorems**. These are:

`sBTcl`	(closure theorem)
`leaf_branch`	(uniqueness theorem)
`leaf1_sel`	(selection lemma)
`branch1_sel`	(selection lemma)
`branch2_sel`	(selection lemma)
`leaf1_selu`	(selection-uniqueness lemma)
`branch1_selu`	(selection-uniqueness lemma)
`branch2_selu`	(selection-uniqueness lemma)
`leaf1_canc`	(cancellation theorem)
`branch1_canc`	(cancellation theorem)
`branch2_canc`	(cancellation theorem)
`leaf_dec`	(constructor-decidability lemma)
`branch_dec`	(constructor-decidability lemma)
`eqdecsBT`	(eq-decidability theorem)

The operation of `newtype` has been timed, using the C Prolog interpreter on a Hewlett-Packard 9000/375 workstation. The run detailed above, for `sBT`, including proofs of fourteen lemmas and theorems, takes 196 CPU seconds. The total number of proof steps involved in this is 2419. The most substantial example which has been done is the parse tree example, which included constructing proofs of twenty six separate lemmas and theorems, involving a total of 9959 proof steps, and took 27.5 minutes of CPU time.

9 Conclusion

Using the opportunity provided by the clean and uniform structure of the language and logic of CTT and its reflection in the proof system PICT, a tool has been constructed which enables a user very quickly and easily to extend the basic CTT by adding theories for new data structures. Such new data structures can then be used in specifications which are the starting points for the program synthesis process, just as though they had been present in the basic theory, since the language structures, rules, lemmas and theorems have been constructed which can form the basis for reasoning with them.

It may prove possible to extend this provision, by constructing further useful definitions, lemmas and theorems (and perhaps strategies), but these are likely to be more particular, applicable only to structures of certain forms. This is because, as observed earlier, the number of useful theorems which are absolutely general is quite limited. It may also be possible to extend the work by considering the addition of new mutually recursive types and of new type structures which are not freely generated.

With regard to definitions in particular, the generation of new definitions would be assisted further by provision of a facility to construct a *definiens*, given a definiendum and an implicit (equational, possibly recursive) specification of its properties.

Acknowledgements

The ideas on which this work is based have been common knowledge for some little time. I am grateful to Roland Backhouse for his substantial efforts in clarifying them. And I am

particularly grateful to Andrew Ireland for numerous discussions about what may be possible in practice. The work described here is a realisation of some of our joint ideas.

Appendix: CTT Rules in PICT

1. Prolog Forms

Here are the rules for *And*, to give an indication of the nature and structure of the rules of CTT generally, and of the way that they represented in PICT. Each of the following is a Prolog fact. In each case the last two arguments are respectively the conclusion of the rule and the list of premisses. Here these have been written with a commented line of dashes between them to indicate the structure. Note that identifiers which are, or which start with, upper-case letters are Prolog variables. Note also that because the rules are applied *backwards*, the assumption list (`L`) in the conclusion is carried into all of the premisses. In PICT, the non-canonical constant *split* is abbreviated to `spl`.

```
ttrule(108,form,aND,[],[],[],

    L /- typ(aND^(A,B)),
    /*-----------------*/
    [L /- typ(A),
     L /- typ(B)]).


ttrule(109,form,aND,[],[],[],

    L /- eqtyp(aND^(A,B),aND^(C,D)),
    /*-----------------------------*/
    [L /- eqtyp(A,C),
     L /- eqtyp(B,D)]).


ttrule(213,intr,aND,[_],[],[],

    L /- elem(pr^(X,Y),aND^(A,B)),
    /*---------------------------*/
    [L /- elem(X,A),
     L /- elem(Y,B)]).


ttrule(214,intr,aND,[_],[],[],

    L /- eqelem(pr^(X1,Y1),pr^(X2,Y2),aND^(A,B)),
    /*------------------------------------------*/
    [L /- eqelem(X1,X2,A),
     L /- eqelem(Y1,Y2,B)]).
```

```
ttrule(315,elim,aND,[Q],[Cx,Cy],[A,B],

    L /- elem(spl^(P,B),G^P),
    /*----------------------------------------------------------*/
    [L /- elem(P,aND^(A,B)),
     [elem(Q,aND^(A,B))|L] /- typ(G^Q),
     [elem(Cx,A),elem(Cy,B)|L] /- elem(B^(Cx,Cy),C^(pr^(Cx,Cy)))]).


ttrule(316,elim,aND,[Q],[Cx,Cy],[A,B],

    L /- eqelem(spl^(P1,B1),spl^(P2,B2),G^P1),
    /*----------------------------------------------------------*/
    [L /- eqelem(P1,P2,aND^(A,B)),
     [elem(Q,aND^(A,B))|L] /- typ(G^Q),
     [elem(Cx,A),elem(Cy,B)|L]
                /- eqelem(B1^(Cx,Cy),B2^(Cx,Cy),C^(pr^(Cx,Cy)))]).


ttrule(410,comp,aND,[Q],[Cx,Cy],[A,B],

    L /- eqelem(spl^(pr^(X,Y),D),D^(X,Y),G^(pr^(X,Y))),
    /*----------------------------------------------------------*/
    [L /- elem(X,A),
     L /- elem(Y,B),
     [elem(Q,aND^(A,B))|L] /- typ(G^(Q)),
     [elem(Cx,A),elem(Cy,B)|L] /- elem(D^(Cx,Cy),C^(pr^(Cx,Cy)))]).
```

2. Standard Notation

Now, for comparison, here are the same rules, expressed in a more normal format and notation. (Note that ++ stands for list concatenation.)

$$\begin{array}{l} L \vdash A \ \textit{type} \\ L \vdash B \ \textit{type} \\ \hline L \vdash \textit{And}(A,B) \ \textit{type} \end{array} \textit{And-form}$$

$$\begin{array}{l} L \vdash A = C \\ L \vdash B = D \\ \hline L \vdash \textit{And}(A,B) = \textit{And}(B,D) \end{array} \textit{And-formeq}$$

$$\begin{array}{l} L \vdash x : A \\ L \vdash y : B \\ \hline L \vdash \textit{pr}(x,y) : \textit{And}(A,B) \end{array} \textit{And-intr}$$

$$\frac{\begin{array}{l} L \vdash x1 = x2 : A \\ L \vdash y1 = y2 : B \end{array}}{L \vdash pr(x1, y1) = pr(x2, y2) : \mathit{And}(A, B)}\,\mathit{And\text{-}intreq}$$

$$\frac{\begin{array}{l} L \vdash p : \mathit{And}(A, B) \\ [q : \mathit{And}(A, B)] \mathbin{++} L \vdash G(q)\ \mathit{type} \\ [cx : A, cy : B] \mathbin{++} L \vdash b(cx, cy) : G(pr(cx, cy)) \end{array}}{L \vdash \mathit{spl}(p, b) : G(p)}\,\mathit{And\text{-}elim}$$

$$\frac{\begin{array}{l} L \vdash p1 = p2 : \mathit{And}(A, B) \\ [q : \mathit{And}(A, B)] \mathbin{++} L \vdash G(q)\ \mathit{type} \\ [cx : A, cy : B] \mathbin{++} L \vdash b1(cx, cy) = b2(cx, cy) : G(pr(cx, cy)) \end{array}}{L \vdash \mathit{spl}(p1, b1) = \mathit{spl}(p2, b2) : G(p)}\,\mathit{And\text{-}elimeq}$$

$$\frac{\begin{array}{l} L \vdash x : A \\ L \vdash y : B \\ [q : \mathit{And}(A, B)] \mathbin{++} L \vdash G(q)\ \mathit{type} \\ [cx : A, cy : B] \mathbin{++} L \vdash d(cx, cy) : G(pr(cx, cy)) \end{array}}{L \vdash \mathit{spl}(pr(x, y), d) = d(x, y) : G(pr(x, y))}\,\mathit{And\text{-}comp}$$

References

[B86] R.C. Backhouse. On the Meaning and Construction of the Rules in Martin-Löf's Theory of Types. Computing Science Notes CS8606, Department of Mathematics and Computing Science, University of Groningen, 1986.

[Ch87] P. Chisholm. Derivation of a Parsing Algorithm in Martin-Löf's Theory of Types. Science of Computer Programmimg 8, pp 1-42, 1987.

[Co86] R.L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice Hall, 1986.

[GMW79] M.J.C. Gordon, R. Milner, C. Wadsworth. *Edinburgh LCF.* Springer-Verlag, LNCS 78, 1979.

[H85] A.G. Hamilton. Program Construction in Martin-Löf Type Theory. Technical Report TR24, Department of Computing Science, University of Stirling, 1985.

[H92] A.G. Hamilton. The PICT Guide, Technical Report TR99, Department of Computing Science, University of Stirling, 1992.

[H91] A.G. Hamilton. The PICT System. In C. Rattray and R.G. Clark, editors, *The Unified Computation Laboratory.* Proceedings of the IMA Conference, Stirling, 1990, pp 437-448. Oxford University Press, 1991.

[I89] A. Ireland. Mechanization of Program Construction in Martin-Löf's Theory of Types. Ph.D. thesis, University of Stirling, 1989.

[K86] A.M.A. Khamiss. Program Construction in Martin-Löf's Theory of Types. Ph.D. thesis, University of Essex, 1986.

[I91] A. Ireland. On Exploiting the Structure of Martin-Löf's Theory of Types. Proceedings of the Seventh Austrian Conference on Artificial Intelligence, Vienna, 1991, pp 126-136. Springer-Verlag, Informatik-Fachberichte 287, 1991.

[M-L82] P. Martin-Löf. Constructive Mathematics and Computer Programming. In L.J. Cohen, J. Los, H. Pfeiffer and K-P. Podewski, editors, *Logic, Methodology and Philosophy of Science VI*, pp 153-175, North-Holland, 1982.

[M-L84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[NPS90] B. Nordström, K. Petersson, J. Smith. *Programming in Martin-Löf's Type Theory.* Oxford University Press, 1990.

[Pa86] L.C. Paulson. Natural Deduction as Higher-Order Resolution. Journal of Logic Programming 3, pp 237-258, 1986.

[Pe82] K. Petersson. A Programming System for Type Theory. LPM Memo 21, Department of Computer Science, Chalmers University of Technology, Göteborg, 1982.

[S89] A. Stevens. An Improved Method for the Mechanisation of Inductive Proof. Ph.D. thesis, University of Edinburgh, 1989.

[T91] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.