On Exploiting the Structure of Martin-Löf's Theory of Types[1]

ANDREW IRELAND

Department of Artificial Intelligence, University of Edinburgh, Edinburgh EH1 1HN, UK

Program synthesis in Martin-Löf's Theory of Types is a theorem proving activity. We show how the uniform structure of the deductive system may be exploited in the mechanization of this activity. Basic properties of data type constructors are shown to exhibit a general structure in the way in which they are expressed and derived. A proof procedure for negation is developed, based upon uniqueness, closure and cancellation properties. As a consequence our proof procedure may be extended uniformly to incorporate new data types.

Received December 1990

1. INTRODUCTION

Martin-Löf's Theory of Types [14, 16, 20] is a rich logic in which provably correct programs can be synthesized. Program synthesis is a theorem proving activity. This paper investigates how the structure of the theory can be exploited in the mechanization of this activity. In particular we develop a proof procedure for negation based upon general properties of data types which can be constructed mechanically. Following this methodology leads to a proof procedure which is uniformly extendible.

In Section 2 the general structure of the deductive system is presented. Based upon this presentation we show, in Section 3, how properties of data type constructors may be expressed and derived within the theory. By way of illustration, we consider the formulation of uniqueness, closure and cancellation properties. The derivations of these properties also exhibit a general structure. This general structure provides the basis for mechanically deriving properties of arbitrary data type constructors. In Section 4 we develop a proof procedure for negation based upon uniqueness, closure and cancellation properties. Related work and conclusions are presented in Sections 5 and 6, respectively.

2. THE STRUCTURE OF THE DEDUCTIVE SYSTEM

Martin-Löf's theory builds upon the Curry–Howard [11] isomorphism. That is, a proposition is identified with the *type* of its proofs. Given a proposition represented by the type A, proof corresponds to demonstrating that the type A is non-empty. Because of the constructive nature of the logic this involves constructing an *object* a that inhabits the type A. In programming terms types are identified with specifications and objects are identified with programs. As a consequence program and proof development are one and the same. Our presentation of the deductive system draws upon Backhouse's [2] explanation of Martin-Löf's theory.

2.1. Judgements

Assertions are expressed within the logic through four *judgement* forms. The first two judgements take the form:

$$A \ type \qquad A = B$$

The first states that A is a type while the second states that A and B are equal types. The remaining judgements relate objects and types:

$$a : A \qquad a = b : A$$

The first states that the object a inhabits the type A. The second states that a and b are equal objects in the type A.

Judgements which depend upon assumptions are called *hypothetical.* We adopt Backhouse's [2] scoping notation for representing hypothetical judgements. To illustrate, consider the following hypothetical judgement:

$$|[x_1 : X_1; \ldots; x_n : X_n \rhd a : A(x_1, \ldots, x_n)]|$$

Note that $|[$ and $]|$ delimit the scope of the variables $x_1, \ldots, x_n$ and that $\rhd$ separates assumptions from the conclusion. Collectively, the assumptions denote a context. The scoping notation extends to the level of derivations providing a block structured style of presentation.

[1] A shorter version of this paper appeared in the proceedings of the Seventh Austrian Conference on Artificial Intelligence 1991, Springer-Verlag, Informatik-Fachberichte 287.

2.2. Rules

Rules are presented in a natural deduction style [6]. A type is defined by four kinds of rules: *formation*, *introduction*, *elimination* and *computation*. To illustrate their structure we present the rules for the *List* type. Note that the first three kinds of rules have equality versions which are not presented for reasons of space.

The well-formedness of a type expression is prescribed by formation rules. The formation rule for the *List* type constructor takes the form:

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

Following Backhouse we call A and B *formation variables*. For an arbitrary type constructor Θ with formation variables $A_1, \ldots, A_m$ we will abbreviate the type expression $\Theta(A_1, \ldots, A_m)$ as $\Theta(\bar{A})$.

An introduction rule prescribes how *canonical* (data) objects are constructed. The *List* type constructor has two canonical constructors *nil* and :: and consequently two introduction rules:

$$\frac{A\ type}{nil : List(A)}\ List\text{-introduction}_{nil} \qquad \frac{u_1 : A \quad u_2 : List(A)}{u_1 :: u_2 : List(A)}\ List\text{-introduction}_{::}$$

u_1 and u_2 are called *introduction variables*. Because u_2 is a member of the type *List*(A) it is also referred to as a *recursive introduction variable*. For the arbitrary type constructor Θ with k canonical constructors there exists k introduction rules. The ith introduction rule defines the canonical constructor θ_i. Assuming θ_i is non-nullary then it will have introduction variables $u_{i1}, \ldots, u_{in_i}$ which we abbreviate as $\bar{u}_i$. In addition we use B_{ij} to denote the type of u_{ij}. Note that if u_{ij} is recursive then B_{ij} will denote the type $\Theta(\bar{A})$.

Each type constructor has an unique most general *non-canonical* (program) object constructor which is introduced by its elimination rules. The elimination rule for the *List* type takes the form:

$$\frac{\begin{array}{l} |[y : List(A) \rhd D(y)\ type]| \\ x : List(A) \\ z_1 : D(nil) \\ |[u_1 : A;\ u_2 : List(A);\ v_1 : D(u_2) \rhd z_2(u_1, u_2, v_1) : D(u_1 :: u_2)]| \end{array}}{elim_{List}(x, z_1, z_2) : D(x)}\ List\text{-elimination}$$

Note that z_2 denotes the abstraction $[u_1, u_2, v_1]z_2(u_1, u_2, v_1)$ and that u_1 and u_2 correspond to the introduction variables. For each recursive introduction variable there exists a corresponding *elimination variable*, in this case denoted by v_1. The $elim_{List}$ constructor enables us to define primitive recursive functions over the *List* type. For example, list concatenation may be defined as follows

$$x <> y \equiv elim_{List}(x, y, [u_1, u_2, v_1]u_1 :: v_1)$$

The non-canonical constructor associated with the arbitrary type Θ is $elim_\Theta$. Assuming that Θ has k canonical constructors then Θ-elimination introduces a non-canonical object of the form $elim_\Theta(x_1, z_1, \ldots, z_k)$ where x belongs to $\Theta(\bar{A})$. If θ_i is non-nullary then z_i denotes an abstraction of the form $[\bar{u}_i, \bar{v}_i]z_i(\bar{u}_i, \bar{v}_i)$ where $\bar{u}_i$ and $\bar{v}_i$ denote introduction and elimination variables, respectively.

The evaluation of a non-canonical constructor is defined by computation rules. For each canonical constructor there exists a corresponding computation rule. Consequently, the *List* type has two computation rules:

$$\frac{\begin{array}{l} |[y : List(A) \rhd D(y)\ type]| \\ z_1 : D(nil) \\ |[u_1 : A;\ u_2 : List(A);\ v_1 : D(v) \rhd z_2(u_1, u_2, v_1) : D(u_1 :: u_2)]| \end{array}}{elim_{List}(nil, z_1, z_2) = z_1 : D(nil)}\ List\text{-computation}_{nil}$$

$$\frac{\begin{array}{l} |[y : List(A) \rhd D(y)\ type]| \\ h : A \\ t : List(A) \\ z_1 : D(nil) \\ |[u_1 : A;\ u_2 : List(A);\ v_1 : D(v) \rhd z_2(u_1, u_2, v_1) : D(u_1 :: u_2)]| \end{array}}{elim_{List}(h :: t, z_1, z_2) = z_2(h, t, elim_{List}(t, z_1, z_2)) : D(h :: t)}\ List\text{-computation}_{::}$$

3. DERIVING PROPERTIES OF DATA TYPES

We now exploit the uniform structure of the deductive system presented in Section 2. Given an arbitrary data type constructor we show how uniqueness, closure and cancellation properties may be expressed within the theory. For each property, inspection of the corresponding derivation reveals a general structure. This general structure provides the basis for mechanizing the derivation of these properties. This work provides the foundations for the proof procedure for negation presented in Section 4. A detailed account of this work appears in Ireland [12].

3.1. Uniqueness Properties

An uniqueness property states that distinct object constructors build distinct object expressions. Consider, for instance, the canonical constructors *nil* and :: associated with *List* type. The corresponding uniqueness property is expressed by the type[2]

$$(\forall\, h:T)(\forall\, t:List(T))\neg(h::t =_{List(T)} nil)$$

The equality type $(a =_A b)$ expresses the proposition 'a and b are equal objects in the type A'. Negation is not a primitive of the theory. The proposition $\neg A$ is defined as $A \rightarrow false$. Note that we use *false* and *true* as abbreviations for the empty type and the one element type, respectively. To prove $\neg(h::t =_{List(T)} nil)$ we require a function which maps an arbitrary object in $(h::t =_{List(T)} nil)$ into an object in *false*. That is, we must show that $(h::t =_{List(T)} nil)$ is contradictory. The required contradiction is achieved by assuming the equality

$$h::t =_{List(T)} nil$$

from which an equality judgement of the form

$$elim_{List}(h::t, false, [u_1, u_2, v_1]true)$$
$$= elim_{List}(nil, false, [u_1, u_2, v_1]true): U_1$$

follows by the rules defining *List*, U_1 and the general rules of the theory. Note that U_1 denotes the 'type of types', the first in a cumulative hierarchy of universes. By transitivity and the computation rules for the *List* type a judgement is derived of the form

$$true = false : U_1$$

from which it is trivial to derive an object in *false*.

The above reasoning exhibits a general structure. To demonstrate this, consider an arbitrary data type constructor Θ. Assuming that Θ has k canonical object constructors $\theta_1, \ldots, \theta_k$, then for each pair of distinct constructors θ_i and θ_j there exists a uniqueness property which can be expressed by the type:

$$(\forall\, p_{i1}:B_{i1})\ldots(\forall\, p_{in_i}:B_{in_i})(\forall\, _{pj1}:B_{j1})\ldots(\forall\, p_{jn_j}:B_{jn_j})$$
$$\neg(\theta_i(\bar{p}_i) =_{\Theta(\bar{A})} \theta_j(\bar{p}_j))$$

To establish this property we assume the equality

$$(\theta_i(\bar{p}_i) =_{\Theta(\bar{A})} \theta_j(\bar{p}_j))$$

and derive, using the rules for Θ and U_1, an intermediate equality judgement of the form

$$elim_\Theta(\theta_i(\bar{p}_i), z_1, \ldots, z_k) = elim_\Theta(\theta_j(\bar{p}_j), z_1, \ldots, z_k): U_1$$

We take z_i to be the abstraction $[\bar{x}]true$ and we take all the other z's, including in particular z_j (z_i and z_j are the only relevant ones), to be $[\bar{y}]false$. The computation rules for Θ yields the required contradiction:

$$true = false : U_1$$

3.2. Closure Properties

A universal closure property expresses completely the structure of the canonical objects defined by a type constructor. Such a property can be expressed, in general, as a universally quantified disjunction, where each disjunct corresponds to a distinct canonical constructor. For instance, the universal closure property for the *List* type may be expressed as

$$(\forall\, x:List(T))((x =_{List(T)} nil)$$
$$\vee (\exists\, h:T)(\exists\, t:List(T))(x =_{List(T)} h::t)) \qquad (1)$$

Alternatively, the closure may be expressed using a single equality type

$$(\forall\, x:List(T))(elim_{List}(x, nil, [u_1, u_2, v_1]u_1::u_2) =_{List(T)} x).$$

Although this formulation is more uniform we selected (1) because the resulting justification provides a function for discriminating between distinct canonical forms. We make use of this function in Section 4.2 where rules for reasoning about equalities are developed.

A proof of (1) proceeds as follows. Firstly, we assume

$$x: List(T) \qquad (2)$$

Secondly, in the *nil* case we construct the judgement[3]

$$eq:(nil =_{List(T)} nil)$$

from which we derive

$$inject_1(eq):(nil =_{List(T)} nil)$$
$$\vee (\exists\, h:T)(\exists\, t:List(T))(nil =_{List(T)} h::t) \qquad (3)$$

Secondly, in the :: case we introduce assumptions for the introduction variables associated with *List*-introduction$_{::}$:

$$u_1:T$$
$$u_2:List(T)$$

Since u_2 is a recursive introduction variable we introduce an additional assumption of the form:

$$v_1:(x =_{List(T)} nil)$$
$$\vee (\exists\, h:T)(\exists\, t:List(T))(u_2 =_{List(T)} h::t))$$

[2]In the interests of wider readability the Π and Σ type constructors have been translated into the $\forall$ and $\exists$ quantifiers which they are identified with, respectively.

[3]The constant *eq* is the trivial construction demonstrating equality.

In this context we can derive:

$$eq : (u_1 :: u_2 =_{List(T)} u_1 :: u_2)$$

and by $\exists$-introduction derive:

$$\langle u_1, \langle u_2, eq\rangle\rangle : (\exists\, h : T)(\exists\, t : List(T))(u_1 :: u_2 =_{List(T)} h :: t)$$

By $\vee$-introduction we get:

$$inject_2(\langle u_1, \langle u_2, eq\rangle\rangle) : (u_1 :: u_2 =_{List(T)} nil) \vee (\exists\, h : T)(\exists\, t : List(T))(u_1 :: u_2 =_{List(T)} h :: t) \quad (4)$$

Finally, from (2), (3) and (4) by *List*-elimination followed by $\forall$-introduction we get:

$$\lambda x.closure_{List}(x) : (\forall\, x : List(T))(x =_{List(T)} nil) \vee (\exists\, h : T)(\exists\, t : List(T))(x =_{List(T)} h : t)$$

where $closure_{List}(x)$ is an abbreviation for the object

$$elim_{List}(x, inject_1(eq), [u_1, u_2, v_1]inject_2(\langle u_1, \langle u_2, eq\rangle\rangle))$$

The structure of the derivation presented above is determined completely by the rules which define the *List* type. To demonstrate the general structure of the above reasoning consider again the type constructor Θ. The closure property for Θ may be expressed by the type

$$(\forall\, x : \Theta(\bar{A}))P_1(x) \vee \cdots \vee P_k(x)$$

Note that the ith disjunct ($1 \leqslant i \leqslant k$) expresses the structure of the canonical constructor defined by Θ-introduction$_{\theta_i}$. Proof is by induction on x giving rise to k cases. We begin by assuming

$$x : \Theta(\bar{A})$$

A proof of the ith case involves constructing an injection into the type

$$P_1(\theta_i(\bar{u}_i)) \vee \cdots \vee P_k(\theta_i(\bar{u}_i))$$

where $P_j(\theta_i(\bar{u}_i))$ denotes, for $j = 1, \ldots, k$, the type

$$(\exists\, a_{i1} : B_{i1}) \ldots (\exists\, a_{in_i} : B_{in_i}) (\theta_i(u_{i1}, \ldots, u_{in_i}) =_{\Theta(\bar{A})} \theta_i(a_{i1}, \ldots, a_{in_i}))$$

Note that the case where θ_i is nullary is a simplification of the general case presented here. Construction of an object in this type takes place in a context C_i which is constructed from assumptions of the form:

$$u_{ir} : B_{ir}$$

$$v_{it} : P_1(u_{is}) \vee \cdots \vee P_k(u_{is})$$

Note that r and s range over the introduction and recursive introduction variables respectively while t ranges over the elimination variables associated with Θ. Within this context a judgement is constructed of the form

$$\begin{aligned} &|[C_i \\ &\triangleright \langle u_{i1}, \ldots \langle u_{in_i}, eq\rangle \ldots\rangle \\ &\quad : (\exists\, a_{i1} : B_{i1}) \ldots (\exists\, a_{in_i} : B_{in_i}) \\ &\quad (\theta_i(u_{i1}, \ldots, u_{in_i}) =_{\Theta(\bar{A})} \theta_i(a_{i1}, \ldots, a_{in_i})) \\ &]| \end{aligned}$$

Denoting the ith injection operator by $inject_i$ then the process described above generates k judgements of the form

$$|[C_i \triangleright inject_i(\langle u_{i1}, \ldots \langle u_{in_i}, eq\rangle \ldots\rangle) : P_1(\theta_i(\bar{u}_i)) \vee \cdots \vee P_k(\theta_i(\bar{u}_i))]|$$

By an application of the Θ-elimination rule followed by $\forall$-introduction the closure property is established:

$$\lambda x.closure_{\Theta}(x) : (\forall\, x : \Theta(\bar{A}))P_1(x) \vee \cdots \vee P_k(x)$$

Note that $closure_{\Theta}(x)$ is an abbreviation for the object $elim_{\Theta}(x, z_1, \ldots, z_k)$. In the cases where θ_i is non-nullary z_i denotes the abstraction $[\bar{u}_i, \bar{v}_i]inject_i(\langle u_{i1}, \ldots \langle u_{in_i}, eq\rangle \ldots\rangle)$

3.3. Cancellation Properties

Cancellation properties express the fact that for two canonical objects to be equal implies that their component parts are also equal. For example, two non-empty lists to be equal implies that the first elements from each list are also equal. Such a cancellation property may be expressed by the type

$$(\forall\, a : T)(\forall\, c : T)(\forall\, b : List(T))(\forall\, d : List(T)) ((a :: b =_{List(T)} c :: d) \rightarrow (a =_T c)) \quad (5)$$

As is the case with uniqueness properties for the type *List*, the construction of an object in this type is completely determined by the rules defining the *List* type. In the context of the assumptions:

$$a : T$$

$$b : List(T)$$

$$c : T$$

$$d : List(T)$$

a proof of (5) may be constructed by assuming

$$x : (a :: b =_{List(T)} c :: d) \quad (6)$$

and then using the computation rules for the *List* type to construct judgements of the form:

$$eq : elim_{List}(a :: b, a, [u_1, u_2, v_1]u_1) =_T a \quad (7)$$

$$eq : elim_{List}(c :: d, a, [u_1, u_2, v_1]u_1) =_T c \quad (8)$$

$$eq : elim_{List}(w, a, [u_1, u_2, v_1]u_1) =_T elim_{List}(w, a, [u_1, u_2, v_1]u_1) \quad (9)$$

From (6) and (9), by substitution, we get:

$$eq : elim_{List}(a :: b, a, [u_1, u_2, v_1]u_1) =_T elim_{List}(c :: d, a, [u_1, u_2, v_1]u_1)$$

Together with (7) and (8), transitivity gives us

$$a =_T c$$

Finally, by $\rightarrow$-introduction, we discharge (6) to give

$$\lambda x.eq : (a :: b =_{List(T)} (c :: d) \rightarrow (a =_T c)$$

To demonstrate the generality of this reasoning consider the ith canonical constructor associated with Θ type constructor. The constructor θ_i has n_i associated cancellation properties and the jth cancellation property may be expressed by the type

$$
\begin{aligned}
&(\forall\, p_{i1}:B_{i1})\ldots(\forall\, p_{in_i}:B_{in_i})\\
&\quad(\forall\, q_{i1}:B_{i1})\ldots(\forall\, q_{in_i}:B_{in_i})\\
&\qquad(\theta_i(\bar{p}_i)=_{\Theta(\bar{A})}\theta_i(\bar{q}_i))\rightarrow(p_{ij}=_{B_{ij}}q_{ij})
\end{aligned}
$$

where j ranges over the introduction variables associated with θ_i. Establishing a cancellation property in general we construct a context from assumptions of the form:

$$p_{ij}:B_{ij}$$
$$q_{ij}:B_{ij}$$

Extending this context with the assumption

$$x:(\theta_i(\bar{p}_i)=_{\Theta(\bar{A})}\theta_i(\bar{q}_i))$$

we can construct judgements of the form

$$eq:elim_{\Theta}(\theta_i(\bar{p}_i),z_1,\ldots,z_k)=_{B_{ij}}p_{ij}$$
$$eq:elim_{\Theta}(\theta_i(\bar{q}_i),z_1,\ldots,z_k)=_{B_{ij}}q_{ij}$$
$$eq:elim_{\Theta}(w,z_1,\ldots,z_k)=_{B_{ij}}elim_{\Theta}(w,z_1,\ldots,z_k)$$

where z_i denotes an abstraction of the form $[\bar{u}_i,\bar{v}_i]u_{ij}$. By substitution we get

$$eq:elim_{\Theta}(\theta_i(\bar{p}_i),z_1,\ldots,z_k)=_{B_{ij}}elim_{\Theta}(\theta_i(\bar{q}_i),z_1,\ldots,z_k)$$

and transitivity yields

$$eq:(p_{ij}=_{B_{ij}}q_{ij})$$

Finally, by $\rightarrow$-introduction we get

$$\lambda x.eq:(\theta_i(\bar{p}_i)=_{\Theta(\bar{A})}\theta_i(\bar{q}_i))\rightarrow(p_{ij}=_{B_{ij}}q_{ij})$$

4. A PROOF PROCEDURE FOR NEGATION

The construction of a formal proof typically involves a few key steps and many tedious ones. A principal objective of mechanizing formal proof is to bridge the gaps between the key steps. Proof procedures provide the basis for achieving this objective. In the context of program development within type theory proof obligations which have no computational content are prime candidates for the use of proof procedures since the structure of such proofs has no effect on the structure of the synthesized program.

Propositional equality is the simplest example of what has been described as this *mismatch* between programs and proofs. Another basic example is the proof of a negation. In terms of our intuitions as programmers, a proof of a negation denotes an error state. This is illustrated by considering Chisholm's [4] derivation of a parsing algorithm:

$$(\forall\, w:Word)Parse(w)\vee\neg Parse(w)$$

A program satisfying this specification is a function which maps an arbitrary object of type *Word* into an object in the disjunction $Parse(w)\vee\neg Parse(w)$. A left injection is generated in the case of a parsable word while a right injection denotes an error. Another example of this general form of specification is given in Ireland [12], where a table look-up function is derived. Both derivations were formally checked using machine assistance. In each case a significant proportion of the overall proof effort was taken up with proving negations.

This observation motivated the development of a proof procedure for negation which is presented below. Our approach to proving negations is based upon a process of *reduction* and *decomposition* at the level of equalities which builds upon the uniqueness, closure and cancellation properties described in Section 3. Given a negated goal $\neg A$ our process generates an AND/OR tree of logical consequences of A. If successful, then for each AND branch a contradiction is identified. Backward propagation of these contradictions yields a proof of $\neg A$.

4.1. Proving Negations in Type Theory

As mentioned in Section 3 a proof of a negation is constructed by establishing a contradiction. Given a proposition $\neg A$, and a list of assumptions C, proof proceeds by adding A, as an assumption, to C and showing that a contradiction follows. Contradictions arise in type theory either at the level of propositions or through the surrounding context. Propositional contradictions are based upon uniqueness properties, whereas contextual contradictions are established by arguing forwards from the assumptions.

4.2. Derived Rules for Reasoning about Equalities

In our procedure a reduction corresponds to either a *case separation* or an *evaluation* while a decomposition results in the breaking down of data objects into their component parts. Each operation is formalized as a set of derived rules of inference. We present schemata and outline the basis for mechanizing the construction of these rules.

4.2.1. Case Separation Rules

In order to illustrate case separation, consider the following equality type

$$\alpha(x<>y)=_S\beta$$

where $<>$ is the list concatenation operator. Note that α, β and y denote arbitrary terms, and x is a variable covered by an assumption $x:List(T)$. Case separation on x yields a disjunction which may be expressed as

$$
\begin{aligned}
&(\alpha(nil<>y)=_S\beta)\\
&\qquad\vee(\exists\, h:T)(\exists\, t:List(T))(\alpha(h::t<>y)=_S\beta)
\end{aligned}
$$

In general, the number of disjuncts corresponds to the number of introduction rules associated with the type over which the case separation is being performed. The general form of the derived case separation rule is:

$$\begin{array}{ll}
\text{P1} & \lvert[w:\Theta(\bar{A}) \vartriangleright (\alpha(w) =_T \beta)\ \mathit{type}]\rvert \\
\text{P2} & x:\Theta(\bar{A}) \\
\text{P3} & r:(\alpha(x) =_T \beta) \\
\hline
 & \mathit{case}_\Theta(x): R_1 \vee \ldots \vee R_k
\end{array}\ \Theta\text{-case}$$

P1 is a well-formedness premise while P2 specifies the object on which case separation is to be carried out. P3 is the judgement within which the case separation is to be performed. The derivation of the case separation rule for Θ is based upon the closure property for Θ defined in Section 3. Establishing the conclusion to the rule, given the premises, corresponds to building a method which constructs an injection into $R_1 \vee \ldots \vee R_k$ from an arbitrary object in $\Theta(\bar{A})$. The forms of $R_1, \ldots, R_k$ will be discussed shortly. Consider the injection into the ith disjunct. The injected value belongs to the type R_i, the structure of which is determined by the introduction rule for θ_i. Assuming that θ_i is a non-nullary constructor with n_i associated introduction variables then R_i will denote the type

$$(\exists\, a_{i1}:B_{i1})\ldots(\exists\, a_{in_i}:B_{in_i})(\alpha(\theta_i(a_{i1}\ldots a_{in_i})) =_T \beta)$$

To construct an object in this type we introduce an assumption of the form

$$y_i:(\exists\, a_{i1}:B_{i1})\ldots(\exists\, a_{in_i}:B_{in_i})(x =_{\Theta(\bar{A})} \theta_i(a_{i1}, \ldots, a_{in_i}))$$

Stripping off the existential quantification gives rise to a judgement of the form:

$$\begin{array}{l}
\lvert[y_i:(\exists\, a_{i1}:B_{i1})\ldots(\exists\, a_{in_i}:B_{in_i})(x =_{\Theta(\bar{A})} \theta_i(a_{i1}, \ldots, a_{in_i})) \\
\vartriangleright (x = \theta_i(\mathit{fst}(y_i), \ldots, \mathit{fst}(\ldots \mathit{snd}(y_i)\ldots)):\Theta(\bar{A}) \\
]\rvert
\end{array}$$

Taken together with P2 we can construct a judgement of the form:

$$\begin{array}{l}
\lvert[y_i:(\exists\, a_{i1}:B_{i1})\ldots(\exists\, a_{in_i}:B_{in_i})(x =_{\Theta(\bar{A})} \theta_i(a_{i1}, \ldots, a_{in_i})) \\
\vartriangleright r:(\alpha(\theta_i(\mathit{fst}(y_i), \ldots, \mathit{fst}(\ldots \mathit{snd}(y_i)\ldots))) =_T \beta) \\
]\rvert
\end{array}$$

from which the required value of the injection is derived using $\exists$-introduction:

$$\begin{array}{l}
\lvert[y_i:(\exists\, a_{i1}:B_{i1})\ldots(\exists\, a_{in_i}:B_{in_i})(x =_{\Theta(\bar{A})} \theta_i(a_{i1}, \ldots, a_{in_i})) \\
\vartriangleright \langle \mathit{fst}(y_i), \langle \ldots \langle \mathit{fst}(\ldots \mathit{snd}(y_i)\ldots), r\rangle \ldots\rangle\rangle \\
\qquad :(\exists\, a_{i1}:B_{i1})\ldots(\exists\, a_{in_i}:B_{in_i})(\alpha(\theta_i(a_{i1}, \ldots, a_{in_i})) =_T \beta) \\
]\rvert
\end{array}$$

From an object in R_i, the required injection is constructed by using the rule for $\vee$-introduction. Denoting the type of y_i by $P_i(x)$, then the process described above generates k judgements of the form

$$\begin{array}{l}
\lvert[y_i:P_i(x) \vartriangleright \mathit{inject}_i(\langle \mathit{fst}(y_i), \\
\qquad \langle \ldots \langle \mathit{fst}(\ldots \mathit{snd}(y_i)\ldots), r\rangle \ldots\rangle\rangle): R_1 \vee \ldots \vee R_k]\rvert
\end{array}$$

To combine these k judgements a function is required which yields an injection into $P_1(x) \vee \ldots \vee P_k(x)$, given an arbitrary object x in $\Theta(\bar{A})$. The universal closure property for Θ, as defined in Section 3, provides such a function. By $\forall$- and $\vee$-elimination the required conclusion is established. Note that in the conclusion to the derived rule $\mathit{case}_\Theta(x)$ is an abbreviation for the object $\mathit{elim}_{\vee_k}(\mathit{closure}_\Theta(x), z_1, \ldots, z_k)$ where $\mathit{elim}_{\vee_k}$ is an abbreviation for $k-1$ applications of $\mathit{elim}_\vee$ and z_i denotes the abstraction

$$[y_i]\mathit{inject}_i(\langle \mathit{fst}(y_i), \langle \ldots \langle \mathit{fst}(\ldots \mathit{snd}(y_i)\ldots), r\rangle \ldots\rangle\rangle)$$

4.2.2. Evaluation Rules

Evaluation in the context of the equality type

$$\alpha((h::t) <> y) =_S \beta$$

yields

$$\alpha(h::(t <> y)) =_S \beta.$$

The general form of the evaluation rule is:

$$\begin{array}{ll}
\text{P1} & \lvert[x:\Theta(\bar{A});\ y:D(x) \vartriangleright (\alpha(y) =_T \beta)\ \mathit{type}]\rvert \\
\text{P2} & b_{i1}:B_{i1}\ \ldots\ b_{in_i}:B_{in_i} \\
\text{P3} & \lvert[C_1 \vartriangleright z_1(\bar{u}_1, \bar{v}_1):D(\theta_1(\bar{u}_1))]\rvert \\
 & \vdots \\
 & \lvert[C_k \vartriangleright z_k(\bar{u}_k, \bar{v}_k):D(\theta_k(\bar{u}_k))]\rvert \\
\text{P4} & r:(\alpha(\mathit{elim}_\Theta(\theta_i(\bar{b}_i), z_1,, \ldots, z_k)) =_T \beta) \\
\hline
 & \mathit{eval}_{\theta_i}:(\alpha(z_i(\bar{b}_i, \bar{w}_i)) =_T \beta)
\end{array}\ \Theta\text{-eval}_{\theta_i}$$

where $\bar{w}_i$ is the vector of expressions corresponding to the step cases in a recursive definition. There is a direct correspondence between $\bar{v}_i$ and $\bar{w}_i$ which is explained in Backhouse [2]. The premises are divided into four parts. P1 is a well-formedness premise. P2 and P3 correspond to the major[4] premises of the computation rule for θ_i. The assumption lists $C_1, \ldots, C_k$ are constructed in the manner described in Backhouse [2] for elimination rules. P4 denotes the quality in which the evaluation is to be performed. The required conclusion is derived as follows: From the premises the evaluation of the subexpression $\mathit{elim}_\Theta(\theta_i(\bar{b}_i), z_1, \ldots, z_k)$ is achieved using the computation rule for θ_i. By substitution the resulting judgement taken together with a reflexive instance of P1 yields the equality judgement

$$(\alpha(\mathit{elim}_\Theta(\theta_i(\bar{b}_i), z_1, \ldots, z_k)) =_T \beta) = (\alpha(z_i(\bar{b}_i, \bar{w}_i)) =_T \beta)$$

This judgement and P4, by an application of the type equality rule, yields the required conclusion.

4.2.3. Decomposition Rules

A decomposition is possible when both sides of an equality are non-atomic canonical forms with matching outer structure. For example, consider the following equality type

$$a::b =_{List(T)} c::d$$

By decomposition this equality yields a conjunction of

[4]A major premise is any premise which is not a well-formedness premise.

the form

$$(a =_T c) \wedge (b =_{List(T)} d)$$

For an arbitrary canonical constructor θ_i, the number of conjuncts is determined by the number of introduction variables associated with the Θ-introduction$_{\theta_i}$ rule. The general form of the decomposition rule is:

$$\begin{array}{ll} \text{P1} & b_{i1} : B_{i1} \ldots b_{in_i} : B_{in_i} \\ \text{P2} & d_{i1} : B_{i1} \ldots d_{in_i} : B_{in_i} \\ \text{Pr} & r : (\theta_i(\bar{b}_i) =_{\Theta(\bar{A})} \theta_i(\bar{d}_i)) \\ \hline & decomp_{\theta_i} : (b_{i1} =_{B_{i1}} d_{i1}) \wedge \ldots \wedge (b_{in_i} =_{B_{in_i}} d_{in_i}) \end{array}$$

Θ-decomp$_{\theta_i}$

The premises are divided into three parts. P1 and P2 correspond to the major premises of the introduction rule for θ_i. P3 denotes the equality in which the decomposition is performed. If θ_i has n_i associated introduction variables, then there exist n_i cancellation properties as described in Section 3. These cancellation properties, taken together with the premises, give rise to n_i judgements of the form

$$eq : (b_{ij} =_{B_{ij}} d_{ij})$$

The required conclusion follows by n_{i-1} applications of $\wedge$-introduction.

4.3. A Framework for Searching for Contradictions

Our proof procedure builds upon the rules derived above for reasoning about equalities. It is similar to a tableau style proof procedure. The application of the derived rules is controlled by a procedure called *analyse*. Given a judgement of the form $|[B \rhd r : (\alpha =_T \beta)]|$ *analyse* constructs a judgement of the form $|[B \rhd r' : P]|$ where $(\alpha =_T \beta) \rightarrow P$. This construction process works in outline as follows: If either α or β is non-canonical then a reduction is performed. If a non-canonical expression contains a free variable in its recursive argument position then a case separation is performed. A non-canonical expression is open to evaluation if a canonical expression appears in the recursive argument position. If both α and β are non-atomic canonical forms with matching outer structure, then a decomposition is performed. Note that reductions may introduce choice points into the search space. We are primarily concerned with the general framework for supporting the search for contradictions rather than the actual search strategies. The application of *analyse* is controlled by a procedure called *contra-chk*. Assuming we wish to prove $\neg(\alpha =_T \beta)$ in the context of a list of assumptions C, then *contra-chk* first constructs trivial judgement

$$|[C; r : (\alpha =_T \beta) \rhd r : (\alpha =_T \beta)]|$$

to which *analyse* is applicable. Assuming the *analyse* constructs the judgement

$$|[C; r : (\alpha =_T \beta) \rhd r' : P]|$$

then by discharging the assumption denoted by r, a judgement is derived of the form

$$|[C \rhd \lambda r.r' : (\alpha =_T \beta) \rightarrow P]|$$

contra-chk is then recursively applied to P and C. This recursive process generates an AND/OR tree of logical consequences of the initial assumption:

$$r : (\alpha =_T \beta)$$

Depending on the structure of P, the search for a contradiction may branch. The structure of P may take one of four forms, each of which requires different treatment. Firstly, if P takes the form

$$(\alpha_1 =_{T_1} \beta_1)$$

and assuming that the analysis $(\alpha_1 =_{T_1} \beta_1)$ gives rise to $\bar{P}_1$, then the AND/OR tree is extended as follows:

$$\boxed{\begin{array}{l} |[C \\ \rhd \ \lambda r.\bar{r} : (\alpha =_T \beta) \rightarrow (\alpha_1 =_{T^1} \beta_1) \\ ]| \end{array}}$$

$$\downarrow$$

$$\boxed{\begin{array}{l} |[C \\ \rhd \ \lambda r.\bar{r} : (\alpha_1 =_{T^1} \beta_1) \rightarrow \bar{P}_1 \\ ]| \end{array}}$$

Secondly, P may take the form of a conjunction:

$$P_1 \wedge \cdots \wedge P_n$$

where each conjunct embodies an equality. A conjunction arises by decomposition. Assuming that $P_i (1 \leqslant i \leqslant n)$ is of the form $(\alpha_i =_{T_i} \beta_i)$, and that the analysis of $(\alpha_i =_{T_i} \beta_i)$ gives rise to $\bar{P}_i$, then the AND/OR tree is extended as follows:

$$\begin{tikzcd}
& \boxed{\begin{array}{l} |[C \\ \rhd \;\; \lambda r.decomp_{\theta_i}:(\alpha =_T \beta) \rightarrow (\alpha_1 =_{T_1} \beta_1) \wedge \cdots \wedge (\alpha_n =_{T_n} \beta_n) \\ ]| \end{array}} \arrow[dl] \arrow[dr] \arrow[d, phantom, "\vee"] & \\
\boxed{\begin{array}{l} |[C \\ \rhd \;\; \lambda r.\bar{r}_1:(\alpha_1 =_{T_1} \beta_1) \rightarrow \bar{P}_1 \\ ]| \end{array}} & \cdots & \boxed{\begin{array}{l} |[C \\ \rhd \; \lambda r.\bar{r}_n:(\alpha_n =_{T_n} \beta_n) \rightarrow \bar{P}_n \\ ]| \end{array}}
\end{tikzcd}$$

In order for a contradiction to be established it is sufficient for only one conjunct to be shown to be contradictory. Therefore, a $\vee$-node is generated. As mentioned earlier, a choice may exist in applying a reduction. The choice is accommodated within the framework by the introduction of a $\vee$-node. Thirdly, P may take the form of a disjunction:

$$P_1 \vee \cdots \vee P_k$$

where each disjunct embodies an equality. A disjunction arises from case separation on an object level variable. In order for a contradiction to be established, each disjunct must be shown to be contradictory, therefore a $\wedge$-node is introduced. Assuming that P_i $(1 \leqslant i \leqslant k)$ is of the form $(\alpha_i =_{T_i} \beta_i)$, where either α_i or β_i may depend upon the variable x, then case separation on x extends the AND/OR tree as follows:

$$\begin{tikzcd}
& \boxed{\begin{array}{l} |[C \\ \rhd \;\; \lambda r.case_{\Theta}(x):(\alpha =_T \beta) \rightarrow (\alpha_1 =_{T_1} \beta_1) \vee \cdots \vee (\alpha_k =_{T_k} \beta_k) \\ ]| \end{array}} \arrow[dl] \arrow[dr] \arrow[d, phantom, "\wedge"] & \\
\boxed{\begin{array}{l} |[C \\ \rhd \;\; \lambda r.\bar{r}_1:(\alpha_1 =_{T_1} \beta_1) \rightarrow \bar{P}_1 \\ ]| \end{array}} & \cdots & \boxed{\begin{array}{l} |[C \\ \rhd \; \lambda r.\bar{r}_k:(\alpha_k =_{T_k} \beta_k) \rightarrow \bar{P}_k \\ ]| \end{array}}
\end{tikzcd}$$

Finally, a case separation will usually generate existentially quantified equalities. For instance, consider the equality $(\alpha(x) =_T \beta)$ where the variable x is of type $\Theta(\bar{A})$. Case separation on x will, in general, generate a disjunction with k disjuncts where the ith disjunct takes the form

$$(\exists\, a_{i1} : B_{i1}) \ldots (\exists\, a_{in_i} : B_{in_i})(\alpha(\theta_i(a_{i1}, \ldots, a_{in_i})) =_T \beta)$$

Note that *analyse* is not directly applicable to this type because of the proof obligations introduced by the existential quantification. This problem is overcome by a *pre-analysis* step in which the existentially quantified equality is assumed from which follows the judgement

$$\begin{array}{l}
|[C \\
\rhd \; \lambda z.snd(\ldots snd(z) \ldots) \\
\qquad\qquad :(\forall\, z:(\exists\, a_{i1} : B_{i1}) \ldots (\exists\, a_{in_i} : B_{in_i})(\alpha(\theta_i(a_{i1}, \ldots, a_{in_i})) =_T \beta)) \\
\qquad\qquad\qquad\qquad (\alpha(\theta_i(fst(z), \ldots, fst(\ldots snd(z) \ldots))) =_T \beta) \\
]|
\end{array}$$

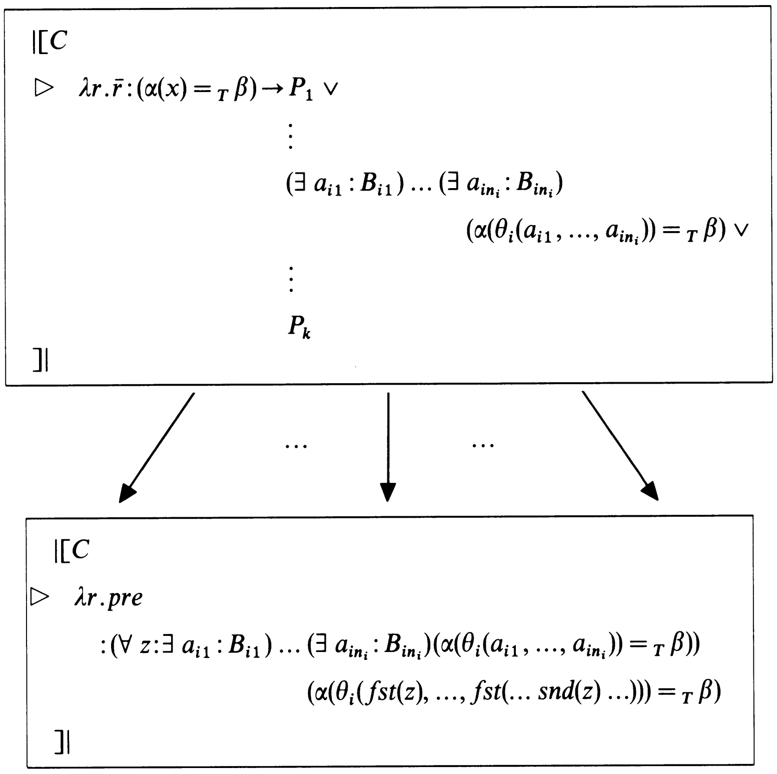
This dependent function removes the existential quantification. By extending the initial context with an assumption of the form

$$z:(\exists\, a_{i1}:B_{i1})\ldots(\exists\, a_{in_i}:B_{in_i})(\alpha(\theta_i(a_{i1},\ldots,a_{in_i}))=_T\beta)$$

thus enabling the search for a contradiction to proceed with the equality

$$\alpha(\theta_i(fst(z),\ldots,fst(\ldots snd(z)\ldots)))=_T\beta$$

The application of the pre-analysis step is reflected in the analysis tree by the introduction of an intermediate node:



$|[C$

$\rhd\ \lambda r.\bar{r}:(\alpha(x)=_T\beta)\rightarrow P_1 \vee$

$\vdots$

$(\exists\, a_{i1}:B_{i1})\ldots(\exists\, a_{in_i}:B_{in_i})$

$(\alpha(\theta_i(a_{i1},\ldots,a_{in_i}))=_T\beta)\vee$

$\vdots$

P_k

$]|$

$|[C$

$\rhd\ \lambda r.pre$

$:(\forall\, z:\exists\, a_{i1}:B_{i1})\ldots(\exists\, a_{in_i}:B_{in_i})(\alpha(\theta_i(a_{i1},\ldots,a_{in_i}))=_T\beta))$

$(\alpha(\theta_i(fst(z),\ldots,fst(\ldots snd(z)\ldots)))=_T\beta)$

$]|$

4.4. Proof Extraction

Given the arbitrary equality $(\alpha=_T\beta)$ and a list of assumptions C, a successful search for a contradiction generates a tree in which every leaf directly descendent from a $\wedge$-node gives rise to a contradiction, and at least one leaf directly descendent from a $\vee$-node gives rise to a contradiction. This tree structure embodies a proof of

$$|[C \rhd \lambda x.x:\neg(\alpha=_T\beta)]|$$

The process of proof extraction involves propagating the contradictions held at the leaf nodes back up through the tree structure. Given the implication

$$(\alpha=_T\beta)\rightarrow P$$

and assuming that by analysis of P a contradiction is identified

$$\neg P$$

then the required negation

$$\neg(\alpha=_T\beta)$$

can be derived by the application of the $\rightarrow$-elimination and $\rightarrow$-introduction rules. This propagation gives rise to a derived rule similar to the *modus tollens* rule of inference:

$$\frac{\begin{array}{l}A\ type\\ f:A\rightarrow B\\ g:\neg B\end{array}}{\lambda x.x:\neg A}$$

Note that for simplicity the derived proof object is replaced by the identity function $\lambda x.x$. If a negated type is non-empty, then the identity function will be present [1]. P may also take the form of a conjunction or a disjunction. If P takes the form of a conjunction

$$P_1\wedge\cdots\wedge P_n$$

then an implication follows of the form

$$(\alpha=_T\beta)\rightarrow P_1\wedge\cdots\wedge P_n$$

In order to establish that $(\alpha=_T\beta)$ is contradictory, it is sufficient to show that at least one of $P_1,\ldots,P_n$ is contradictory. Assuming that the analysis of P_i $(1\leqslant i\leqslant n)$ gives rise to a contradiction

$$\neg P_i$$

then a proof of the negation

$$\neg(\alpha=_T\beta)$$

can be derived using the $\wedge$- and $\rightarrow$-elimination rules together with the $\rightarrow$-introduction rule. The resulting derived rule takes the form:

$$\frac{\begin{array}{l} A\ type \\ f: A \rightarrow B_1 \wedge \cdots \wedge B_n \\ g: \neg B_i \end{array}}{\lambda x.x: \neg A}$$

Alternatively, P may take the form of a disjunction

$$P_1 \vee \cdots \vee P_k$$

giving rise to an implication of the form

$$(\alpha =_T \beta) \rightarrow P_1 \vee \ldots \vee P_k$$

Assuming that analysis of each disjunct gives rise to a contradiction then a proof of the negation

$$\neg(\alpha =_T \beta)$$

can be derived using the $\vee$- and $\rightarrow$-elimination rules together with the $\rightarrow$-introduction rule. The corresponding derived rule takes the form:

$$\frac{\begin{array}{l} A\ type \\ B_1\ type \ldots B_k\ type \\ f: A \rightarrow B_1 \vee \cdots \vee B_k \\ g_1: \neg B_1 \\ \vdots \\ g_k: \neg B_k \end{array}}{\lambda x.x: \neg A}$$

If P takes the form of a disjunction, then one or more of the disjuncts may involve existential quantification resulting in the application of a pre-analysis step. Our process for propagating contradictions must take account of this. As described earlier, if P is a disjunction which contains an existentially quantified disjunct, then the pre-analysis step constructs a dependent function type of the form

$$(\forall u: P)P'(u)$$

allowing the analysis to proceed with $P'(v)$, where v belongs to P. Assuming that $P'(v)$ gives rise to a contradiction, then a proof of $\neg P$ can be derived using the $\rightarrow$-elimination together with the $\rightarrow$- and $\forall$-introduction rules. The resulting derived rule takes the form

$$\frac{\begin{array}{l} P\ type \\ f: (\forall u: P)P'(u) \\ |[v: P \\ \rhd g: \neg P'(v) \\ ]| \end{array}}{\lambda x.x: \neg P}$$

4.5. Generalizing the Search for a Contradiction

In order to generalize the search for contradictions transformations are provided which convert a goal into the required negated equality form. These transformations involve pushing negations through quantifiers and logicals. The transformations operate both at the level of the goal type and at the level of the assumption types.

4.5.1. Goal Type Transformations

In general, a goal type G must be refined before *contra-chk* can be applied. This refinement process is carried out within the context of goal-directed proof and is achieved by the application of goal transformations. These transformations correspond to the subset of DeMorgan's laws for the logicals and quantifiers given by the following implications:

$$(\exists x: A)\neg B(x) \rightarrow \neg(\forall x: A)B(x)$$

$$(\forall x: A)\neg B(x) \rightarrow \neg(\exists x: A)B(x)$$

$$(\neg A \vee \neg B) \rightarrow \neg(A \wedge B)$$

$$(\neg A \wedge \neg B) \rightarrow \neg(A \vee B)$$

4.2.2. Assumption Type Transformations

Given a goal of the form

$$|[a_1: A_1; \ldots; a_n: A_n \rhd v: G]|$$

then A_i $(1 \leqslant i \leqslant n)$ denotes an assumption type. Assumption type transformations represent forwards inference and, like the goal type transformations, correspond to a subset of DeMorgan's laws for the logicals and quantifiers. In general the following implications are not constructively valid:

$$\neg(\forall x: A)B(x) \rightarrow (\exists x: A)\neg B(x) \qquad (10)$$

$$\neg(A \wedge B) \rightarrow (\neg A \vee \neg B) \qquad (11)$$

The missing laws, however, are not important. For example, consider the following hypothetical judgement

$$|[f: (\exists a: A)\neg B(a) \rhd v: G]|$$

Note that the assumption denoted by f corresponds to the consequent of (10). Such an assumption is too weak to be useful. It relates to a particular object v, for which $B(v)$ is contradictory, without specifying which object it is. Similarly, consider the hypothetical judgement

$$|[f: \neg A \vee \neg B \rhd v: G]|$$

where the assumption denoted by f corresponds to the consequent of (11). Again this assumption is too weak to be useful. It does not provide a method for determining which disjunct holds. The presence of the missing laws would, therefore, not enhance the generality of the proof procedure.

4.6. An Example

We illustrate the use of our proof procedure through its application to the proof of a negation arising from the synthesis of a table look-up function, the details of which can be found in Ireland [12].

4.6.1. Program Specification

Our table look-up function is specified by the type[5]

$$(\forall\, a:A)(\forall\, l:List(A \times B))$$
$$Member(a, l, A, B) \vee \neg Member(a, l, A, B) \qquad (12)$$

where A and B denote arbitrary types in U_1. *Member*[6] is defined as

$$Member \equiv [a, l, A, B](\exists\, h: List(A \times B))(\exists\, t: List(A \times B))$$
$$(\exists\, \mathrm{b}: \mathrm{B})(\mathrm{h} <> (\langle \mathrm{a}, \mathrm{b}\rangle :: \mathrm{t}) =_{List(A \times B)} \mathrm{l})$$

4.6.2. Proof of a Negation

In the course of satisfying (12) a goal type of the form

$$\neg Member(a, u :: v, A, B) \qquad (13)$$

is generated in a context which includes the following assumptions:

$$x: \neg(a =_A fst(u))$$
$$y: \neg Member(a, v, A, B)$$

Using the transformations outlined in Section 4.5 (13) may be reduced to

$$\neg(h <> (\langle a, b\rangle :: t) =_{List(A \times B)} u :: v) \qquad (14)$$

In addition to the assumptions for h, t and b the context is extended with the following assumption:

$$y': (\forall\, h': List(A \times B))(\forall\, t': List(A \times B))(\forall\, b': B)$$
$$\neg(\mathrm{h}' <> (\langle \mathrm{a}, \mathrm{b}'\rangle :: \mathrm{t}') =_{List(A \times B)} \mathrm{v})$$

(14) is of the required negated equality form and by case separation on h a disjunction is generated giving rise to the following two types:

$$nil <> (\langle a, b\rangle :: t) =_{List(A \times B)} u :: v \qquad (15)$$

$$(\exists\, p: A \times B)(\exists\, q: List(A \times B))$$
$$((p :: q) <> (\langle a, b\rangle :: t) =_{List(A \times B)} u :: v) \qquad (16)$$

Evaluation reduces (15) to

$$\langle a, b\rangle :: t =_{List(A \times B)} u :: v \qquad (17)$$

and by decomposition (17) is reduced to a conjunction, where the left conjunct

$$\langle a, b\rangle =_{A \times B} u$$

taken together with the assumption denoted by x gives rise to a contradiction. To deal with (16) the existential quantification is first removed introducing the assumption

$$z: (\exists\, p: A \times B)(\exists\, q: List(A \times B))$$
$$((p :: q) <> (\langle a, b\rangle :: t) =_{List(A \times B)} u :: v)$$

[5]Note that $\wedge$ is defined in terms of the cartesian product type $\times$.

[6]This specification of *Member* would be improved with the use of the subset type. However, the subset type was not supported by the underlying implementation of Type Theory [17] with which our proof procedure was developed.

thus allowing the search for a contradiction to proceed with

$$(fst(z) :: fst(snd(z))) <> (\langle a, b\rangle :: t) =_{List(A \times B)} u :: v \qquad (18)$$

By evaluation (18) is reduced to

$$fst(z) :: fst(snd(z)) <> (\langle a, b\rangle :: t)) =_{List(A \times B)} u :: v$$

which decomposes giving rise to a conjunction from which the right conjunct

$$fst(snd(z)) <> (\langle a, b\rangle :: t) =_{List(A \times B)} v$$

taken together with a specialization of the assumption denoted by y' yields a contradiction. Both branches of the case separation have been shown to give rise to contradictions. Backward propagation gives rise to a proof of (14). The proof of (13) follows by the correctness of the goal transformations.

4.7. Implementation

The proof procedure outlined has been implemented and integrated within a proof assistant [8] which is similar in design to the Nuprl [5] interactive proof environment. The goal transformations which generalize the applicability of the proof procedure are implemented using LCF [7] style tactics and tacticals. An implementation of Milner's type check algorithm [13] is used for satisfying well-formedness proof obligations.

5. RELATED WORK

An underlying assumption of the work presented here is that the theory can be extended mechanically to incorporate new data type constructors. Backhouse [2] proposes such a scheme for introducing user-defined extensions. This idea has been taken a stage further in the Calculus of Constructions where a mechanism for introducing inductive definitions has been implemented [18]. This work strengthens the argument for the basic approach we have adopted in developing our proof procedure.

Proofs of negations are an example of what has been called the *mismatch* between programs and proofs [3]. In Nuprl [5] the subset type is used to hide the computationally uninteresting component of an object expression. The need for the subset type is a consequence of the rigid type structure of Martin-Löf's theory. This was a motivation behind the flexible typing adopted by Henson [9] in his constructive set theory TK. In the Calculus of Constructions [15] and PX [10] a syntactic notion of *non-informative* propositions is introduced. These mechanisms, however, do not address the problem of satisfying computationally uninteresting proof obligations. In contrast to the approach adopted here Smith [19] argues for the use of nonconstructive methods in dealing with such proof obligations. By adding the law of the excluded middle to Martin-Löf's theory Smith is able to make use of classical logic in reasoning about programs. Whether the use of classical logic simplifies the theorem proving task remains to be seen. Smith admits to not knowing of any programming example where the correctness proof is considerably simplified by the use of classical logic.

6. CONCLUSIONS

In this paper we develop a proof procedure for negation based upon general properties of data type constructors. We outline the basis on which these properties can be derived mechanically. As a consequence our proof procedure may be extended uniformly to incorporate new data type constructors. The rich type structure of the theory provides an expressive specification language. The price paid for this expressiveness, however, is the loss of decidable type checking. With completeness not an achievable goal, our criteria for evaluating our proof procedure must be based on empirical study. The work presented here represents a starting point; a framework and an algorithm which is both tunable and extendible.

Acknowledgements

I am indebted to Alan Hamilton for numerous discussions during the course of this work and his detailed comments during the preparation of this paper. Paul Chisholm provided useful feedback on an earlier version of this paper for which I am very grateful. The work described here was funded by the *Carnegie Trust for the Universities of Scotland* and was carried out at Stirling University in partial fulfilment of the requirements for the Ph.D. degree.

REFERENCES

[1] R. C. Backhouse, Notes on Martin-Löf's theory of types, parts 1 and 2. *FACS FACTS* (1986).

[2] R. C. Backhouse, *On the Meaning and Construction of the rules in Martin-Löf's Theory of Types. Computing science notes cs 8606*, Department of Mathematics and Computing Science, University of Groningen (1986).

[3] R. C. Backhouse, Overcoming the mismatch between programs and proofs. In *Proc. of the Workshop in Programming Logic*, pp. 116–122, P. Dybjer, B. Nordström, K. Petersson, and J. M. Smith, (eds.). Marstrand, June (1987).

[4] P. Chisholm, Derivation of a parsing algorithm in Martin-Löf's theory of types. *Science of Computer Programming*, **8**, pp. 1–42 (1987).

[5] R. L. Constable *et al.*, *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, NJ.

[6] G. Gentzen, Investigations into logical deduction. In *The Collected Papers of Gerhard Gentzen*, pp. 68–213, M. E. Szabo (ed.). North-Holland, Amsterdam (1969).

[7] M. J. C. Gordon, R. Milner and C. P. Wadsworth, *Edinburgh LCF. Lecture Notes in Computer Science.* Springer-Verlag, Berlin (1979).

[8] A. G. Hamilton, *Program Construction in Martin-Löf Type Theory.* Technical report 24, Department of Computing Science, University of Stirling (1985).

[9] M. C. Henson, Program development in the constructive set theory TK. *Formal Aspects of Computing*, **1**, pp. 173–192 (1989).

[10] S. Hayashi and H. Nakano, *Px, A Computational Logic. Technical report*, Research Institute for Mathematical Science, Kyoto University, 1987.

[11] W. A. Howard, The formulae-as-types notion of construction. In *To H. B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 479–490, J. P. Seldin and J. R. Hindley (eds.). Academic Press, New York, (1980).

[12] A. Ireland, *Mechanization of Program Construction in Martin-Löf's Theory of Types.* PhD Thesis, Department of Computing Science, University of Stirling (1989).

[13] R. Milner, A theory of type polymorphism in programming languages. *J. Computer and System Sci.*, **17**, pp. 348–375 (1978).

[14] P. Martin-Löf, Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pp. 153–175, L. J. Cohen, J. Los, H. Pfeiffer, and K-P. Podewski (eds.). North-Holland, Amsterdam (1982).

[15] C. Mohring, Extracting f_w's programs from proofs in the Calculus of Constructions. In *Proc. of the 16th ACM Symposium of Principles of Programming Languages*, pp. 89–104 (1989).

[16] B. Nordström, K. Petersson and J. Smith, *Programming in Martin-Löf's Type Theory*, Clarendon Press, Oxford (1990).

[17] K. Petersson, *A Programming System for Type Theory. Lpm memo 21*, Department of Computer Science, Chalmers University of Technology, Göteborg (1982).

[18] F. Pfenning and C. Mohring, Inductively defined types in the calculus of constructions. Presented at: *Mathematical Foundations of Programming Language Semantics* (1989).

[19] J. Smith, On a nonconstructive type theory and program derivation. In *Mathematical Logic and its Applications*, pp. 331–340, D. G. Skordev (ed.). Plenum, New York (1987).

[20] S. Thompson, *Type Theory and Functional Programming.* Addison-Wesley, Reading, MA (1991).