1996

Theorem proving algorithms using lambda calculus and combinatory logic

Ramzy Master Rizkalla

University of Wollongong

UNIVERSITY OF WOLLONGONG
COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

This work is copyright. Apart from any use permitted under the Copyright Act 1968, no part of this work may be reproduced by any process, nor may any other exclusive right be exercised, without the permission of the author.

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.

Unless otherwise indicated, the views expressed in this thesis are those of the author and do not necessarily represent the views of the University of Wollongong.

Recommended Citation
Theorem Proving Algorithms Using Lambda Calculus and Combinatory Logic.

A thesis submitted in partial fulfilment of the requirements for the award of the degree

Master of Science (Honours)

from

THE UNIVERSITY OF WOLLONGONG

by

Ramzy M.A.Rizkalla
Gdp. of Science, University of Sydney.

Department of Mathematics
1996
The work of Martin Bunder [4] presents a simple version of the Ben-Yelles Algorithm as a tree. Given a formula of implicational logic, the algorithm determines the (possibly empty) set of $\lambda$-terms which have the given formula as a type. By the formulas as types isomorphism, it follows that this determines the set of its proofs. This is done in a finite numbers of steps. The algorithm which applies to intuitionistic logic, has been extended to some weaker logics such as BCI, BCK etc.

In this thesis we extend the Ben-Yelles algorithm to logics with connectives other than $\rightarrow$, by using the axiom $1 \vdash f \rightarrow A$, Peirce's law or both, i.e. to the logics with implication and intuitionistic negation, classical implicational logic and classical logic, as well as certain intermediate logics.
Originality

I hereby declare that this submission is my own work with the help of my supervisor Prof. Bunder and that to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of a university or other institute of higher learning, except where due acknowledgment or references are made in the text.

The original parts of this thesis are contained largely in Theorems 5.1, 5.2, 6.3, 6.4, 6.6, 7.5, 7.6, 7.7, 8.2, and 8.3, in some of the examples and the overall presentation.

In the following material, the word “we” was used stylistically in order to give the reader a sense of familiarity and involvement, and does not imply that the results joint work with others.

Ramzy Rizkalla

Wollongong, Mar.1996.
Acknowledgments

My sincerest thanks to

* Professor Martin W. Bunder, for his guidance and constant support throughout the course of this research. I would also like to commend his patience in working with me. Every discussion with him left me full of new ideas to explore.

* A. Prof. Nillsen and Dr. Prokop, for their advice and assistance during my study in Australia.

* The staff of the Department of Mathematics at the University of Wollongong, for their help and providing a pleasant workplace.

* Mr. Samuel Zaki, for giving me a chance to use his computer at my home, which was very helpfull.

* My wife Nadia, for being understanding, supportive, loving, caring and just plain beautiful.

Ramzy Rizkalla
# TABLE OF CONTENTS

## CHAPTER 1. Combinatory Logic

1.1 Introduction ........................................... 1
1.2 Elementary concepts ................................. 2
1.3 The abstraction algorithm ......................... 8
1.4 Combinators as functions and types as theorems .... 10
1.5 Combinator reductions and proof reductions ........ 14

## CHAPTER 2. Lambda Calculus

2.1 Introduction ........................................... 16
2.2 Elementary concepts and definitions ............... 18
2.3 Lambda terms as proofs and types as theorems .... 23
2.4 Long normal forms .................................. 26
2.5 Lambda reductions and proof reductions ........... 28

## CHAPTER 3. Translations

3.1 Introduction ........................................... 30
3.2 Relations between some bases of combinators ....... 31
3.3 Translation algorithms .............................. 32
3.4 The bases SK, BCK, BCI and BCIW ................. 36
3.5 Bases without C ...................................... 41
3.6 The bases BB'I and BB'TW ......................... 43
3.7 The base BB'IK ...................................... 47
CHAPTER 1

Combinatory Logic

§1.1 Introduction

Combinators are operators which manipulate terms by cancellation, duplication, bracketing and permutation. Combinatory logic started with Schönfinkel in 1924 (see [13] or [16]) and provides a way of doing logic and mathematics without using variables.

To motivate the combinators, consider the following example:
Define $A_{xy} = x + y$ and $M_{xy} = x.y$ (for all $x, y$) to be the addition and multiplication functions in arithmetic.
And then introduce a function $G$ defined by

$$G_{xyz} = (x.y) + (x.z)$$ (for all $x, y, z$)

i.e.

$$G_{xyz} = A(M_{xy})(M_{xz}).$$

Then the function $G$ is defined in terms of $A$, $M$ and variables. The point of combinatory logic is that it allows us to express $G$ without the use of any variables. Now, let’s assume that we have a function $L$ such that for all $x, y, z, u$ and $v$, we have

$$L_{xyzuv} = x(yzu)(yzv)$$

Then $G$ could be constructed within the function calculus of $L$, $A$ and $M$ by taking $G$ to be $LAM$.

$$G_{xyz} = LAM_{xyz} = A(M_{xy})(M_{xz})$$ as required.
Schönfinkel discovered that it is sufficient to introduce two functions (like $L$) in order to be able to define all other functions. These are the combinators $S$ and $K$, such that for any $X$, $Y$ and $Z$ we have

$$SXYZ = XZ(YZ)$$

and

$$KXY = X$$

If we identify $f(x_1, x_2)$, $(fx_1)x_2$ and $fx_1x_2$ then "the strong composition operator" $S$, has the property $(S(f,g))(x) = f(x,g(x))$ and the "constant-function" $K$, has the property $(K(a))(x) = a$.

A formal definition of combinator will be given below.

§2.2 Elementary Concepts

**Definition 1.1**: (Combinator)

(i) $K$ and $S$ are combinators.

(ii) If $X$ and $Y$ are combinators, so is $(XY)$.

(The operation in (ii) is called **application**.)

**Definition 1.2**: (Term)

(i) $K$, $S$, $x_1$, $x_2$,... are terms.

(ii) If $X$ and $Y$ are terms, so is $(XY)$.

**Notation** :

(1) "X", "Y", "Z", ... denote arbitrary terms.

(2) We use association to the left.

This means, for example, that $XYZW$ stands for $(((XY)Z)W)$. 
(3) If \( M, N, P \) and \( Q \) are combinators, then

(i) \( M \equiv N \) will mean that \( M \) is the same as \( N \).

(ii) If we have \( (M N) \equiv (P Q) \), then \( M \equiv P \) and \( N \equiv Q \).

**Definition 1.3**: (Occurs)

The term \( P \) occurs in \( Q \) (or \( P \) is a part of \( Q \), or \( Q \) contains \( P \)) is defined by induction on \( Q \), thus:

(i) \( P \) occurs in \( P \);

(ii) If \( P \) occurs in \( M \) or in \( N \), then \( P \) occurs in \( (M N) \).

The set of all variables occurring in \( M \) is called \( \text{FV}(M) \).

(The phrase "\( P \) occurs in \( M \)" may be written "\( P \in M \)").

**Definition 1.4**: (Substitution)

For any \( M, N, x \), define \( [N/x]M \) to be the result of substituting \( N \) for every occurrence of \( x \) in \( M \). The precise definition is by induction on \( M \), as follows:

(i) \( [N/x]x \equiv N \);

(ii) \( [N/x]a \equiv a \) where \( a \) is an atom and \( a \neq x \);

(iii) \( [N/x](P Q) \equiv ([N/x]P) ([N/x]Q) \);

For mutually distinct \( x_1, \ldots, x_n \) and for any \( N_1, \ldots, N_n \), we similarly define \( [N_1/x_1, \ldots, N_n/x_n]M \) to be the result of simultaneously substituting \( N_1 \) for \( x_1, N_2 \) for \( x_2, \ldots, N_n \) for \( x_n \) in \( M \).

**Example 1.1**:

\[
[xyz/x](yzx) \equiv ([xyz/ x]y) ([xyz/ x]x) ([xyz/ x]z) \\
\equiv y(xyz)z.
\]

(Using (iii), (ii), (i) and (ii) respectively.)
Definition 1.5 (reduction)

The relation $X \rightarrow Y$ (X reduces to Y) is defined as follows:

(K) $\text{KXY} \rightarrow X$

(S) $\text{SXYZ} \rightarrow \text{XZ(YZ)}$

(ρ) $X \rightarrow X$

(μ) $X \rightarrow Y \Rightarrow UX \rightarrow UY$

(ν) $X \rightarrow Y \Rightarrow XU \rightarrow YU$

(τ) $X \rightarrow Y \text{ and } Y \rightarrow Z \Rightarrow X \rightarrow Z$

Definition 1.6 (Weak Equality)

We say that $X = Y$ holds if this statement can be deduced from the axioms (K), (S), (ρ) and the rules (μ), (ν), (τ) of Definition 1.5 with ‘$\equiv$’ instead of ‘$\rightarrow$’, together with the rule $X=Y \Rightarrow Y=X$.

Now, by using S and K we can build (or define) further useful combinators, such as $I, B, C, T, B', W$ and $S'$ as follows:

Examples 1.2:

(i) The identity operator $I \equiv \text{SKK}$

If $\rightarrow f$.

Proof: $I \equiv \text{SKKf}$

$\rightarrow \text{Kf(Kf)}$

$\rightarrow f$.

(ii) The composition operator $B \equiv \text{S(KS)K}$

$Bfgx \rightarrow f(gx)$. 
Proof: \( Bfgx = S(KS)Kfgx \)
\[ \rightarrow KSf(Kf)\ gx \]
\[ \rightarrow S(Kf)gx \]
\[ \rightarrow Kfx(gx) \]
\[ \rightarrow f(gx). \]

(iii) The permutation operator \( C \equiv S(BBS)(KK) \)
\( Cfxy \rightarrow fyx. \)

Proof: \( Cfxy = S(BBS)(KK)fxy \)
\[ \rightarrow BBSf(KKf)xy \]
\[ \rightarrow B(Sf)Kxy \]
\[ \rightarrow Sf(Kx)y \]
\[ \rightarrow fy(Kxy) \]
\[ \rightarrow fyx. \]

(iv) The combinator \( T \)
\( T \equiv B(B(CSI))(B(BK)(CI)) \)
\( Txyz \rightarrow yxz \)

Proof: \( Txyz = B(B(CSI))(B(BK)(CI))xyz \)
\[ \rightarrow B(CSI)(B(BK)(CI)x)yz \]
\[ \rightarrow CSI(B(BK)(CI)x)yz \]
\[ \rightarrow S(B(BK)(CI)x)yIz \]
\[ \rightarrow B(BK)(CI)xxyz(Iz) \]
\[ \rightarrow BK(CIx)yz(Iz) \]
\[ \rightarrow K(CIx)yz(Iz) \]
\[ \rightarrow CIxy(Iz) \]
\[ \rightarrow Iyx(Iz) \]
\[ \rightarrow yxz \)
(v) The reflex composition operator

\[ B'gf \Rightarrow g(fg) \]

**Proof:** \[ B'gf \equiv CBgf \]
\[ \Rightarrow Bgf \]
\[ \Rightarrow g(fg) \]

(vi) The “diagonalizing” operator

\[ Wfx \Rightarrow fxx. \]

**Proof:** \[ Wfx \equiv SS(KI)fx \]
\[ \Rightarrow Sf(KIf)x \]
\[ \Rightarrow fx(KIfx) \]
\[ \Rightarrow fx(Ix) \]
\[ \Rightarrow fxx. \]

(vii) The combinator

\[ S' \equiv B(BW)(BB'B) \]

\[ S'xyz \Rightarrow yz(xz) \]

**Proof:** \[ S'xyz \equiv B(BW)(BB'B)xyz \]
\[ \Rightarrow BW(BBB'x)yz \]
\[ \Rightarrow W(BBB'xy)z \]
\[ \Rightarrow BBB'xyzz \]
\[ \Rightarrow B(B'x)yzz \]
\[ \Rightarrow B'x(yz)z \]
\[ \Rightarrow yz(xz). \]

The combinators \( S \) and \( K \) are not the only combinators that may be taken as primitive functions, one possibility is to take the combinators

\( B, C, K \) and \( W \) \hspace{1cm} [ we can define \( S \equiv B(B(BW)C)(BB) \) ],

also \( B, B', K \) and \( W \) \hspace{1cm} [ we can define \( S \equiv B(BW)(B(B'B')B') \) ].
Notation:

**BCKW** - combinatory logic or just **BCKW**, will be combinatory logic with **B**, **C**, **K** and **W** as primitive combinators and with the reduction rules in (ii), (iii) and (vi) replacing (**S**).

Also, we can construct **weaker** systems of combinatory logic like **BCK, BCI, BCIW,**...,etc again with appropriate reduction rules from Examples 1.2. **BCK** is “weaker” than **SK**, in the sense that **B** and **C** can be defined using **S** and **K** but **S** can’t be defined using just **B**, **C** and **K** (see Curry and Feys [7] §5H.3). When we refer to the axioms for the members of **Q** later on, we will mean the appropriate ones of the reduction rules given above. **Q** will always be assumed to be a subset of \{**S**, **K**, **I**, **C**, **B**, **W**, **B'**, **S'**, **T**\}.

We now consider combinatory logics based on a set of combinators **Q** as well as other constants and variables.

**Definition 1.7** : (**Q** - terms)

Assume an infinite sequence of symbols called **variables**, **x**, **y**, **z**, **x_1**, **x_2**,... and a finite or infinite set of symbols called **constants**, including a set of basic combinators **Q** (Possibly **Q** = \{**S,K**\}.).

The set of **Q** - **terms** is defined inductively by :

(i) All variables and constants, including the elements of **Q**, are **Q** - **terms**.

(ii) If **X** and **Y** are **Q** - **terms**, then so is (**XY**).
An atom is a variable or constant. A closed term is a term containing no variables. A Q-combinator is a term whose only atoms are elements of Q.

Examples of SK-terms are:

\[ KSx(Kx)yz \quad \text{and} \quad S(KS)K. \quad (\equiv \text{combinator B}). \]

Definition 1.8 (Types)

Assume a sequence of symbols a, b, c,... called atomic types; then we define types as follows:

(i) Each atomic type is a type;
(ii) If \( \alpha \) and \( \beta \) are types, then \( \alpha \to \beta \) is a type.

§1.3 The abstraction algorithm

The following is an algorithm to find, given an expression (possibly involving \( x_1 \)), the SK-combinator which when applied to \( x_1 \) gives back the expression:

(i) \( [x_1]x_1 \equiv I \)
(k) \( [x_1]Y \equiv KY \quad \text{if} \quad x_1 \notin Y \)
(\( \eta \)) \( [x_1]Yx_1 \equiv Y \quad \text{if} \quad x_1 \notin Y \)
(s) \( [x_1]YZ \equiv S(KY)(KZ) \)

For the algorithm to give a unique "abstract" the clauses must be used in the order given i.e.\( (ik\eta s) \). (In the order \( (iks\eta) \), \( (\eta) \) would never be used.)
Combinatorial Completeness:
If for each term $M(X_1, X_2, \ldots, X_n)$ formed by application using zero or more occurrences of each of $X_1, X_2, \ldots, X_n$, there is a Q-combinator $Z$ such that

$$Z X_1 X_2 \ldots X_n > M(X_1, X_2, \ldots, X_n)$$

Q-combinators are said to be combinatorially complete.

As the abstraction algorithm applies to all terms, $\{S, K, I\}$ is combinatorially complete.

This combinator $Z$ can be defined as

$$Z = [x_1] [x_2] \ldots [x_n] M(x_1, x_2, \ldots, x_n)$$

which we abbreviate to $[x_1, x_2, \ldots, x_n] M(x_1, x_2, \ldots, x_n)$

**Theorem 1.1**

(i) $([x_1, x_2, \ldots, x_n] X) x_1 x_2 \ldots x_n > X$

(ii) $([x_1, x_2, \ldots, x_n] X) X_1 X_2 \ldots X_n > [X_1/ x_1, \ldots, X_n/ x_n] X$

**Proof:**


**Example 1.3:**

$$[x_1, x_2] x_1 x_2 x_2 = [x_1] S([x_2] x_1 x_2)([x_2] x_2)$$

$$= [x_1] S x_1 I$$

$$= S([x_1] S x_1)([x_1] I)$$

$$= SS(KI).$$

**Example 1.4:**

$$[x_1, x_2, x_3] x_1 x_3 x_2 = [x_1, x_2] S([x_3] x_1 x_3)([x_3] x_2)$$

$$= [x_1, x_2] S x_1 (K x_2)$$

$$= [x_1] S([x_2] (S x_1))(x_2) K x_2$$

$$= [x_1] S(K(S x_1)) K$$
\[
\begin{align*}
\varepsilon &= S([x_1]S(K(Sx_1)))([x_1]K) \\
\varepsilon &= S(S([x_1]S)([x_1]K(Sx_1)))(KK) \\
\varepsilon &= S(S(KS)(S([x_1]K)([x_1]Sx_1)))(KK) \\
\varepsilon &= S(S(KS)(S(KK)S))(KK). \quad (= C)
\end{align*}
\]

§1.4 Combinators as functions and types as theorems

If \( X \in \alpha \) (we will also write \( X : \alpha \)) and \( Y \in \beta \) in the usual set theoretic sense,

then \( KXY = X \in \alpha \).

So \( KX \) is a function from \( \beta \) into \( \alpha \)

i.e. \( KX : \beta \to \alpha \)

but then \( K \) is a function from \( \alpha \) into \( \beta \to \alpha \)

i.e. \( K : \alpha \to (\beta \to \alpha) \).

If \( Z : \alpha, X : \alpha \to (\beta \to \gamma) \) and \( Y : \alpha \to \beta \)

then \( XZ : \beta \to \gamma \) and \( YZ : \beta \) and so \( XZ(YZ) : \gamma \)

Now since \( SXYZ = XZ(YZ) : \gamma \),

\( SXY : \alpha \to \gamma \)

and \( SX : (\alpha \to \beta) \to (\alpha \to \gamma) \)

and so \( S : (\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma)) \).

These sets will be denoted by expressions called types formed from type-variables \( a, b, c, ... \) by the rule that if \( \alpha \) and \( \beta \) are types then \( (\alpha \to \beta) \) is a type.
The application formation rule for terms with types, illustrated above, is as follows:

\[ \rightarrow_e \quad \text{If } X : \alpha \rightarrow \beta \quad \text{and} \quad Y : \alpha \quad \text{then} \quad (XY) : \beta. \]

This rule is part of the formal system \( TA_c \) of type assignment. The above type assignments to \( K \) and \( S \) are axioms of \( TA_c \).

(For details, see [12] definition 14.5.)

For example:

\[ S : (\alpha \rightarrow ((\beta \rightarrow \alpha) \rightarrow \alpha)) \rightarrow (\alpha \rightarrow (\beta \rightarrow \alpha)) \rightarrow (\alpha \rightarrow \alpha) \]

and

\[ K : \alpha \rightarrow ((\beta \rightarrow \alpha) \rightarrow \alpha) \]

so

\[ SK : (\alpha \rightarrow (\beta \rightarrow \alpha)) \rightarrow (\alpha \rightarrow \alpha) \]

and as

\[ K : \alpha \rightarrow (\beta \rightarrow \alpha) \]

so

\[ SKK = I : \alpha \rightarrow \alpha. \]

What we have above is a Hilbert style proof of the intuitionistic implicational theorem \( \alpha \rightarrow \alpha \) from axioms which are substitution instances of the types of \( K \) and \( S \). The whole proof is represented by the single combinator \( SKK \). We have the following theorem.

**Theorem 1.2**

(i) Every combinator that has a type, has one which is a theorem of intuitionistic implicational logic.

(ii) Every theorem of intuitionistic implicational logic is the type of at least one combinator that represents a Hilbert style proof of that theorem.

**Proof:**

(i) Any combinator is built up from \( K \)'s and \( S \)'s by application and the type of the combinator is built up by rule \( \rightarrow_e \) above from the types of \( K \)
and S. The part of this derivation dealing with the types only is an intuitionistic proof of the formula represented by the type.

(ii) Every theorem of intuitionistic implicational logic is derivable by modus ponens from the axioms which are the types of K and S. Hence a combinator representing this proof can be constructed with one application for each use of modus ponens.

**Example 1.5 :**

By the algorithm above, we have

\[ [x_1, x_2]x_2x_1 = [x_1]S([x_2]x_2)([x_2]x_1) \]
\[ = [x_1]SI(Kx_1) \]
\[ = S([x_1]SI)([x_1]Kx_1) \]
\[ = S(K(SI))K. \]

Now, if \( x_2x_1 : \alpha \) and \( x_1 : \beta \) for some set \( \beta \), then \( x_2 : \beta \to \alpha \) and thus, if \( Lx_1x_2 = x_2x_1 \) then

\[ Lx_1 : (\beta \to \alpha) \to \alpha \]

and so \( L \equiv S(K(SI))K : \alpha \to (\beta \to \alpha) \to \alpha \)

Thus \( L \) represents a Hilbert style proof of \( \alpha \to (\beta \to \alpha) \to \alpha \).

**Notation :**

Parentheses will often be omitted from types, and the reader should restore them in such a way that, for example,

\( (\alpha \to \beta \to \gamma) \to \alpha \to \beta \equiv ((\alpha \to (\beta \to \gamma)) \to (\alpha \to \beta)) \)

This restoration rule is called **association to the right**.
Example 1.6:

To find type of the combinator $B = S(KS)K$ consider

$$S : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$

(To simplify writing, we denote the type of this $S$ by $\alpha_i$, i.e. $S: \alpha_i$)

and $K : \alpha_i \to (\beta \to \gamma) \to \alpha_i$

then $KS : (\beta \to \gamma) \to \alpha_i$

now $S : ((\beta \to \gamma) \to \alpha_i) \to ((\beta \to \gamma) \to \alpha \to \beta \to \gamma) \to (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$

then $S(KS) : ((\beta \to \gamma) \to \alpha \to \beta \to \gamma) \to (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$

and so $B : (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$

Hence, we have a Hilbert style proof of $(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$ which is represented by the combinator $B$.

Note that in Chapter 4, we will show how we can find natural deduction style proofs of implicational formulas by using the Ben - Yelles algorithm and by using the translations in Chapter 3, we can translate these proofs back to combinators such as $B$ representing Hilbert- style proofs.

Definition 1.9: (Principal type, p.t.)

$\alpha$ is a p.t. of a Q - term $X$ if $\vdash X : \alpha'$ holds for a type $\alpha'$ when and only when $\alpha'$ is a substitution instance of $\alpha$.
Example 1.7:

I has p.t.s. a → a.


The principal types of the combinators considered in Examples 1.2 are:

- **I**: a → a
- **K**: a → b → a
- **S**: (a → b → c) → (c → b) → a → c
- **B**: (a → b) → (c → a) → c → b
- **C**: (a → b → c) → b → a → c
- **W**: (a → a → b) → a → b
- **B'**: (a → b) → (b → c) → a → c
- **S'**: (a → b) → (a → b → c) → a → c

All substitution instances of these will be available as axioms.

Note that not every combinator has a type, for example WW has no type. The first W must have type \( (α → α → β) → α → β \). The second must then have type \( α → α → β \equiv (γ → γ → δ) → γ → δ \) for some γ and δ. This gives \( α = (γ → γ → δ) = γ \), which is impossible.

§ 1.5 Combinator reductions and Proof reductions

We note that the reduction rules for K and S, when applied to typed terms, correspond to proof reductions or proof normalizations.

Now, using \( →_e \) we can see the normalization of the (K) and (S) reduction rules as follows:-
\[(K)\] \(KXY \rightarrow X\) corresponds to the normalization

\[
\begin{align*}
\text{K} &: \alpha \rightarrow \beta \rightarrow \alpha \\
\text{D}_1 \\
\rightarrow_e & \quad \text{KX} : \beta \rightarrow \alpha \\
\text{D}_2 \\
\rightarrow_e & \quad \text{KXY} : \alpha \\
\text{D}_3
\end{align*}
\]

where \(\text{D}_3\) is obtained from \(\text{D}_3\) by replacing appropriate occurrences of \(\text{KXY}\) by \(X\).

\[(S)\] \(SXYZ \rightarrow XZ(YZ)\) corresponds to the normalization

\[
\begin{align*}
\text{S} &: (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma \\
\text{D}_1 \\
\rightarrow_e & \quad \text{SX} : (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma \\
\text{D}_2 \\
\rightarrow_e & \quad \text{SXY} : \alpha \rightarrow \gamma \\
\text{D}_3 \\
\rightarrow_e & \quad \text{SXYZ} : \gamma \\
\text{D}_4
\end{align*}
\]

To

\[
\begin{align*}
\text{D}_1 & \\
\rightarrow_e & \quad \text{XZ} : \beta \rightarrow \gamma \\
\text{D}_3 \\
\rightarrow_e & \quad \text{YZ} : \beta \\
\text{D}_4
\end{align*}
\]

where \(\text{D}_4\) is obtained from \(\text{D}_4\) by replacing appropriate occurrences of \(\text{SXYZ}\) by \(XZ(YZ)\).
CHAPTER 2

Lambda Calculus

§2.1 Introduction

The \( \lambda \)-calculus is a formal system based on a function notation invented by Alonzo Church in the 1930's. The main feature is that, as for combinatory logic, it is higher order (i.e. it gives a systematic notation for operators whose input and output values may be other operators). It considers functions as unary rules, it allows the replacement of any higher order function of 2-places or more by a function of one-place.

We use \( \lambda x. X \) to mean the function whose value at \( x \) is \( X \).

For example \( \lambda x. \sin x = \sin \) and using \( (\lambda x. \sin x ) \ 2 \) rather than \( (\lambda x. \sin x ) \ (2) \) we have:

\[
(\lambda x. \sin x ) \ 2 = \sin 2
\]

Also \( (\lambda x. x^2 ) \ x = x^2 \)

and so \( (\lambda x. x^2 ) \ 2 = 4. \)

In everyday mathematics, an expression such as "2x + 5y" can be considered as defining either a function \( f \) of \( x \) or a function \( g \) of \( y \).

A convenient way to distinguish these two functions is to introduce a symbol "\( \lambda \)" and define

\[
f = \lambda x.2x + 5y \quad \text{and} \quad g = \lambda y.2x + 5y
\]

We say that prefixing "\( \lambda x \)" abstracts the function \( \lambda x.2x + 5y \) from the expression "2x + 5y". This gives us a systematic way of constructing,
for each expression involving “ x ”, a notion for the corresponding function of x. This is the starting point of the theory of \( \lambda \) - calculus.

For the function \( f \), we have as an example

\[
f_3 = 6 + 5y \quad \text{and} \quad f_5 = 10 + 5y
\]

In \( \lambda \) - notation these equations became

\[
(\lambda x.2x + 5y) 3 = 6 + 5y \quad \text{and} \quad (\lambda x.2x + 5y) 5 = 10 + 5y
\]

This notation can be extended to functions of more than one variable. For example, the expression “ 2x + 5y ” correspond to two functions \( f_1 \) and \( g_1 \) of two variables, defined by

\[
f_1(x, y) = 2x + 5y \quad \text{and} \quad g_1(y, x) = 2x + 5y
\]

Writing \( f_1xy \) for \( f_1(x, y) \) and \( g_1xy \) for \( g_1(x, y) \), this can be denoted by

\[
f_1 = \lambda xy.2x + 5y \quad \text{and} \quad g_1 = \lambda yx.2x + 5y
\]

However, we can avoid the need for a special notation for functions of several variables by using functions whose values are not numbers but functions. For example, instead of using the two place function \( f_1 \), we can represent \( f_1 \) by a one- place function \( f_1^* \) defined by

\[
f_1^* = \lambda x. (\lambda y.2x + 5y)
\]

For each number \( a \), we have \( f_1^* a = \lambda y.2a + 5y \)

Hence, for each pair of numbers \( a \) and \( b \), we have

\[
(f_1^* a) b = (\lambda y.2a + 5y) b = 2a + 5b = f_1 (a,b)
\]

thus, \( f_1^* \) can be viewed as “ representing ” \( f_1 \).

So we shall only need a \( \lambda \) - notation for functions of one variable.

A formal definition of lambda terms will be given below.
§2.2 Elementary Concepts and Definitions

We assume that we have an infinite sequence of variables (denoted by: x, y, z,...etc) and a finite (possibly empty) or infinite sequence of constants.

Definition 2.1 (λ-terms)

The set of expressions, called λ-terms, is defined inductively as follows:
(i) All variables and constants are λ-terms and are called atoms.
(ii) If M and N are any λ-terms, then (MN) is a λ-term and is called an application.
(iii) If M is any λ-term and x is any variable, then λx.M is a λ-term and is called an abstraction.

Examples of λ-terms:

(λx.xz), (x(λx.(λx.x))) and ((λy.y)(λz.(xy))).

Notation: "≡"

If M, N, P and Q are λ-terms, then
(1) M ≡ N will mean that M is the same term as N.
(2) If we have MN ≡ PQ, then M ≡ P and N ≡ Q.
(3) If λx.M ≡ λy.P, then x ≡ y and M ≡ P.

Parentheses and repeated λ's will often be omitted in such a way that, for example;
\[ \lambda xyz.N \equiv (\lambda x.(\lambda y.(\lambda z.N))) \]

This way of restoring parentheses is called **association to the right** for \( \lambda \)-abstractions.

However, we use **association to the left** for the applications such as

\[ NMPQ \equiv (((NM)P)Q). \]

**Definition 2.2** (Occurs)

The \( \lambda \)-term \( P \) **occurs** in \( Q \) (or \( P \) is subterm of \( Q \), or \( Q \) contains \( P \)) is defined by induction on \( Q \), thus:

(i) \( P \) occurs in \( P \);

(ii) if \( P \) occurs in \( M \) or in \( N \), then \( P \) occurs in \( (MN) \);

(iii) if \( P \) occurs in \( M \) or \( P \equiv x \), then \( P \) occurs in \( (\lambda x.M) \).

For the second example above, \( (x(\lambda x.(\lambda x.x))) \), the first occurrence of \( x \) is not the same as the other \( x \)'s. This difference is explained in the following definition.

**Definition 2.3** (Free and bound variables)

An occurrence of a variable \( x \) in a term \( P \) is **bound** if it occurs in a part of \( P \) of the form \( \lambda x.M \); otherwise it is **free**. If \( x \) has at least one free occurrence in \( P \), it is called a **free variable** of \( P \). The set of all such variables is called \( \text{FV}(P) \).

For example, in the lambda term \( P = vux (yz) (\lambda v.vy) \) the first \( v \) is free but the other \( v \)'s are bound, \( u, x, y \) are all free and so

\[ \text{FV}(P) = \{v, u, x, y\} \].
**Definition 2.4** (Substitution)

For any $M$, $N$, $x$, define $[N/x]M$ to be the result of substituting $N$ for every free occurrence of $x$ in $M$, and making any changes of bound variables needed to prevent variables free in $N$ from becoming bound in $[N/x]M$. The precise definition is by induction on $M$, as follows:

(i) $[N/x]x ≡ N$;
(ii) $[N/x]a ≡ a$ where $a$ is an atom and $a \neq x$;
(iii) $[N/x](PQ) ≡ ([N/x]P) ([N/x]Q)$;
(iv) $[N/x] (\lambda x.P) ≡ \lambda x.P$;
(v) $[N/x] (\lambda y.P) ≡ \lambda y.P$ if $x \notin \text{FV}(P)$;
(vi) $[N/x] (\lambda y.P) ≡ \lambda y.[N/x]P$ if $x \in \text{FV}(P)$ and $y \notin \text{FV}(N)$;
(vii) $[N/x] (\lambda y.P) ≡ \lambda z.[N/x] [z/y] P$ if $x \in \text{FV}(P)$ and $y \in \text{FV}(N)$.
(In (vii), $z$ is chosen to be the first variable such that $z \notin \text{FV}(NP)$.)

**Example 2.1** :

(a) $[y(\lambda x. x)/x] (y(\lambda y. y)) ≡ ([y(\lambda x. x)/x] y) ( [y(\lambda x. x)/x] (\lambda y.y))$ $\equiv y (\lambda y. y)$.

(Using (iii), then (ii) and (v).)

(b) $[(\lambda y.x) y] (\lambda y. x(\lambda x. x))$ $\equiv \lambda y. ( [(\lambda y.x) y] (\lambda x. x) )$

(Using (vi), as $x \in \text{FV}(x(\lambda x. x))$ and $y \notin \text{FV}(\lambda y. x))$ $\equiv \lambda y. ( [(\lambda y.x) y] (x) ) ( [(\lambda y.x) y] (\lambda x. x) )$

$\equiv \lambda y. (\lambda y.x) (\lambda x. x)$

(Using (iii), then (i) and (iv).)

**Definition 2.5** (Closed terms)

A closed term is a term with no free variables.
Examples of closed terms:

\[ \lambda x. x, \lambda xy. x, \lambda xyz. xz(yz), \lambda xy. yz(xy) \text{ and } \lambda xy. yyy. \]

Now, let a term \( P \) contain an occurrence of \( \lambda x. M \), and let \( y \notin \text{FV}(M) \).

The act of replacing this \( \lambda x. M \) by \( \lambda y. [y/x]M \) is called a change of bound variable in \( P \) (\( \alpha \)-conversion), i.e. we have the following rule

\[(\alpha) \quad \lambda x. M \rightarrow \lambda y. [y/x]M \quad \text{where } y \notin \text{FV}(M).\]

We say \( P \equiv \alpha Q \) if \( Q \) has been obtained from \( P \) by using a finite series of changes of bound variables.

Example 2.2:

\[ \lambda xyz. xz(yz) \equiv \lambda x. (\lambda y. (\lambda z. xz(yz))) \]
\[ \equiv \alpha \lambda x. (\lambda y. (\lambda v. xv(yv))) \]
\[ \equiv \alpha \lambda x. (\lambda u. (\lambda v. xv(uv))) \]
\[ \equiv \alpha \lambda w. (\lambda u. (\lambda v. wv(uv))) \]
\[ \equiv \lambda wuv. wv(uv). \]

Definition 2.6 (\( \beta \)-reduction)

Any term of the form \( (\lambda x. M)N \) is called a \( \beta \)-redex and the corresponding term \( [N/x]M \) is called its contractum, and we call \( (\lambda x. M)N \rightarrow_{\beta} [N/x]M \) a \( \beta \)-contraction.

We say \( P \beta \)-reduces to \( Q \), or \( P \rightarrow_{\beta} Q \)

if \( Q \) is obtained from \( P \) by a finite (perhaps empty) series of \( \beta \)-contractions and changes of bound variables.
Consider the multiple abstraction
\[ \lambda x. \lambda y. (x+y) \], the addition function.

Then
\[ (\lambda x. \lambda y. (x+y))5 \beta (\lambda y. (5+y)) \]
and so
\[ ((\lambda x. \lambda y. (x+y))5)6 \beta (\lambda y. (5+y))6 \]
\[ \beta 5+6 = 11 \]

**Lemma**: (i) \[ P \equiv_\alpha Q \implies \text{FV}(P) = \text{FV}(Q). \]
(ii) \[ P \beta Q \implies \text{FV}(P) \supseteq \text{FV}(Q) \]


To define \( \beta \)-reduction more formally we mention the following rules:

\( \eta \)
\[ \lambda x. Mx \beta M \]
\( \xi \)
\[ M \beta N \implies \lambda x. M \beta \lambda x. N \]

and also \( \mu \), \( v \), \( \tau \) and \( \rho \) from Definition 1.4.

A reduction that includes rule \( \eta \) is called \( \beta \eta \)-reduction and often denoted by \( \beta_\eta \). A reduction that does not include \( \eta \) is called \( \beta \)-reduction and denoted by \( \beta \). If only rule \( \eta \) is used we sometimes write \( \beta_\eta \).

**Example 2.3**:

(i) \( (\lambda x. y) M \beta [M/x] y \equiv y \)

(ii) \( (\lambda x. (\lambda y. yx)z) v \beta [v/x] ((\lambda y. yx) z) \equiv (\lambda y. yv) z \)

(iii) \( (\lambda x. x(xy)) M \beta M(My) \)

(iv) \( (\lambda x. xxy)(\lambda x. xxy) \beta (\lambda x. xxy)(\lambda x. xxy) y \)

etc
§2.3 Lambda terms as proofs and types as theorems

\(\lambda\)-terms can be assigned types (like combinators).

For example, if \(x \in Z\), \(y \in Z\) then \(x+y \in Z\), so \(\lambda y.x+y : Z \rightarrow Z\)

and so \(\lambda x.\lambda y.x+y : Z \rightarrow (Z \rightarrow Z)\).

The formal system of type-assignment used here is \(TA_\lambda\) of [12].

Definition 15.6. We regard \(\alpha\) - convertible terms as identical and so omit that definition's \(\alpha\) - rule. When assigning types to variables we assume that no variable receives more than one type (cf. [12], above Lemma 14.30). The two type - assignment rules are :

\[\rightarrow_e\quad \text{if } X : \alpha \rightarrow \beta \quad \text{and} \quad Y : \alpha \quad \text{then} \quad (XY) : \beta\]

\[\rightarrow_i\]

\[\begin{array}{l}
[x : \alpha] \\
\vdots \\
\vdots \\
Y : \beta \\
\end{array} \]

\[---------------------\]

\(\lambda x.Y : \alpha \rightarrow \beta\)

**Examples 2.4 :**

(i) If \(x \in \alpha\) and \(y \in \beta\) then \(\lambda y.x : \beta \rightarrow \alpha\)

and \(\lambda xy.x : \alpha \rightarrow (\beta \rightarrow \gamma)\)

(ii) If \(x \in \alpha \rightarrow (\beta \rightarrow \gamma)\), \(y \in \alpha \rightarrow \beta\) and \(z \in \alpha\) then \(xz : \beta \rightarrow \gamma\) and \(yz : \beta\)

so \(xz(yz) : \gamma\)

and \(\lambda z.xz(yz) : \alpha \rightarrow \gamma\) etc.
Now, we can put these examples in a tree form using $\rightarrow_e$ and $\rightarrow_i$ as follows:

For (i)

\[
\begin{align*}
\text{(1)} & \quad y : \beta \\
\text{(2)} & \quad x : \alpha \\
\rightarrow_i \quad & \quad \lambda y.x : \beta \rightarrow \alpha \\
\rightarrow_i \quad & \quad \lambda xy.x : \alpha \rightarrow (\beta \rightarrow \alpha)
\end{align*}
\]

\text{fig. 2.1}

For (ii)

\[
\begin{align*}
\text{(1)} & \quad x : \alpha \rightarrow (\beta \rightarrow \gamma) \\
\text{(3)} & \quad z : \alpha \\
\rightarrow_e \quad & \quad xz : \beta \rightarrow \gamma \\
\rightarrow_e \quad & \quad xz(yz) : \gamma \\
\rightarrow_i \quad & \quad \lambda z.xz(yz) : \alpha \rightarrow \gamma \\
\rightarrow_i \quad & \quad \lambda yz.xz(yz) : (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma) \\
\rightarrow_i \quad & \quad \lambda xyz.xz(yz) : (\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))
\end{align*}
\]

\text{fig. 2.2}

Now, what we have in figures 2.1 and 2.2, are the natural deduction style proofs of the intuitionistic implicational theorems $\alpha \rightarrow (\beta \rightarrow \alpha)$ and $((\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))$.

The proofs are represented by the lambda terms $\lambda xy.x$ and $\lambda xyz.xz(yz)$.

We have the following general theorem.
**Theorem 2.1:**

(i) Every lambda term that has a type, has one which is a theorem of intuitionistic implicational logic.

(ii) Every theorem of intuitionistic implicational logic is the type of at least one lambda term which represents a natural deduction style proof of that theorem.

**Proof:** Similar to that of Theorem 1.2.

In Chapter 4, through the Ben-Yelles algorithm, we will be able to find $\lambda$-terms, if any, that constitute proofs of a given formula of the intuitionistic implicational logic.

We can translate lambda terms to combinators and vice versa as follows:

\[
\begin{align*}
\lambda x.X & \Rightarrow [x] X \\
K & \Rightarrow \lambda xy.x \\
S & \Rightarrow \lambda xyz.xz(yz)
\end{align*}
\]

So, any natural deduction style proof in intuitionistic implicational logic can be translated to a Hilbert style proof and vice versa.

**Note that** it is not true that every lambda term has a type, for example $\lambda x.xx$ has no type.

To see this, suppose that $\lambda x.xx$ has a type $A \rightarrow B$, then we have $x : A$ and $xx : B$ and so $x : A$ and $x : A \rightarrow B$ which is impossible.
§ 2.4 Long normal forms

Definition 2.7: (β - normal form)

A term \( P \) which contains no \( \beta \) - redexes is called a term in \( \beta \) - normal form. If a term \( P \overset{\beta}{\rightarrow} Q \) in \( \beta \) - normal form, then \( Q \) is called a \( \beta \) - normal form of \( P \). Also we define an \( \beta \eta \) - normal form to be a term without \( \beta \eta \) - redexes.

The Church-Rosser Theorem (see Hindley and Seldin [10] App.1) implies that \( \beta \) - normal forms and \( \beta \eta \) - normal forms are unique.

Example 2.5:

From example 2.3(ii) we have \( (\lambda x.(\lambda y.yx)z)v \overset{\beta}{\rightarrow} zv \) and so \( zv \) is the \( \beta \) - normal form of \( (\lambda x.(\lambda y.yx)z)v \).

But in 2.3(iv) we have

\[
(\lambda x. xx)(\lambda x. xx) \overset{\beta}{\rightarrow} (\lambda x. xx)(\lambda x. xx) y \\
\overset{\beta}{\rightarrow} (\lambda x. xx)(\lambda x. xx) yy \ldots \text{etc}
\]

where these are the only possible reductions, so \( (\lambda x. xx)(\lambda x. xx) \) has no \( \beta \) - normal form.

Any \( \lambda \) - term in normal form is of the form \( \lambda x_1 \ldots x_n . x_i X_1 \ldots X_k \) where \( n \geq 0, k \geq 0 \) and \( X_1, \ldots, X_k \) are in normal form.

Definition 2.8: (long normal form)

A \( \lambda \) - term \( \lambda x_1 \ldots x_n . x_i X_1 \ldots X_k \) with type \( \tau \), \( n \geq 0 \) and \( k \geq 0 \), is said to be in long normal form with respect to \( \tau \) if it is in \( \beta \) - normal form, \( X_1, \ldots, X_k \) are in long normal form with respect to types \( \tau_1, \ldots, \tau_k \) and \( x_i \) has type \( \tau_1 \rightarrow \cdots \rightarrow \tau_k \rightarrow a \) for an atomic type \( a \).
Note that: (i) Any \( \lambda \)-term \( \lambda x_1 \ldots x_n . x_iX_1 \ldots X_k \), where \( X_1, \ldots, X_k \) are in long normal form and \( x_i \) has type \( \tau_1 \to \cdots \to \tau_m \to a \) where \( m > k \), has long normal form

\[ \lambda x_1 \ldots x_n . x_i \ldots x_{i_m-k} . x_iX_1 \ldots X_kx_i \ldots x_{i_m-k} . \]

where \( x_{i_1}, \ldots, x_{i_m-k} \) are not free in \( x_iX_1 \ldots X_k \).

(ii) Any \( \lambda \)-term with a type has a \( \beta \)-normal form and so long normal form (see Hindley [11] §2D5).

**Definition 2.9**: (Inhabitant, Long(\( \tau \)) )

An inhabitant of a type \( \tau \) is a closed \( \lambda \)-term \( M \) in \( \beta \)-normal form such that \( M : \tau \) is provable using rules \( \to_e \) and \( \to_i \) only. The set of inhabitants in long normal form of the given type \( \tau \) is called Long(\( \tau \)).

**Example 2.6**:

Let \( \tau \equiv (a \to b) \to a \to b \), \( M \equiv \lambda x . . x \) and \( N \equiv \lambda xy . xy \).

From figures 2.3 and 2.4 below, we have \( \vdash M : \tau \) and \( \vdash N : \tau \).

\[
\begin{array}{l}
(1) \\
\begin{array}{c}
\text{x : a \to b} \\
\to_e \\
\text{xy : b}
\end{array} \\
\begin{array}{c}
\text{y : a} \\
\to_i \\
\text{\( \lambda y . xy : a \to b \)}
\end{array}
\end{array}
\]

\[
\begin{array}{l}
(2) \\
\begin{array}{c}
\text{\( \lambda y . xy : (a \to b) \to a \to b \)} \\
\to_i \\
\text{\( \lambda xy . xy : (a \to b) \to a \to b \)}
\end{array}
\end{array}
\]

fig. 2.3 fig. 2.4
But $M \not\in \text{Long}(\tau)$ because in the deduction of $M : \tau$ the head-variable $x$ receives the type $a \to b$, thus making $x$ "hungry" for an argument with type $a$. On the other hand $N \in \text{Long}(\tau)$ because $x$ has an argument in $N$.

The set of long inhabitants of type $(\tau)$ can be reduced to the set of inhabitants in $\beta \eta$-normal form of the same type $(\tau)$ by $\eta$-reduction. The reduced inhabitants now represent normalized proofs.

§2.5 Lambda Reductions and Proof Reductions

We show how the $\lambda$-reduction rules, $(\beta)$ and $(\eta)$ when applied to typed terms, correspond to proof reductions or proof normalizations:

$$(\beta) \quad (\lambda x.M) N \Rightarrow [N/x] M$$

corresponds to the normalization

$$(I) \quad \begin{array}{l}
\text{x : } \alpha \\
D_1 \\
M : \beta \\
\rightarrow_i \quad \text{------------------------ (1) } \quad D_2 \\
\lambda x. M : \alpha \to \beta \\
\rightarrow_e \quad \text{-----------------------------} \\
(\lambda x. M) N : \beta
\end{array} \quad \text{to} \quad \begin{array}{l}
N : \alpha \\
D_1 \\
[N/x] M : \beta
\end{array}$$

This reduction procedure, when carried out at all possible places in a proof, normalizes the proof in the sense that no $\rightarrow_e$ rule has as its major premise a formula (type) formed by an $\rightarrow_i$ rule.
Note that in most cases this procedure gives a shorter proof, however if \( x : \alpha \) is used several times in \( D_1 \) and \( D_2 \) is long, the length of the proof may be increased by this reduction.

\[
(\eta) \quad \lambda x.Mx \to M, \ x \notin FV(M) \quad \text{corresponds to the normalization}
\]

\[
\begin{align*}
&D_1 \\
&\rightarrow_e \quad \text{------------------} \\
&M : \alpha \to \beta \quad x : \alpha \\
&\rightarrow_i \quad \text{-------------} (1) \quad \text{to} \\
&\lambda x.Mx : \alpha \to \beta
\end{align*}
\]

where \( x \) does not occur free in \( D_1 \) and hence does not occur free in \( M \).
CHAPTER 3

Translations

§3.1 Introduction

We know, from Chapter 2, that for each \( \lambda \)-term there is a corresponding combinatory term formed using the combinators \( K \) and \( S \). Also for each \( SK \)-term there is a \( \lambda \)-term. Each typed \( \lambda \)-term represents a natural deduction style proof, the corresponding combinator represents a Hilbert style proof of the same theorem.

In this chapter we will discuss the translations of \( Q \)-combinators, where \( Q \) is weaker than \( SK \), into \( \lambda \)-terms and the translation of certain \( \lambda \)-terms into \( Q \)-combinators. These \( \lambda \)-terms will represent proofs in natural deduction systems equivalent to the Hilbert-style \( Q \)-logics. This forms a basis for proof generating programs for these logics developed in Chapter 9.

Trigg et al [17], has algorithms for finding the terms \( \lambda x_1x_2...x_n.Y \) where \( Y \) is \( \lambda \)-free, which can be translated into combinatory terms using a particular base set \( Q \). The set of these \( Y \)'s is the set of terms for which there is a term \( M \), made up of elements of the basis \( Q \) and perhaps variables other than \( x_1,x_2,...,x_n \) such that

\[ Mx_1x_2...x_n >_Q Y \]

where \( >_Q \) is reducibility by the axioms for the members of \( Q \). For each base, such an \( M \) was obtained by a particular algorithm.
In Bunder [6], some of these algorithms were simplified, extended to some more bases and extended to translate arbitrary $\lambda$-terms, rather than just those of the form $\lambda x_1 \ldots x_n . Y$ with $Y$ $\lambda$-free, into $Q$-combinators.

First, we will list the relations between some bases.

§3.2 Relations between some bases of combinators

In Chapter 1, we introduced a $Q$-combinator as a term formed by elements of a set $Q$ by using the application operation. Now, we define a relation “strictly weaker than” between these $Q$'s as follows :-

Definition 3.1 (Strictly weaker “$\vartriangleleft$”)

We say that $Q'$ is strictly weaker than $Q$ or $Q' \vartriangleleft Q$ if:

(i) Every member $X$ of $Q'$ is definable in $Q$, in the sense that there is a $Q$-combinator $X^*$ satisfying the reduction axiom for $X$.

(ii) There is a member of $Q$ which is not definable in $Q'$.

From Trigg et al [17] we list the following relations between some bases and add two more.
§ 3.3 Translation algorithms

λ Q-terms are formed by application and λ - abstraction from variables, elements of Q and perhaps noncombinator constants.

A Q-term (or λ Q-term) Y can be translated into a λ-term $Y_\lambda$ by replacing the elements of Q in Y as follows:

- I by $I_\lambda = \lambda x_1.x_1$
- K by $K_\lambda = \lambda x_1x_2.x_1$
A mapping * from \( \lambda Q \) - terms to \( Q \) - terms will be defined by

(i) \( a^* = a \)

(ii) \((MN)^* = M^* N^* \)

(iii) \((\lambda x . M)^* = \lambda^* x . M^* \)

where * is usually a sequence of some of the following clauses :-

(i) \( \lambda^* x . x = I \)

(k) \( \lambda^* x . M = KM \) if \( x \not\in \text{FV}(M) \)

(\eta) \( \lambda^* x . Mx = M \) if \( x \not\in \text{FV}(M) \)

(s) \( \lambda^* x . MN = S( \lambda^* x . M)( \lambda^* x . N) \)

(b) \( \lambda^* x . MN = BM( \lambda^* x . N) \) if \( x \not\in \text{FV}(M) \)

(c) \( \lambda^* x . MN = C( \lambda^* x . M)N \) if \( x \not\in \text{FV}(N) \)

Note that an atom can be a variable or a constant that is not translated (such as \( f_e \) in Chapter 6).

Example 3.1:

If \( * = (ik\eta s) \) the abstraction algorithm used in §1.3, then

\[
\lambda^* x_1 x_2 x_3 . x_1(x_2x_3) = \lambda^* x_1 . (\lambda^* x_2 . (\lambda^* x_3 . x_1(x_2x_3)))
\]

\[
= \lambda^* x_1 . (\lambda^* x_2 . (S(\lambda^* x_3 . x_1)(\lambda^* x_3 . x_2x_3))))
\]
\[ \lambda^* x_1 . (\lambda^* x_2 . S(Kx_1) x_2) \]
\[ \equiv \lambda^* x_1 . S(Kx_1) \]
\[ \equiv S(\lambda^* x_1 . S)(\lambda^* x_1 . Kx_1) \]
\[ \equiv S(KS)K. \quad \text{(In SK- logic).} \]

But if \( * = (\eta \iota b) \) then
\[ \lambda^* x_1 x_2 x_3 . x_1(x_2 x_3) \equiv \lambda^* x_1 . (\lambda^* x_2 . (\lambda^* x_3 . x_1(x_2 x_3))) \]
\[ \equiv \lambda^* x_1 . (\lambda^* x_2 . (Bx_1(\lambda^* x_3 . x_2 x_3))) \]
\[ \equiv \lambda^* x_1 . (\lambda^* x_2 . Bx_1 x_2) \]
\[ \equiv \lambda^* x_1 . Bx_1 \]
\[ \equiv B \]

So by using different translation algorithms \( * \), which we define formally below, we can find Hilbert-style proofs for a given implicational formula in different logics.

We will be interested in a particular kind of mapping from \( \lambda \) - terms to \( Q \) - terms called a " \( Q \) - translation Algorithm " for \( Q = BCI, BCK, BB'I,...etc. \).

**Definition 3.2 :** (\( Q \) - translation algorithm)

A mapping \( * \) from \( \lambda \) - terms to \( Q \) - terms is said to be a \( Q \) - translation algorithm if:-

(A) For every \( Q \)-combinator \( X, X_{\lambda^*} \) is defined and
\[ X_{\lambda^*} \equiv X \]

(B) If for a \( \lambda \)-term \( Y, Y_\ast \) is defined and is a \( Q \)-term, then there is a \( \lambda \)-term \( Y_1 \) such that
\[ Y_{\lambda^*} \triangleright_Q Y_1 \triangleleft_\eta Y \]
where $\triangleright_Q$ means that only full or partial reductions involving the $\lambda$-versions of $Q$-combinators are used and $\triangleleft_\eta$ involves only $\eta$ and $\alpha$ reductions.

A $\lambda$-term $Y$ is $Q$-definable if there is a $Q$-translation algorithm $*$ for which $Y^*$ is a $Q$-term.

**Example 3.2**: If $Q = SK$

By Example 3.1

\[
(\lambda x_1 x_2 x_3 . x_1(x_2x_3))^*_{\lambda} = (S(KS)K)_{\lambda}
\]

\[
= S\lambda(K\lambda S\lambda)K\lambda
\]

\[
=(\lambda x_1 x_2 x_3 . x_1x_3(x_2x_3))(K\lambda S\lambda)(K\lambda)
\]

\[
\triangleright_{SK} \lambda x_3 . K\lambda S\lambda x_3(K\lambda x_3)
\]

\[
= \lambda x_3 . (\lambda x_1 x_2 . x_1)S\lambda x_3(K\lambda x_3)
\]

\[
\triangleright_{SK} \lambda x_3 . S\lambda (K\lambda x_3)
\]

\[
= \lambda x_3 . (\lambda x_4 x_5 x_6 . x_4x_6(x_5x_6))(K\lambda x_3)
\]

\[
\triangleright_{SK} \lambda x_3 . (\lambda x_5 x_6 . K\lambda x_3 x_6(x_5x_6))
\]

\[
= \lambda x_3 . (\lambda x_5 x_6 . (\lambda x y . x) x_3 x_6(x_5x_6))
\]

\[
\triangleright_{SK} \lambda x_3 . (\lambda x_5 x_6 . x_3(x_5x_6))
\]

\[
= \lambda x_3 x_5 x_6 . x_3(x_5x_6).
\]

**Properties of $Q$-translation algorithms.**

(For proofs see Bunder [6])

(1) If $*$ is a $Q$-translation algorithm, then for every $Q$-term $X$, $X^*_{\lambda}$ is defined and

\[
X^*_{\lambda} \equiv X
\]
(2) If $*$ is a Q-translation algorithm, then if $U$ and $V$ are $\lambda$-terms for which $U*$ and $V*$ are defined and are Q-terms, we have

$$U =_{\beta \eta} V \iff U* =_{\beta \eta} V*$$

where $=_{\beta \eta}$ is the equality which is the closure of $>_Q$ under $(\xi)$, $(\mu)$, $(v)$, $(\sigma)$, $(\tau)$ and $(\eta)$.

(3) If $*$ is a Q-translation algorithm and $Y*$ is defined and is a Q-term, then we have

$$(\lambda x . Y)*x =_{\beta \eta} Y*$$

(4) If $Q$ is any basis set for which $X$ and $Y$ are Q-terms, then

$X >_Q Y \Rightarrow X_\lambda >_Q Y_\lambda$

(5) If $Z$ is Q-definable, then there is a Q-term $X$ such that

$$Z =_{\beta \eta} X_\lambda$$

(6) If $X$ is a Q-term and there is a Q-translation algorithm, then $X_\lambda$ is Q-definable.

§3.4 The bases SK, BCK, BCI and BCIW

To find translation algorithms for these bases we need the following lemmas.

Lemma 3.1

If $U$ is a $\lambda$-term for which $U*$ is defined, then

(i) If $*$ is the (i $\eta$ ksa) algorithm, $(\lambda^*x.U*)x >_{KS} U_*$

(ii) If $*$ is the (i $\eta$ kbc) algorithm, $(\lambda^*x.U*)x >_{BCK} U_*$
(iii) If * is the (iηbc) algorithm, $(λ^*x.U_*)x >_{BCI} U_*$
(iv) If * is the (iηbcs) algorithm, $(λ^*x.U_*)x >_{BCIW} U_*$

Proof (ii) :

By induction on the length of $U_*$ (where $I ≡ CKK$).

If $U_* \equiv x$, then

$$(λ^*x.U_*)x \equiv (λ^*x.x)x ≡Ix>_{BCK}x≡U_*$$

If $U_* \equiv U_1x$ where $x ∉ FV(U_1)$, then

$$(λ^*x.U_*)x \equiv (λ^*x.U_1)x \equiv U_1x \equiv U_*$$

If $x ∉ FV(U_*)$, then

$$(λ^*x.U_*)x \equiv Ku_*x >_{BCK} U_*$$

If $U_* \equiv U_1U_2$ where $x ∉ FV(U_1)$ - FV(U_2) and $U_2 ≠ x$, then

$$(λ^*x.U_*)x \equiv BU_1(λ^*x.U_2)x >_{BCK} U_1((λ^*x.U_2)x) >_{BCK} U_1U_2 ≡ U_*$$

by the induction hypothesis.

If $U_* \equiv U_1U_2$ where $x ∉ FV(U_1)$ - FV(U_2) then

$$(λ^*x.U_*)x \equiv C(λ^*x.U_1)U_2x >_{BCK} (λ^*x.U_1)xU_2 >_{BCK} U_1U_2 ≡ U_*$$

by induction hypothesis.

Cases (i), (iii) and (iv) are similar.

Lemma 3.2

(i) (iηks) is an SK - translation algorithm.
(ii) (iηkbc) is a BCK - translation algorithm.
(iii) (iηbc) is a BCI - translation algorithm.
(iv) (iηbcs) is a BCIW - translation algorithm.

Proof (ii) :

For every BCK- combinator $X$, if $* = (iηkbc)$ then $X_{λ^*}$ is defined and $X_{λ^*} ≡ X$. So Definition 3.2(A) holds.
We prove (B) by induction on the length of the λ-term Y.

If Y is an atom, then \( Y_\lambda \equiv Y \)

If Y ≡ UV, then \( Y_\lambda \equiv U_\lambda \lambda V_\lambda \) and by the inductive hypothesis we have a \( U_1 \) and \( V_1 \) such that

\[
\lambda x. U_1 \vdash_{\text{BCK}} U_1V_1 \prec_\eta UV \equiv Y
\]

If Y ≡ λx.Xx where \( x \notin \text{FV}(X) \), then

\[
Y_\lambda \equiv (\lambda x. Xx)_\lambda \equiv X_\lambda
\]

By the induction hypothesis there is an \( X_1 \) such that

\[
Y_\lambda \equiv X_\lambda \vdash_{\text{BCK}} X_1 \prec_\eta X \prec_\eta Y
\]

If Y ≡ λx.UV where \( x \in \text{FV}(U) - \text{FV}(V) \), then

\[
Y_\lambda \equiv B \lambda \lambda (\lambda x. V)_\lambda \lambda \equiv (\lambda uvx.u(vx)) (U_\lambda) (\lambda x. V)_\lambda \lambda \vdash_{\text{BCK}} \lambda x. U_\lambda ((\lambda x. V)_\lambda) \equiv \lambda x. U_\lambda ((\lambda^* x. V^*)_x)_\lambda
\]

\[
\vdash_{\text{BCK}} \lambda x. U_\lambda V_\lambda \quad \text{by Lemma 3.1 and property (4).}
\]

On the other hand,

if Y ≡ λx.UV where \( x \in \text{FV}(U) - \text{FV}(V) \), then

\[
Y_\lambda \equiv C \lambda \lambda (\lambda x. U)_\lambda \lambda \lambda V_\lambda \lambda \equiv (\lambda uvx.uux) (\lambda x. U)_\lambda \lambda \lambda V_\lambda \lambda \vdash_{\text{BCK}} \lambda x. (\lambda x. U)_\lambda \lambda x V_\lambda \lambda \equiv \lambda x. ((\lambda^* x. U^*)_x)_\lambda V_\lambda \lambda \vdash_{\text{BCK}} \lambda x. U_\lambda \lambda \lambda V_\lambda \lambda \quad \text{also by Lemma 3.1 and property (4).}
\]

Now by the induction hypothesis there exist \( U_1 \) and \( V_1 \) such that

\[
\lambda x. U_1 \lambda V_1 \vdash_{\text{BCK}} \lambda x. U_1V_1 \prec_\eta \lambda x.UV \equiv Y
\]

Hence (iηkbc) is a BCK-translation algorithm.

Cases (i), (iii) and (iv) are similar.

In order to clarify the λ-terms definable using these bases Q, we need the following definition.
Definition 3.3:
If $x_1, x_2, \ldots$ is a given sequence of variables and \{ $x_i, \ldots, x_n$ \} is a subset of these variables, then
(i) A $\lambda$ -, $Q$ - or $\lambda Q$ - term is in $\text{Once}(i_1, \ldots, i_n)$ if each of $x_i, \ldots,$ $x_n$ appears free exactly once in the term and if in every subterm $\lambda x_{j_1} \ldots x_{j_k} \cdot Y$ of the term, $Y$ is in $\text{Once}(j_1, \ldots, j_k)$.
(We write $\text{Once}_n$ for $\text{Once}(1, 2, \ldots, n)$.)
(ii) A $\lambda$ -, $Q$ - or $\lambda Q$ - term is in $\text{Once}^-(i_1, \ldots, i_n)$ if each of $x_i, \ldots,$ $x_n$ appears free at most once in the term and if in every subterm $\lambda x_{j_1} \ldots x_{j_k} \cdot Y$ of the term, $Y$ is in $\text{Once}^-(j_1, \ldots, j_k)$.
(We write $\text{Once}^-_n$ for $\text{Once}^-(1, 2, \ldots, n)$.)
(iii) A $\lambda$ -, $Q$ - or $\lambda Q$ - term is in $\text{Once}^+(i_1, \ldots, i_n)$ if each of $x_i, \ldots,$ $x_n$ appears free at least once in the term and if in every subterm $\lambda x_{j_1} \ldots x_{j_k} \cdot Y$ of the term, $Y$ is in $\text{Once}^+(j_1, \ldots, j_k)$.
(We write $\text{Once}^+_n$ for $\text{Once}^+(1, 2, \ldots, n)$.)

Example 3.2:
\[
x_1 x_2 x_3 (\lambda x_4 x_5 \cdot x_4 x_6 x_5) x_7 x_7 \\
\in \text{Once}(1, 2, 3, 6) \cap \text{Once}^-(1, 2, 3, 4, 5, 6) \cap \text{Once}^+(1, 2, 3, 6, 7).
\]

Theorem 3.1
(i) The set of $\textbf{SK}$- definable terms is $\Lambda$. ($\Lambda$ denotes all $\lambda$- terms)
(ii) The set of $\textbf{BCI}$- definable terms is $\Lambda \cap \text{Once}(\ )$.
(iii) The set of $\textbf{BCK}$- definable terms is $\Lambda \cap \text{Once}^-\(\ )$.
(iv) The set of $\textbf{BCIW}$- definable terms is $\Lambda \cap \text{Once}^+(\ )$. 
Proof (iii):

If \( Y \in \Lambda \cap \text{Once}^-(i_1,i_2, \ldots, i_n) \) for some \( i_1,i_2, \ldots, i_n \) we show by induction on \( Y \) that \( Y \) is BCK-definable using the \( \text{(i} \eta \text{kbc)} \) algorithm.

**Case 1:** If \( Y \) is an atom, then \( Y_{\text{(i} \eta \text{kbc)}} = Y \)

**Case 2:** If \( Y \equiv UV, U \in \Lambda \cap \text{Once}^-(j_1,j_2, \ldots, j_r) \) and \( V \in \Lambda \cap \text{Once}^-(m_1,m_2, \ldots, m_s) \) where \( (j_1,j_2, \ldots, j_r) \) and \( (m_1,m_2, \ldots, m_s) \) are disjoint subsequences of \( (i_1,i_2, \ldots, i_n) \) and \( r+s \leq n \), then by the induction hypothesis \( U \) and \( V \) are BCK-definable using the \( \text{(i} \eta \text{kbc)} \) algorithm and

\[
Y_{\text{(i} \eta \text{kbc)}} = U_{\text{(i} \eta \text{kbc)}} V_{\text{(i} \eta \text{kbc)}}
\]

**Case 3:** If \( Y \equiv \lambda x_p. \, Zx_p \) where \( x_p \not\in \text{FV}(Z) \) then \( Zx_p \in \Lambda \cap \text{Once}^-(i_1,i_2, \ldots, i_r,p) \). Then by the induction hypothesis \( Zx_p \) is BCK-definable as \( Z_{\text{(i} \eta \text{kbc)}} x_p \) and

\[
Y_{\text{(i} \eta \text{kbc)}} = Z_{\text{(i} \eta \text{kbc)}}
\]

**Case 4:** If \( Y \equiv \lambda x_p. \, UV \) where \( UV \in \Lambda \cap \text{Once}^-(i_1,i_2, \ldots, i_r,p) \) and so \( x_p \in \text{FV}(V) - \text{FV}(U) \) or \( x_p \in \text{FV}(U) - \text{FV}(V) \)

Then, for similar disjoint sequences to the above we have

\[
U \in \Lambda \cap \text{Once}^-(j_1,j_2, \ldots, j_r) \quad \text{and} \quad V \in \Lambda \cap \text{Once}^-(m_1,m_2, \ldots, m_s,p)
\]

or

\[
U \in \Lambda \cap \text{Once}^-(j_1,j_2, \ldots, j_r,p) \quad \text{and} \quad V \in \Lambda \cap \text{Once}^-(m_1,m_2, \ldots, m_s)
\]

In the former case \( U,V \) and \( \lambda x_p. \, V \) are BCK-definable and

\[
Y_{\text{(i} \eta \text{kbc)}} = BU_{\text{(i} \eta \text{kbc)}} (\lambda_{\text{(i} \eta \text{kbc)}} x_p \cdot V_{\text{(i} \eta \text{kbc)}})
\]

In the latter case \( U,V \) and \( \lambda x_p. \, U \) are BCK-definable and

\[
Y_{\text{(i} \eta \text{kbc)}} = C (\lambda_{\text{(i} \eta \text{kbc)}} x_p \cdot U_{\text{(i} \eta \text{kbc)}}) V_{\text{(i} \eta \text{kbc)}}
\]

Thus the set of BCK-definable terms is \( \Lambda \cap \text{Once}^-(\_ \_ \_ \_ \_ \_ \_ \_ ) \).

(i), (ii) and (iv) are similar, except that in (iv) \( (m_1,m_2, \ldots, m_s) \) and \( (j_1,j_2, \ldots, j_r) \) need not to be disjoint and \( r+s \) need not be \( \leq n \).
Examples 3.3:

The following λ-terms are definable in the given logics:

(i) \( \lambda x_1 . x_1 , \lambda x_1 x_2 . x_2 x_1 \) and \( \lambda x_1 x_2 . x_2(\lambda x_3 . x_1 x_3) \)
are in SK, BCI, BCK and BCIW - logics.

(ii) \( \lambda x_1 x_2 . x_1 , \lambda x_1 x_2 x_3 . x_1 x_2 \) and \( \lambda x_1 x_2 x_3 . x_1(\lambda x_4 . x_3 x_4) \)
are in SK and BCK - logics.

(iii) \( \lambda x_1 x_2 . x_1 x_2 x_2 , \lambda x_1 x_2 . x_1 x_2(\lambda x_3 . x_2 x_3) \)
and \( \lambda x_1 x_2 x_3 . x_1 x_2(\lambda x_4 . x_4 x_2 x_3) \)
are in SK and BCIW - logics.

§3.5 Bases without C

For all \( x_i , x_j \) and \( x_p \), the abstraction algorithms (iηsk) and (isk) have \( \lambda^* x_i . \lambda^* x_j . X \) and \( \lambda^* x_p . \lambda^* x_j . X \) defined in the basis SK, but in bases such as BB'I and BB'IW there are algorithms which can define only one of them. For example, we can define \( \lambda^* x_3 . x_1 (x_2 x_3) \) as \( B x_1 x_2 \) and

\( \lambda^* x_2 . \lambda^* x_3 . x_1 (x_2 x_3) \) as \( B x_1 \) but it is not easy to define \( \lambda^* x_1 . \lambda^* x_3 . x_1 (x_2 x_3) \) unless we have the clause (c).

To cover this problem, Trigg et al defined simultaneous multiple abstractions (See also Bunder [6]).

We will say that a Q-term \( X \) is \( Q(i_1,i_2,...,i_n) \)-abstractable if there is a Q-term \( M \) such that \( \text{FV}(M) \cap \{ x_{i_1}, x_{i_2}, \ldots, x_{i_n} \} = \emptyset \) and

\( M x_{i_1} \ldots x_{i_n} \triangleright X \)

If each \( i_m = m \), then this is called \( Q_n \)-abstractability.

Note that the property \( \lambda^* x_i x_j . X \equiv \lambda^* x_i . \lambda^* x_j . X \) will not hold in general for every *.
We will use $\lambda_{x_{i_1}, \ldots, x_{i_n}}^x \cdot X$ to be an abstraction with respect to $x_{i_{n+1}}$, which is designed to facilitate abstraction with respect to $x_{i_n}$ later, $x_{i_{n-1}}$ after that, etc. This abstraction will be tied to some basis set $Q$ and certain algorithm clauses which we still denote by $\ast$.

For the $BBI$, $BBIK$ and $BBIW$ - translation algorithms, we will write $Y^\ast$ as $\left( ; Y \right)^\ast$. The definition below changes the $\lambda$'s in $Y$ to appropriate $\lambda_{x_{i_1}, \ldots, x_{i_n}}^x$'s and then performs abstraction (which, when $n=0$ we will also call $\lambda^\ast$) according to a list of clauses in $\ast$ such as i,k etc.

**Definition 3.4:**

$\left( x_{i_1}, \ldots, x_{i_n}; Y \right)^\ast$ and in particular $\left( ; Y \right)^\ast$ are given by:

\[
\begin{align*}
(x_{i_1}, \ldots, x_{i_n}; a)^\ast & \equiv a \text{ if } a \text{ is an atom} \\
(x_{i_1}, \ldots, x_{i_n}; MN)^\ast & \equiv (x_{i_1}, \ldots, x_{i_n}; M)^\ast (x_{i_1}, \ldots, x_{i_n}; N)^\ast \\
(x_{i_1}, \ldots, x_{i_n}; \lambda x_{i_{n+1}} \cdot P)^\ast & \equiv \lambda_{x_{i_1}, \ldots, x_{i_n}}^{x_{i_{n+1}}}. (x_{i_1}, \ldots, x_{i_n}, x_{i_{n+1}}; P)^\ast \\
(\lambda x_{i_1} \cdot P)^\ast & \equiv \lambda^\ast x_{i_1} \cdot (x_{i_1}; P)^\ast
\end{align*}
\]

**Example 3.4:**

\[
\begin{align*}
(\lambda x_{i_1} \cdot x_2 \cdot x_3(\lambda x_4 \cdot x_1(\lambda x_5 \cdot x_2x_6))(\lambda x_2x_3 \cdot x_3(\lambda x_6 \cdot x_3)))^\ast \\
& \equiv \lambda^\ast x_{i_1} \cdot (x_1; \lambda x_2 \cdot x_3(\lambda x_4 \cdot x_1(\lambda x_5 \cdot x_2x_6))(\lambda x_2x_3 \cdot x_3(\lambda x_6 \cdot x_3)))^\ast \\
& \equiv \lambda^\ast x_{i_1} \cdot \lambda x_1 x_2 \cdot (x_1, x_2; x_3(\lambda x_4 \cdot x_1(\lambda x_5 \cdot x_2x_6))(\lambda x_2x_3. x_3(\lambda x_6 \cdot x_3)))^\ast \\
& \equiv \lambda^\ast x_{i_1} \cdot \lambda x_1 x_2 \cdot (x_1, x_2; x_3)^\ast (x_1, x_2; \lambda x_4 \cdot x_1(\lambda x_5 \cdot x_2x_6))^\ast (x_1, x_2; \lambda x_2x_3 \cdot x_3(\lambda x_6 \cdot x_3))^\ast
\end{align*}
\]
\[ \lambda^* x_1 . \lambda x_1 x_2 . x_3 (\lambda x_1 x_2 . x_1 (\lambda x_1 x_2 x_4 . x_2 x_6)) (\lambda x_1 x_2 . (\lambda x_1 x_2 x_2 . x_2 . x_3 (\lambda x_1 x_2 x_2 . x_3))]. \]

The situation in Example 3.4 where a variable is repeated in the superscript of a \( \lambda \) causes problems which are illustrated by Example 3.6 and Theorem 3.2 below.

We will therefore assume that for all the algorithms without \( C \) the \( \lambda Q \) term to be translated has been transformed by (\( \alpha \)) - conversion so that no \( \lambda x_i \) occurs twice in \( X \).

Now, we will discuss a translation algorithm for the bases \( BB'I \) and \( BB'TW \).

§ 3.6 The bases \( BB'I \) and \( BB'TW \)

Without the combinator \( C \), we can translate into at most a subset of \( \Lambda \). To describe such subsets we define, for each basis \( Q \), a set \( HRMQ(i_1, \ldots, i_n) \) of Hereditary Right Maximal terms with respect to \( x_{i_1} \ldots x_{i_n} \) as follows:

1. Every variable and every basis combinator is in \( HRMQ(i_1, \ldots, i_n) \).
2. If \( M, N \in HRMQ(i_1, \ldots, i_n) \) and \( \text{idx}(M, i_1, \ldots, i_n) \leq \text{idx}(N, i_1, \ldots, i_n) \) then \( MN \in HRMQ(i_1, \ldots, i_n) \), where
   \[ \text{idx}(M, i_1, \ldots, i_n) = \max\{p : 1 \leq p \leq n \text{ and } x_{i_p} \in \text{FV}(M)\} \]
   and \( \text{idx}(M, ) = 0 \).
3. If \( M \in HRMQ(i_1, \ldots, i_n, i_{n+1}) \) then \( \lambda x_{i_{n+1}} . M \in HRMQ(i_1, \ldots, i_n) \).

Usually the \( Q \) in \( HRMQ(...) \) will be omitted as it will be clear from the context.
Example 3.5:

HRM_{BB'I}(1, 2, 3, 4) contains
\(x_4, B, x_1x_2x_3, BIx_1x_2, x_3Bx_1x_2, x_2x_3(x_5x_3)\) and \(x_2x_3(x_1x_4)\)
but not \(x_3x_2, x_1x_3(x_1x_2)\) or \(x_2B\).

Example 3.6:

\(\lambda x_1x_2 . (\lambda x_2x_3x_4 . x_1x_2x_3(x_2x_4)) \notin HRM_{BB'I}(\quad)\) while
\(\lambda x_1x_5 . (\lambda x_2x_3x_4 . x_1x_5x_3(x_2x_4)) \in HRM_{BB'I}(\quad).\)

The algorithm that we use to evaluate \(\lambda^{x_{i_1,\ldots,x_{i_n}}}_{x_{i_1,\ldots,x_{i_n}}}.P\) for the basis BB'I is \((i \eta bb'\quad)\) where \((i\quad)\) and \((\eta)\) are as in the previous sections and 
\((b\quad)\) and \((b')\) are given by:

\[
(b) \quad \lambda^{x_{i_1,\ldots,x_{i_n}}}_{x_{i_1,\ldots,x_{i_n}}}.PQ \equiv BP(\lambda^{x_{i_1,\ldots,x_{i_n}}}_{x_{i_1,\ldots,x_{i_n}}}.Q)
\]
if \(x_{i_{n+1}} \notin FV(P)\) and either \(idx(P, i_1, \ldots, i_n) \leq idx(Q, i_1, \ldots, i_n)\)
or \(x_{i_1,\ldots,x_{i_n}}\) is replaced by \(*\).

\[
(b') \quad \lambda^{x_{i_1,\ldots,x_{i_n}}}_{x_{i_1,\ldots,x_{i_n}}}.PQ \equiv B'(\lambda^{x_{i_1,\ldots,x_{i_n}}}_{x_{i_1,\ldots,x_{i_n}}}.Q)P
\]
if \(idx(P, i_1, \ldots, i_n) > idx(Q, i_1, \ldots, i_n)\) and \(x_{i_{n+1}} \notin FV(P)\).

The algorithm for the basis BB'IW will be \((i \eta bb'ss')\) where \((s\quad)\) and
\((s')\) are given by:

\[
(s) \quad \lambda^{x_{i_1,\ldots,x_{i_n}}}_{x_{i_1,\ldots,x_{i_n}}}.PQ \equiv S(\lambda^{x_{i_1,\ldots,x_{i_n}}}_{x_{i_1,\ldots,x_{i_n}}}.P)(\lambda^{x_{i_1,\ldots,x_{i_n}}}_{x_{i_1,\ldots,x_{i_n}}}.Q)
\]
if \(idx(P, i_1, \ldots, i_n) \leq idx(Q, i_1, \ldots, i_n)\) or \(x_{i_1,\ldots,x_{i_n}}\)
is replaced by \(*\).
(s') \( \lambda_{x_{i_1}, \ldots, x_{i_n}}^x \cdot P Q \equiv S^\prime (\lambda_{x_{i_1}, \ldots, x_{i_n}}^x \cdot Q) (\lambda_{x_{i_1}, \ldots, x_{i_n}}^x \cdot P) \)
if \( \text{idx}(P, i_1, \ldots, i_n) \) > \( \text{idx}(Q, i_1, \ldots, i_n) \).

Note that \( S \) and \( S' \) can be defined as \( B(BW)(B(B'B')B') \) and \( B(BW)(BB'B') \) respectively.

Example 3.7:

If * is \((i\eta\eta \eta \eta b'b')\) then
\[
(\lambda x_1 x_2 x_3 . x_2 (\lambda x_4 . x_1 (\lambda x_5 . x_3 x_4 x_5)))^*
\]
\[
\equiv \lambda^* x_1 \cdot (x_1 ; \lambda x_2 x_3 . x_2 (\lambda x_4 . x_1 (\lambda x_5 . x_3 x_4 x_5)))^*
\]
\[
\equiv \lambda^* x_1 \cdot \lambda_{x_2}^x \cdot (x_1, x_2 ; \lambda x_3 . x_2 (\lambda x_4 . x_1 (\lambda x_5 . x_3 x_4 x_5)))^*
\]
\[
\equiv \lambda^* x_1 \cdot \lambda_{x_2}^x \cdot \lambda_{x_3}^x \cdot \lambda_{x_4}^x \cdot \lambda_{x_5}^x \cdot (x_1, x_2, x_3, x_4 ; \lambda x_5 . x_3 x_4 x_5))^*)
\]
\[
\equiv \lambda^* x_1 \cdot \lambda_{x_2}^x \cdot \lambda_{x_3}^x \cdot \lambda_{x_4}^x \cdot \lambda_{x_5}^x \cdot (x_1, x_2, x_3, x_4 ; \lambda x_5 . x_3 x_4 x_5))^*)
\]
\[
\equiv \lambda^* x_1 \cdot \lambda_{x_2}^x \cdot \lambda_{x_3}^x \cdot \lambda_{x_4}^x \cdot \lambda_{x_5}^x \cdot (x_1, x_2, x_3, x_4 ; \lambda x_5 . x_3 x_4 x_5))^*)
\]
\[
\equiv \lambda^* x_1 \cdot \lambda_{x_2}^x \cdot \lambda_{x_3}^x \cdot \lambda_{x_4}^x \cdot \lambda_{x_5}^x \cdot (x_1, x_2, x_3, x_4 ; \lambda x_5 . x_3 x_4 x_5))^*)
\]
\[
\equiv \lambda^* x_1 \cdot \lambda_{x_2}^x \cdot \lambda_{x_3}^x \cdot \lambda_{x_4}^x \cdot \lambda_{x_5}^x \cdot (x_1, x_2, x_3, x_4 ; \lambda x_5 . x_3 x_4 x_5))^*)
\]
\[
\equiv \lambda^* x_1 \cdot \lambda_{x_2}^x \cdot \lambda_{x_3}^x \cdot \lambda_{x_4}^x \cdot \lambda_{x_5}^x \cdot (x_1, x_2, x_3, x_4 ; \lambda x_5 . x_3 x_4 x_5))^*)
\]
\[
\equiv \lambda^* x_1 \cdot \lambda_{x_2}^x \cdot \lambda_{x_3}^x \cdot \lambda_{x_4}^x \cdot \lambda_{x_5}^x \cdot (x_1, x_2, x_3, x_4 ; \lambda x_5 . x_3 x_4 x_5))^*)
\]
\[
\equiv \lambda^* x_1 \cdot \lambda_{x_2}^x \cdot \lambda_{x_3}^x \cdot \lambda_{x_4}^x \cdot \lambda_{x_5}^x \cdot (x_1, x_2, x_3, x_4 ; \lambda x_5 . x_3 x_4 x_5))^*)
\]
\[
\equiv \lambda^* x_1 \cdot \lambda_{x_2}^x \cdot \lambda_{x_3}^x \cdot \lambda_{x_4}^x \cdot \lambda_{x_5}^x \cdot (x_1, x_2, x_3, x_4 ; \lambda x_5 . x_3 x_4 x_5))^*)
\]
Now, we list a lemma and the main theorems for BB'I and BB'IW, proofs are in Bunder [6].

**Lemma 3.3**

The following are translation algorithms:

(i) \((;)(^{inbb'})\), for BB'I .

(ii) \((;)(^{inbb'ss'})\), for BB'IW .

**Theorem 3.2**

(i) The set of BB'I- definable terms is \(HRM() \cap \Lambda \cap \text{Once}()\).

(ii) The set of BB'IW- definable terms is \(HRM() \cap \Lambda \cap \text{Once}^+(())\).

We define an \(\eta\)-normal form to be a term without \(\eta\)-redexes.

Also by considering the above bases without I and letting \(\Lambda^-\) be \(\Lambda\) without terms with \(\eta\)-normal forms of the form \(\lambda x_i ... x_i x_i ... x_i\) or \(\lambda x_i ... x_i x_i x_i x_{i+1} x_{i+2} ... x_i\), we have:

**Theorem 3.3**

(i) The set of BB'- definable terms is \(HRM() \cap \Lambda^- \cap \text{Once}()\).

(ii) The set of BB'W- definable terms is \(HRM() \cap \Lambda^- \cap \text{Once}^+(())\).

Below, we will discuss the translation algorithm for the base BB'IK.
§ 3.7 The base BB'IK

The BB'IK(i_1, ..., i_n) - translation algorithm will have as part of it a special ordering algorithm as follows:

**The Full Ordering Algorithm**

**Aim**: To extend a $\lambda$-BB'IK term $Y \in \text{Once}^{-}(i_1, ..., i_n)$ to a $\lambda$-BB'IK term $Y^o \in \text{HRM}(i_1, ..., i_n) \cap \text{Once}(i_1, ..., i_n)$ so that $Y^o \triangleright_{\text{KI}} Y$.

**Method**: First step:
1. If $Y \equiv a$, an atom not in $\{x_{i_1}, x_{i_2}, ..., x_{i_n}\}$ then
   $$Y^o \equiv K_{\lambda}(a).$$

2. If $Y \equiv x_{i_m}$, and $1 \leq m < n$ then
   $$Y^o \equiv K_{\lambda} x_{i_m}(x_{i_1} ... x_{i_{m-1}} x_{i_{m+1}} ... x_{i_n}).$$

3. If $Y \equiv x_{i_n}$ then
   $$Y^o \equiv K_{\lambda} I_{\lambda}(x_{i_1} ... x_{i_{n-1}})x_{i_n}.$$

4. If $Y \equiv \lambda x_{i_{n+1}.Z}$, find if possible, $Z^o$ such that
   $$Z^o \in \text{HRM}(i_1, ..., i_{n+1}) \cap \text{Once}(i_1, ..., i_{n+1})$$
   and $Z^o \triangleright_{\text{KI}} Y$.

5. If $Y \equiv UV$, find if possible, a term $U^o$ and a subsequence $(x_{j_1} ... x_{j_r})$ of $(x_{i_1} ... x_{i_n})$ such that:
   (i) $\text{FV}(U) \cap \{x_{i_1}, ..., x_{i_n}\} \subseteq \{x_{j_1}, ..., x_{j_r}\}$ and $\text{FV}(V) \cap \{x_{j_1}, ..., x_{j_r}\} = \emptyset$.
   (ii) $U^o \in \text{HRM}(j_1, ..., j_r) \cap \text{Once}(j_1, ..., j_r)$
   (iii) $U^o \triangleright_{\text{KI}} U$. 
(iv) \( j_r \neq i_n \)
(v) \( \max\{p \mid x_j \in \{x_{j_1}, \ldots, x_{j_r}\} - \text{FV}(U)\} \) is minimal.
(vi) Given (v), the number of variables in \( \{x_{j_1}, \ldots, x_{j_r}\} - \text{FV}(U) \) is minimal. Now let \((x_{s_1}, \ldots, x_{s_i})\) be the sequence obtained from \((x_{i_1}, \ldots, x_{i_n})\) by removing \((x_{j_1}, \ldots, x_{j_r})\).

Now, if possible, find \( V^0 \) such that
(vii) \( V^0 \in \text{HRM}(s_1, \ldots, s_i) \cap \text{Once}(s_1, \ldots, s_i) \) and
(viii) \( V^0 \gg_{KI} V. \)
then \( Y^0 \equiv U^0 V^0. \)

The BB'TK - translation algorithm

**Aim**: To translate a \( \lambda \)-BB'TK - term \( Y \) into a BB'TK - term.

**First step**: apply the full ordering algorithm to \( Y \) to give
\[ Y^0 \in \text{HRM}(\ ) \cap \text{Once}(\ ) \]

**Second step**: apply \(( ; )^{(\eta \text{bb}')}\) to \( Y^0. \)

**Example 3.8**:

To order \( Y = x_3 x_2 x_1(\lambda x_4 . x_5 x_4) \) relative to \((1, 2, 3, 5, \ldots, 8)\), we have
\[ Y^0 = x_3(K\lambda x_2 x_6)(K\lambda x_1 x_7)(\lambda x_4 . (K\lambda x_5 x_8) x_4) \]
\[ \in \text{HRM}(1, 2, 3, 5, \ldots, 8) \cap \text{Once}(1, 2, 3, 5, \ldots, 8) \]

Hence \(( ; Y^0)^{(\eta \text{bb}')}\) \(\equiv x_3(Kx_2 x_6)(Kx_1 x_7)(\lambda x_4^{(\eta \text{bb}')} . (K\lambda x_5 x_8) x_4) \)
\[ \equiv x_3(Kx_2 x_6)(Kx_1 x_7)(Kx_5 x_8) \]

The BB'TK - translatable \( \lambda \)-terms will be represented in terms of a generalisation of the class \( \text{HRM}(i_1, \ldots, i_n). \)
Definition 3.5: (Potentially Right Maximal \((i_1,\ldots,i_n) - \lambda\)-terms) 
(or \(\text{PRM}(i_1,\ldots,i_n) - \lambda\)-terms)

1. If \(a\) is an atom, then \(a \in \text{PRM}(\lambda)\).
2. \(x_e \in \text{PRM}(e)\).
3. If \(X \in \text{PRM}(i_1,\ldots,i_{n-1})\) and \(x_{i_n} \notin \text{FV}(X)\) then \(X \in \text{PRM}(i_1,\ldots,i_{n-1},i_n)\).
4. If \(X \in \text{PRM}(i_1,\ldots,i_{n+1})\) then \(\lambda x_{i_{n+1}} . X \in \text{PRM}(i_1,\ldots,i_n)\).
5. If \(X \in \text{PRM}(j_1,\ldots,j_p)\) and \(Y \in \text{PRM}(r_1,\ldots,r_q)\)
   where either, \(p = q = n = 0\), or \(r_q = i_n\), \(\{j_1,\ldots,j_p\} \cap \{r_1,\ldots,r_q\} = \emptyset\), \(\text{FV}(X) \cap \{x_{r_1},\ldots,x_{r_q}\} = \emptyset\), \(\text{FV}(Y) \cap \{x_{j_1},\ldots,x_{j_p}\} = \emptyset\) and \((i_1,\ldots,i_n)\)
   consists of the elements of \((j_1,\ldots,j_p)\) and \((r_1,\ldots,r_q)\) with the orders preserved,
   then \(XY \in \text{PRM}(i_1,\ldots,i_n)\).

Example 3.9:

\[x_1 \in \text{PRM}(1)\] then \(x_1 \in \text{PRM}(1,3)\) and \(x_2 \in \text{PRM}(2)\)
therefore \(x_2x_1 \in \text{PRM}(1,2,3)\)

The proofs of the following lemmas and theorems are in Bunder [6].

Lemma 3.4

If \(Y\) when ordered relative to \((i_1,\ldots,i_n)\) by the full ordering algorithm, becomes \(Y^0\) then \(Y^0 \not\geq_{KI} Y\)
where each single \(K\) - reduction eliminates one or more of \(x_{i_1},\ldots,x_{i_n}\).
Lemma 3.5
\[ Y \in PRM(i_1,\ldots,i_n) \cap \text{Once}^{-}(i_1,\ldots,i_n) \quad \Leftrightarrow \]
there is a \( Y^o \in HRM(i_1,\ldots,i_n) \cap \text{Once}(i_1,\ldots,i_n) \)
defined by the full ordering algorithm.

Lemma 3.6
\( (;\eta^{bb'}) \) is a \( BB'IK \) - translation algorithm.

Theorem 3.4
(i) The set of \( BB'IK \)- definable terms is \( PRM( ) \cap \Lambda \cap \text{Once}^{-}( ) \).
(ii) The set of \( BB'K \)- definable terms is \( PRM( ) \cap \Lambda^{-} \cap \text{Once}^{-}( ) \).

Now, we have a translation algorithm for the bases \( SK, BCK, BCI, BCIW, BB'I, BB'IW, BB', BB'W, BB'IK \) and \( BB'K \). By using a technique such as the Ben - Yelles algorithm, given in Chapter 4, we can find \( \lambda \) - terms representing the proofs of a given implicational formula and hence, we can see whether any of these \( \lambda \) - terms are translatable into a combinator that corresponds to a specific logic.

There are many efficient theorem proving algorithms such as that of Slaney and Scott [15] and many matrix based programs that can disprove potential theorems, such as Slaney and Meglicki is MaGIG [14]. These however do not in general guarantee a proof or disproof of a given formula. The Ben - Yelles algorithm of Chapter 4 do this for many logics.

Many of the logics we consider are well known substructural logics studied for example in Dosen [9] §1.
CHAPTER 4

Theorem Proving Algorithms for Implicational Logics

§ 4.1 Introduction

The classifications of the various classes of Q-definable λ-terms in Chapter 3 give in each case, a complete characterization of the class of normalized proofs of theorems in the corresponding Q-logic.

In this chapter, we introduce the tree form of the Ben-Yelles algorithm as in Bunder [4]. Given a formula of implicational logic, the algorithm determines the set (possibly empty) of λ-terms in long β-normal form which have a given formula as a type. Hence all the normalized natural deduction style proofs of the formula in intuitionistic logic can be obtained.

The search for long β-normal form λ-terms is inherently parallel and particularly efficient. For intuitionistic logic, the algorithm has a bound dependent on the length of the given formula.

Any one of these λ-terms will represent a proof of an implicational formula in intuitionistic implicational logic and it will be possible to determine whether the λ-term also represents a proof of the formula in a weaker logics such as BCK, BCI, ... etc. This uses the translations of Chapter 3.

We restrict ourselves here to the systems SK, BCK, BCI, BCIW, BB'IW, BB'IK which are well known substructural implicational logics.
The same logics without T, and those without C but with I can be treated in the same way, as classes of λ-terms translated into these sets of combinators have been developed in Bunder [6]. Others corresponding to less interesting logics can no doubt be developed.

Some important notation follows.

\textbf{Notation :}

If \( \tau = \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow a \) where \( a \) is an atomic type then \( \text{tail} (\tau) = a \).

\( |\tau| \) = the number of atom occurrences in \( \tau \).

\( ||\tau|| \) = the number of distinct atom occurrences in \( \tau \).

It is clear that \( ||\tau|| \leq |\tau| \), For example;

If \( \tau = (a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c \) then \( |\tau| = 7 \) and \( ||\tau|| = 3 \).

If \( \Gamma = \{x_1 : \tau_1 , \ldots , x_n : \tau_n\} \) then we use

\[ S(\Gamma) = \{ \tau_1 , \ldots , \tau_n \} \]

\[ \text{FV}(\Gamma) = \{x_1 , \ldots , x_n \} \]

\[ |\Gamma| = \sum_{i=1}^{n} |\tau_i| \]

\[ ||\Gamma|| = ||\tau_1 \rightarrow \ldots \rightarrow \tau_n|| \]

\textbf{Definition 4.1} (Depth of a type \( \tau \))

The depth of a type \( \tau \), \textbf{Depth} (\( \tau \)) is defined recursively, thus:

(i) \( \text{Depth} (a) = 1 \)

(ii) \( \text{Depth} (\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow a) \)

\[ = 1 + \text{Maximum} \{\text{Depth} (\tau_1) , \ldots , \text{Depth} (\tau_n)\} \].
In $\tau = \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow a$, $\tau$ itself is the positive component of depth 0 of $\tau$.

A positive component of $\tau_i$ of depth $n$ is a negative component of $\tau$ of depth $n+1$ and a negative component of $\tau_i$ of depth $n$ is a positive component of $\tau$ of depth $n+1$.

A particular subtype of $\tau$ may appear as a negative or positive component and at various depths.

Now, we will discuss the algorithm for finding proofs in $\lambda$-calculus.

§4.2 The Long ($\tau$) Generating Algorithm

Aim : Given $m > 1$ and $\tau_1, \ldots, \tau_{m-1}$ and $\tau$, to find all long normal forms $X$ such that

$$FV(X) \subseteq \{x_1, \ldots, x_{m-1}\}$$

and for $m \geq 1$

$$x_1 : \tau_1, \ldots, x_{m-1} : \tau_{m-1} \vdash X : \tau$$

- (A)

(At this stage $X$ in (A) is an unknown $\lambda$-term.)

Method : 

Let $\tau = \tau_m \rightarrow \ldots \rightarrow \tau_n \rightarrow a$ where $a$ is an atom and $n \geq m -1$.

Then $X$ must be of the form

$$X \equiv \lambda x_m \ldots x_n . X'$$

where for $m \leq i \leq n$, $x_i : \tau_i$. Thus (A) is solved if we can solve:

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash X' : a$$

- (B)

where $X'$ is a shorter unknown $\lambda$-term than $X$. 
Let $\tau_{i_1}, \ldots, \tau_{i_p}$ be the $\tau_i$'s ($1 \leq i \leq n$) that have $\text{tail}(\tau_i) = a$ and let for $1 \leq k \leq p$

$$\tau_{i_k} = \tau_{k,1} \rightarrow \ldots \rightarrow \tau_{k,r_k} \rightarrow a.$$ 

Then $X'$ must have the form, for at least one value of $k$

$$X' = x_{i_k} X_{k,1} \ldots X_{k,r_k}.$$ 

and so a solution for any $k$ ($1 \leq k \leq p$) of

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash x_{i_k} X_{k,1} \ldots X_{k,r_k} : a$$  \hspace{1cm} - (C)

for $X_{k,1}, \ldots, X_{k,r_k}$ provides a solution to (B).

Thus there is a solution to (B) and so to (A) if for any $k$, ($1 \leq k \leq p$), we have for all $s$ such that $1 \leq s \leq r_k$ solutions to

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash X_{k,s} : \tau_{k,s}.\hspace{1cm} - (D)$$

and (D) is solved as (A) by the same steps as above.

The algorithm produces a type - inhabitant (or TI) tree as in the following figure where we use the abbreviation

$$\Gamma_j = \{x_1 : \tau_1, \ldots, x_j : \tau_j\}.$$ 

\[ \begin{array}{c}
\Gamma_n \vdash X_{1,1} : \tau_{1,1} \ldots \Gamma_n \vdash X_{1,r_1} : \tau_{1,r_1} \ldots \Gamma_n \vdash X_{p,1} : \tau_{p,1} \ldots \Gamma_n \vdash X_{p,r_p} : \tau_{p,r_p} \hspace{1cm} (A) \\
\Gamma_n \vdash x_{i_1} X_{1,1} \ldots X_{1,r_1} : a \ldots \Gamma_n \vdash x_{i_p} X_{p,1} \ldots X_{p,r_p} : a \hspace{1cm} (C) \\
\Gamma_n \vdash X' : a \hspace{1cm} (B) \\
\Gamma_{m-1} \vdash X : \tau \hspace{1cm} (A) \\
\end{array} \] 

fig. 4.1
Figure 4.1 has (A) at the root which is called an (A) node. From (A) a branch goes up to a (B) node and from there \( p \) branches go up to the \( p \) nodes of type (C). From all of these there are branches to nodes (D), which will be also nodes of type (A). Above these there may be further nodes of type (B) etc.

An **ABC-subbranch** of a TI-tree is a branch of the tree consisting of any (A) node, the (B) node immediately above it and one of the (C) node directly above it.

A **minimally complete** (or MC-) **subtree** of a TI tree is one with the same root as that of the TI tree, which has only one branch going up from any (B) node.

An ABC subbranch is said to **terminate** if the (C) node is of the form \( \Gamma_n \vdash x_j : a \). This then provides a solution \( X' = x_j \) to the (B) node below it and a solution \( X \) in the (A) node below that.

A sub-tree starting at an (A) node is said to **terminate** if that node is part of an ABC subbranch which either terminates or which has all the sub-trees from the (A) nodes above the (C) node terminating. In this case, also, the (A) node has a solution for \( X \). Any TI-tree that has a terminating subtree will have an MC subtree that terminates (a TMC subtree) and may have several infinite branches.

**Note that** a TMC-subtree with all variable terms replaced by their solutions, becomes a "type assignment deduction" or \( TA_\lambda - \) deduction as
in Hindley and Seldin [10]. The TI - tree can be seen as the expanded union of all TA^λ deductions for a given type. The expansion involves separating (B) and (C) nodes to allow for alternative solutions.

If in a branch at a (B) node, as in figure 4.1, there is no τ_i with tail a, then there is no solution X' to (B) and so no solution X for the (A) node below it and so no solution to the (C) node (if any) below that. In this case the branch dies and we prune it above the next (B) node below this (C).

If a whole tree dies and is pruned, then there is no solution for the X in the (A) node at the root.

A term Y is said to be a subargument of

\[ X = \lambda x_1 \ldots x_n \cdot x_iX_1 \ldots X_k \]

if Y is X_j for 1 ≤ j ≤ k or Y is a subargument of such an X_j.

When we compare an (A) node Γ_k \vdash Y : μ with a lower (A) node Γ_s \vdash Z : ζ, in the same branch of a TI- tree, we can see that μ must be a positive proper subtype of ζ or a negative subtype of one of the types in Γ_s. Also Γ_s ⊆ Γ_k and any solution found through this branch for Z will have Y as a subargument of Z.

We will call a branch of a TI - tree which has nodes Γ \vdash Y : b and Γ^+ \vdash Z : b, a limited type repetition (or LTR) branch. If also S(Γ) = S(Γ^+) it will be called a type repetition (or TR-) branch.
Now, we will find the number of (B) nodes in the Ben-Yelles algorithm which we have to consider to give a decision procedure for the intuitionistic implicational (SK-) logic and also to give a proof, if any, of a formula in that logic.

§4.3 Finding Proofs and Deciding Infiniteness in Intuitionistic Implicational Logic

Theorem 4.1

(i) $\vdash X: \tau$ has a solution for $X$ iff the TI-tree with $\vdash X: \tau$ at the root has a TMC subtree with no TR-branches.

(ii) If $\vdash X: \tau$ has a solution for $X$ in $\text{SK}$, then there is a TMC subtree of TI-tree in which each branch has at most

$$|\tau| - 1 \parallel \tau\parallel (B) \text{ nodes.}$$

Proof:

(i) We have: $\vdash X: \tau$ has a solution for $X$ iff the TI-tree has a TMC-subtree. If we consider the TMC-subtree of minimal size, then this TMC-subtree will have a TR-branch iff the $\text{TA}_\lambda$ deduction obtained from this has a TR-branch; the $\text{TA}_\lambda$ deduction will also be of minimal size.

Assume the $\text{TA}_\lambda$ deduction has a TR-branch with node $\Gamma^+ \vdash Z: b$ above a node $\Gamma \vdash Y: b$. We then replace every $x_i$ in $Z$ such that $(x_i : \tau_i) \in \Gamma^+ - \Gamma$ by an $x_j$ such that $(x_j : \tau_i) \in \Gamma$. (This can be done
because \( S(\Gamma^+) = S(\Gamma^-) \). If this produces \( Z^+ \), then this is a solution for \( Y \) obtained by a smaller subtree. But this is impossible because we assumed that TMC - subtree has a minimal size and so the \( \text{TA}_\lambda \) deduction and the subtree can have no \( \text{TR} \ - \) branches.

(ii) An upper bound on the number of \((B)\) nodes in any branch of this subtree is given by the number of atomic types that can appear to the right of the \( \text{I} \)- times the maximal length of the nondecreasing sets \( S(\Gamma) \) for \( \Gamma \) on the left of the \( \text{I} \)-. A branch that is any longer must have type repetition. Thus this upper bound is \( (|\tau| - 1) \| \tau \| (B) \) nodes.

**Corollary:**

If in the TI - tree with \( \text{I} \)- \( X : \tau \) as root, there is no solution for \( X \) in any subtree without type repetition, then there is no solution at all. The maximal number of \((B)\) nodes in any branch that needs to be checked is

\[ (|\tau| - 1) \| \tau \| \]

Below are some examples for which proofs are found using the Long \((\tau)\) Generating Algorithm.

**Example 4.1**

To find Long \(((a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c)\).
We explain the steps in fig 4.2 from bottom to top as follows:

First(A) : Assume that this formula is a type of a \( \lambda \)-term in long normal form \( X \).

First(B) : For \( X \) to be in long normal form, it must be of the form \( \lambda x_1x_2x_3. X' \) where \( x_1 : a \rightarrow b \rightarrow c \), \( x_2 : b \), \( x_3 : a \) and \( X' : c \).

We abbreviate \( x_1 : a \rightarrow b \rightarrow c \), \( x_2 : b \), \( x_3 : a \) to \( \tau_1 \).

First(C) : Check each \( x_i : \tau_i \)

(i) If \( \tau_i \) is equal to \( c \) (the tail of the formula) then \( x_i = X' \) is a possible solution for \( X' : c \). (This fails in this case.)

(ii) If the tail of \( \tau_i \) is \( c \) then we write \( x_iX_1X_2...X_n : c \) where \( X_1,X_2,...,X_n \) are inhabitants of the long negative subtypes of \( \tau_i \).

Here, we have \( x_1 : a \rightarrow b \rightarrow c \) and so \( x_1X_1X_2 : c \) where \( X_1 : a \) and \( X_2 : b \).
(iii) Otherwise, this branch dies (not in this case).

Top (A) and (B) : We compare the type of $X_1$ and $X_2$ with the types in $\Gamma_3$. As $x_3 : a$, $x_2 : b \in \Gamma_3$, we can define $X_1 = x_3$ and $X_2 = x_2$.

From fig. 4.2, we can see that the only proof is

$$\lambda x_1 x_2 x_3 . x_1 x_3 x_2$$

i.e. $\text{Long} \left( (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c \right) = \{ \lambda x_1 x_2 x_3 . x_1 x_3 x_2 \}$.

**Example 4.2**

To find $\text{Long} \left( (a \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow c \right)$

$$\Gamma_2, x_3 : a \vdash X'_1 : b$$  \hfill (B)

$$\Gamma_2 \vdash X_1 : a \rightarrow b$$  \hfill (A)

$$\Gamma_2 \vdash x_1 X_1 : c$$  \hfill (C)

$$x_1 : (a \rightarrow b) \rightarrow c, x_2 : b \rightarrow a \vdash X' : c$$  \hfill (B)

$$\vdash X : ((a \rightarrow b) \rightarrow c) \rightarrow (b \rightarrow a) \rightarrow c$$  \hfill (A)

fig. 4.3

From fig. 4.3, we see that the only branch, and so the tree, dies as $x_3 : a, x_2 : b \rightarrow a$ and $x_1 : (a \rightarrow b) \rightarrow c$ do not have tail $b$.

i.e. $\text{Long} \left( ((a \rightarrow b) \rightarrow c) \rightarrow (b \rightarrow a) \rightarrow c \right) = \emptyset$
Example 4.3

To find Long \(((a \rightarrow (b \rightarrow c)) \rightarrow d) \rightarrow (a \rightarrow b \rightarrow c) \rightarrow a \rightarrow d\)

\[
\begin{array}{c}
\text{To find Long } ((a \rightarrow (b \rightarrow c) \rightarrow d) \rightarrow (a \rightarrow b \rightarrow c) \rightarrow a \rightarrow d)\\
\end{array}
\]
Example 4.4

To find long \(((a \rightarrow b \rightarrow c) \rightarrow c) \rightarrow (a \rightarrow c) \rightarrow c\)

\[
\begin{align*}
\Gamma_4, x_5 : a, x_6 : b & \vdash X_4 : c \quad (B) \\
\frac{\text{\(X_3 = X_3\)}}{\Gamma_4 \vdash X_3 : a} \quad (A) \\
\Gamma_2, x_3 : a, x_4 : b & \vdash x_2 X_3 : c \quad (C) \\
\frac{\text{this branch dies}}{\Gamma_2 \vdash x_2 X_3 : c} \quad (B) \\
\Gamma_2 & \vdash x_1 X_2 : c \\
\frac{\text{fig. 4.5}}{\Gamma_2 \vdash X : ((a \rightarrow b \rightarrow c) \rightarrow c) \rightarrow (a \rightarrow c) \rightarrow c} \quad (A)
\end{align*}
\]

Long \(((a \rightarrow b \rightarrow c) \rightarrow c) \rightarrow (a \rightarrow c) \rightarrow c\)

\[
= \{ \lambda x_1 x_2. x_1(\lambda x_3 x_4. x_2 x_3), \lambda x_1 x_2. x_1(\lambda x_3 x_4. x_1(\lambda x_5 x_6. x_2 x_5)), \\
\lambda x_1 x_2. x_1(\lambda x_3 x_4. x_1(\lambda x_5 x_6. x_1(\lambda x_7 x_8. x_2 x_7))), \ldots \} 
\]
This example shows that the type repetition can be eliminated from a TMC sub tree but a limited type repetition cannot always be eliminated.

**Note** that the proof obtained without type repetition,
\[ \lambda x_1 x_2 . x_1 (\lambda x_3 x_4 . x_2 x_3) \]
has as principal type scheme: \(((a \rightarrow b \rightarrow c) \rightarrow d) \rightarrow (a \rightarrow c) \rightarrow d\)
of which the required formula is a substitution instance. All the other solutions have the required formula as principal type scheme.

**Example 4.5:**

To find \(\text{Long}(a \rightarrow (b \rightarrow c) \rightarrow a)\)

\[
\Gamma_2 \vdash x_1 : a \\
x_1 : a, x_2 : b \rightarrow c \vdash X : a \\
\vdash X : a \rightarrow (b \rightarrow c) \rightarrow a
\]

fig. 4.6

We get \(\text{Long}(a \rightarrow (b \rightarrow c) \rightarrow a) = \{ \lambda x_1 x_2 . x_1 \}\)

But \(\lambda x_1 x_2 . x_1 : a \rightarrow (b \rightarrow a)\)

i.e. Again the \(\lambda\) - term obtained by the algorithm has a principal type scheme of which the theorem we wanted to prove is a substitution instance.

Now, we will show when an implicational formula has an infinite number of proofs.
Theorem 4.2

If a TI - tree with root \( \vdash X : \tau \) has a solution for \( X \), it has infinite number of long solutions for \( X \) iff it has a TMC - subtree with an LTR branch. Whether \( \text{Long}(\tau) \) is infinite can be decided by examining at most \((|\tau| - 1) \| \tau \| (B)\) nodes in each branch above the root of the TI - tree.

Proof :

Suppose that we have the shortest TMC - subtree with a TR - branch as in the figure 4.7 and the corresponding TA\( _\lambda \) deduction in figure 4.8:

\[
\begin{align*}
\Gamma^+ & \vdash Y : a \\
\vdots & \\
\vdots & \\
\Gamma & \vdash Z : a \\
\vdots & \\
\vdash X : \tau & \\
\text{fig. 4.7} & \\
\Gamma^+ & \vdash Y^* : a \\
\vdots & \\
\vdots & \\
\Gamma & \vdash Z^* : a \\
\vdots & \\
\vdash X^* : \tau & \\
\text{fig. 4.8} &
\end{align*}
\]

We have \( \text{FV}(\Gamma^+) \supseteq \text{FV}(\Gamma) \supseteq \text{FV}(Z^*) \), so \( Z^* \) is an alternative solution for \( Y \) in fig.4.7 and as \( Y \) is a proper part of \( Z \) and so \( Y^* \) will be a proper part of \( Z^* \) and hence \( [Z^*/Y^*] Z^* \) is a distinct additional solution for \( Z \), giving a new solution for \( X \). The new TA\( _\lambda \) deduction providing this new solution will have the deduction in fig 4.8 down to
\[ \Gamma^+ \vdash Y^* : a \] replaced by the deduction down to \[ \Gamma \vdash Z^* : a \], with \[ \Gamma^+ \] replacing \[ \Gamma \] and below the old \[ \Gamma^+ \vdash Y^* : a \] each \[ Y^* \] is replaced by \[ Z^* \].

As \[ [Z^*/Y^*][Z^*/Y^*] Z^* \], ...etc will provide further distinct solutions for \( Z \) and hence for \( X \), then the number of such solutions is infinite.

In any TMC - subtree we need to examine only \( \parallel \tau \parallel (B) \) nodes in each branch to check on limited type repetition, however by Theorem 4.1 up to \( (l \tau l - 1 ) \parallel \tau \parallel (B) \) nodes still need to be checked in each branch of the \( TI - \text{tree} \) before we are certain whether or not there is a solution to \( \vdash X : \tau \) at all.

**Example 4.5** To find long \(((a \rightarrow a) \rightarrow a \rightarrow a)\)

\[
\begin{array}{c}
\vdash \\
\Gamma_2 \vdash X_1 : a \\
\Gamma_2 \vdash X_1 : a \\
\Gamma_2 \vdash x_1X_1 : a \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma_2 \vdash x_2 : a \\
\end{array}
\]

\[
\vdash X : (a \rightarrow a) \rightarrow a \rightarrow a
\]

\[ X' = x_2 \]

**fig. 4.9**
The tree terminates at $\Gamma_2 \vdash x_2 : a$ above the bottom (B) node. The type $a$ and $\Gamma_2$ at that (B) node are repeated in the other branch above that (B) node, so $\text{Long} ((a \rightarrow a) \rightarrow a \rightarrow a)$ is infinite.

Note that Bunder [4] gives an upper bound of $(|\tau| - 1)!^{\tau!}(|\tau| - 1)$ (B) nodes in total that might be generated in a TI-tree without type repetition. This bound however is very much larger than the number of (B) nodes generated in any example that we have tried.

We now extend the $\text{Long} (\tau)$ Generating Algorithm to cover some weaker logics than $\text{SK}$.

§ 4.4 Subintuitionistic Implicational Logics

Chapter 3 showed us that if we start with a set of combinators weaker than $\text{SK}$, we can only generate a subset of the set of all $\lambda$-terms $\Lambda$.

To find Q-inhabitants of a given type $\tau$, we therefore have to adapt the $\text{SK}$-Algorithm to look for only the lambda terms in the appropriate subset. (Q-inhabitants are inhabitants that are Q-terms.)

Logics without W

A combinator defined in terms of $I, B, B', C$ or $K$, has by Theorems 3.1 and 3.4 a corresponding $\lambda$-term in $\text{Once}^\sim()$.

As each (C) node in the TI-tree (see §4.2) introduces a variable $x_i$ into a solution, for the solution to be in $\text{Once}^\sim()$, no branch can have two (C) nodes introducing that variable. The $\text{BCK}$-Algorithm is therefore the
SK - Algorithm or Long (τ) - Generating Algorithm with each branch restricted to having no variable introduced twice.

The BCI - Algorithm has the additional restriction that in every subterm $λx_i . M$ that is formed, $x_i ∈ FV(M)$.

The BB’IK - Algorithm is the BCK - Algorithm except that only terms in HRM ( ) obtained from the restricted TI - tree are solutions.

The BB’T - Algorithm is the BCI - Algorithm except only terms in HRM ( ) are solutions.

Theorem 4.3

If $I - Y : τ$ has a solution for $Y$ as a BCK, BCI, BB’T or BB’IK - λ term, then there is a terminating $W$ - pruned MC - subtree above this root with no branches having more than $|τ|/2$ (B) nodes, that provides that solution. The set of all such solutions is finite.

Proof :

As all the variables introduced in a branch of the TI - tree in the BCK, BCI, BB’T and BB’IK Algorithms, must be distinct, the number of distinct variables that can be used is by Theorem 4.1, the number of long negative subtypes of $τ$. It is easy to show, by induction on $|τ|$, that this number is less than $|τ|/2$ (see Bunder [4] §4.)

Theorem 4.4

The maximum number of (B) nodes that need to be examined to determine whether $I - X : τ$ has a BCK, BB’T, BB’IK or BCI solutions for $X$, or an infinite number of such solutions is $(|τ| - 2)^{|τ|/4}$ where $[|τ|/4]$ is the integral part of $|τ|/4$. 

Theorems 4.3 and 4.4 provide us with decision procedure for BCK, BCI, BB'I and BB'IK - logics.

Logics without K but with W [ BB'IW and BCIW ]

A combinator defined without K has, by Theorems 3.1 and 3.2, a corresponding \( \lambda \) - term in \( \text{Once}^+() \). In a TI - tree of \( \vdash X : \tau \) we can prune any SK - solutions that are not of this form and we will eventually obtain a BCIW - solution if there is one, but it seems that we cannot bound the search in the way we did for earlier logics using type repetition. For example, in figure 4.10, with \( S(\Gamma_m) = S(\Gamma_n) \)

\[
\begin{array}{c}
\Gamma_m \vdash Z' : \rightarrow a \\
\Gamma_n \vdash X_2 : \beta_p \\
\Gamma_n \vdash Y_1 : \beta_j \\
\Gamma_n \vdash x_i Y_1 Y_2 : a \\
\Gamma_n \vdash X' : a \\
\vdash X' : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow a
\end{array}
\]

fig. 4.10
If there is a BCIW - λ - term solution $X'*$ for $X'$, this may involve an $x_p$ ($p \leq n$) which is part of a solution $Y_1*$ for $Y_1$, but not part of any solution for $Y_2$ or for $Z'$. If we replace variables from $\Gamma_m - \Gamma_n$ that appear in $Z'*$ the solution for $Z'$ by ones in $\Gamma_n$ we get a new (SK-) solution for $Z'$ which is also an SK- solution for $X'$. This will not contain $x_p$ and so does not provide a BCIW - solution for $X$. Thus the algorithm does not supply us with a decision procedure for BCIW - logic ($R_{\rightarrow}$) or similarly for BB'IW logic ($T_{\rightarrow}$).

Note that BCIW logic is decidable but is of extremely large complexity.

Note that the algorithm given in this chapter has been implemented by A.H. Dekker [8], to provide a decision procedure and proofs for intuitionistic implicational logic SK, BCK and BCI.

Next, we will extend this algorithm to cover logics with forms of negation and classical implication.
CHAPTER 5

Implicational logics with weak negation

In this chapter, we extend the Ben-Yelles algorithm to some logics with connectives other than $\rightarrow$, such as negation, conjunction and disjunction, which could be defined in terms of $\rightarrow$ and a type constant $f$, as follows:

Definition 5.1:

(i) $\sim A = A \rightarrow f$
(ii) $A \& B = \sim (A \rightarrow \sim B)$
    $= (A \rightarrow B \rightarrow f) \rightarrow f$
(iii) $A \land B = \sim ((A \rightarrow B) \rightarrow \sim A)$
     $= ((A \rightarrow B) \rightarrow A \rightarrow f) \rightarrow f$
(iv) $A \lor B = \sim A \rightarrow B$
     $= (A \rightarrow f) \rightarrow B$
(v) $A \lor_1 B = (A \rightarrow B) \rightarrow B$
(vi) $\Diamond A = A \rightarrow A \rightarrow A$

The algorithm for finding proofs in $\lambda$-calculus of any theorem containing any combination of negation, conjunction and disjunction (as above) is the same as the Ben-Yelles algorithm in the previous chapter in which $f$ acts like a type variable.
Thus, by using translation algorithms * in Chapter 3 we can find proofs in the Hilbert style, using combinators in different implicational logics such as SK, BCK, BCI, BB'1, ... etc.

**Example 5.1:**

To find the proof of

\[ \Gamma \vdash (a \rightarrow b) \rightarrow \neg b \rightarrow \neg a \]

i.e. \[ \Gamma \vdash (a \rightarrow b) \rightarrow (b \rightarrow f) \rightarrow (a \rightarrow f) . \]

We have

\[
\begin{align*}
X_2 &= x_3 \\
\Gamma_3 &\vdash X_2 : a \\
\Gamma_3 &\vdash x_1 X_2 : b \\
\Gamma_3 &\vdash X_1 : b \\
\Gamma_3 &\vdash x_2 X_1 : f \\
\end{align*}
\]

\[(C) \quad (A) = (B) \quad (C) \]

\[
\begin{align*}
x_1 : a \rightarrow b \; , \; x_2 : b \rightarrow f \; , \; x_3 : a & \vdash X \vdash : f \\
\vdash X : (a \rightarrow b) \rightarrow (b \rightarrow f) \rightarrow a \rightarrow f . \\
\end{align*}
\]

fig. 5.1

From figure 5.1 we can see that the proof is

\[ X = \lambda x_1 x_2 x_3 . x_2( x_1 x_3) \]

which is translatable into \textbf{SK, BCK, BCI} and \textbf{BB'1} - logics.
Note that "Brouwer 6.0" (see Dekker [8]) performs this algorithm as well as the translations for SK, BCK and BCI logics.

Further examples which are included in Theorems 5.1 and 5.2 can also be generated by Brouwer 6.0.

Theorem 5.1

The following are theorems of the given logics below but not of the others in the set \{SK, BCK, BCI, BCIW, BB'I, BB'IW, BB'IK\}:

1. \( a \rightarrow \neg \neg a \) in SK, BCK, BCI and BCIW - logics
2. \((a \rightarrow b) \rightarrow \neg b \rightarrow \neg a\) in SK, BCK, BCI, BCIW, BB'I, BB'IW and BB'IK.
   [or in all logics that contain B' or another combinator with the principal type of B'.]
3. \((a \rightarrow \neg a) \rightarrow \neg a\) in SK, BCIW and BB'IW.
   [or in all logics that contain W or another combinator with the principal type of W.]
4. \((a \rightarrow \neg b) \rightarrow b \rightarrow \neg a\) in SK, BCIW, BCI and BCK.
   [or in all logics that contain C or have another combinator with the principal type of C.]
5. \(a \rightarrow b \vee a\) in all logics with K.
6. \((a \rightarrow b) \& (a \rightarrow c) \rightarrow a \rightarrow b \& c\) in SK and BCIW
7. \((a \rightarrow b) \land (a \rightarrow c) \rightarrow a \rightarrow b \land c\) in SK
8. \(a \rightarrow \neg a \rightarrow \neg a\) in SK.
(9) $\Diamond a \& \Diamond b \rightarrow \Diamond (a \& b)$ in $\text{SK, BCK, BCI, BCIW, BB}^{'1}, \text{BB}^{'1}W \text{and BB}^{'1}K$.

(10) $\Diamond a \land \Diamond b \rightarrow \Diamond (a \land b)$ in $\text{SK}$

**Proof:**

In each case, this follows from the given $\lambda$-proof and the translations of Chapter 3;

(1) $\lambda x_1x_2 . x_2x_1 : a \rightarrow (a \rightarrow f) \rightarrow f$

(2) $\lambda x_1x_2x_3 . x_2(x_1x_3) : (a \rightarrow b) \rightarrow (b \rightarrow f) \rightarrow a \rightarrow f$

(3) $\lambda x_1x_2 . x_1x_2x_2 : (a \rightarrow a \rightarrow f) \rightarrow a \rightarrow f$

(4) $\lambda x_1x_2x_3 . x_1x_3x_2 : (a \rightarrow b \rightarrow f) \rightarrow b \rightarrow a \rightarrow f$

(5) $\lambda x_1x_2 . x_1 : a \rightarrow (b \rightarrow f) \rightarrow a$

(6) $\lambda x_1x_2x_3 . x_1(\lambda x_4 x_5 . x_3(x_4x_2)(x_5x_2)) :$

\(((a \rightarrow b) \rightarrow (a \rightarrow c) \rightarrow f) \rightarrow f) \rightarrow a \rightarrow (b \rightarrow c \rightarrow f) \rightarrow f$

(7) $\lambda x_1x_2x_3 . x_1(\lambda x_4x_5 . x_3(\lambda x_6 . x_4x_5x_2)(x_5x_2)) :$

\(((a \rightarrow b) \rightarrow a \rightarrow c) \rightarrow(a \rightarrow b) \rightarrow f) \rightarrow f) \rightarrow a \rightarrow ((b \rightarrow c) \rightarrow b \rightarrow f) \rightarrow f$

(8) $\lambda x_1x_2x_3 . x_2x_3 : a \rightarrow (a \rightarrow f) \rightarrow a \rightarrow f$
(9) $\lambda x_1 x_2 x_3 x_4. x_1(\lambda x_5 x_6. x_3(\lambda x_7 x_8. x_2(\lambda x_9 x_{10}. x_4(\lambda x_7 x_9)(x_6 x_8 x_{10}))))$:

$$(((a \rightarrow a \rightarrow a) \rightarrow (b \rightarrow b \rightarrow b) \rightarrow f) \rightarrow f) \rightarrow (((a \rightarrow b \rightarrow f) \rightarrow f) \rightarrow ((a \rightarrow b \rightarrow f) \rightarrow f) \rightarrow (a \rightarrow b \rightarrow f) \rightarrow f$$

(10) $\lambda x_1 x_2 x_3 x_4. x_2(\lambda x_5 x_6. x_3(\lambda x_7 x_8. x_1(\lambda x_9 x_{10}. x_4(\lambda x_{11}. x_9 x_{10}(x_5 x_6)(x_7 x_{11}))) x_8))$:

$$(((a \rightarrow a \rightarrow a) \rightarrow b \rightarrow b \rightarrow b) \rightarrow (a \rightarrow a \rightarrow a) \rightarrow f) \rightarrow f) \rightarrow (((a \rightarrow b) \rightarrow a \rightarrow f) \rightarrow f) \rightarrow (((a \rightarrow b) \rightarrow a \rightarrow f) \rightarrow f) \rightarrow ((a \rightarrow b) \rightarrow a \rightarrow f) \rightarrow f.$$

**Theorem 5.2**

The Ben-Yelles algorithm shows that the following are not types of any $\lambda$-terms, and so not theorems of $\text{SK}$ or any of its sublogics :-

(1) \[ \vdash (\sim a \rightarrow \sim b) \rightarrow b \rightarrow a \]
(2) \[ \vdash \sim \sim a \rightarrow a \]
(3) \[ \vdash a \lor a \rightarrow a \]
(4) \[ \vdash a \rightarrow a \lor b \]
(5) \[ \vdash a \lor b \rightarrow b \lor a \]
(6) \[ \vdash a \& b \rightarrow a \]
(7) \[ \vdash a \& b \rightarrow b \]
(8) \[ \vdash a \land b \rightarrow a \]
(9) \[ \vdash a \land b \rightarrow b \]
(10) \[ \vdash (a \rightarrow c) \land (b \rightarrow c) \rightarrow (a \lor b) \rightarrow c \]
(11) \[ \vdash (a \rightarrow c) \land (b \rightarrow c) \rightarrow (a \lor b) \rightarrow c \]
(12) \[ \vdash (a \rightarrow c) \land (b \rightarrow c) \rightarrow (a \lor_1 b) \rightarrow c \]
(13) \[ \vdash (a \rightarrow c) \land (b \rightarrow c) \rightarrow (a \lor_1 b) \rightarrow c \]
(14) \[ \vdash a \land (b \lor c) \rightarrow (a \land b) \lor c \]
(15) \[ \vdash a \land (b \lor c) \rightarrow (a \land b) \lor c \]
(16) \[ \vdash a \land (b \lor_1 c) \rightarrow (a \land b) \lor_1 c \]
(17) \[ \vdash a \land (b \lor_1 c) \rightarrow (a \land b) \lor_1 c. \]
Chapter 6

Logics with Implication and Intuitionistic Negation

§ 6.1 Introduction

In this chapter, we extend the Ben-Yelles algorithm to implicational logics with the constant \( f \) and the extra axiom \( \vdash f \rightarrow \alpha \).

In \( \lambda \)-calculus

Here \( f \) can be considered as an empty type and the natural deduction proofs can be extended to include a new rule \( f_e \) as follows:

\[
\text{Rule } f_e \\
\Delta \vdash X : f \\
\frac{}{\Delta \vdash X : \alpha}
\]

for any \( \alpha \).

In combinatory logic

We extend combinatory logic to include an extra "combinator" \( f_e \) with the type

\[ f_e : f \rightarrow \alpha \quad \text{for any } \alpha. \]

In certain logics a simplified version of the axiom and the rule may be used.
**Theorem 6.1**

In logics that have $K$, the use of $\vdash f \rightarrow \alpha$ can be replaced by $\vdash f \rightarrow a$ where $a$ is the atom which is the tail of $\alpha$.

**Proof :**

(i) In $\lambda$-calculus:

If $\alpha = \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow a$, where $a$ is an atom, then a deduction

\[
\begin{array}{c}
\vdash Y : f \\
\vdash Y : \alpha
\end{array}
\]

\[
\begin{array}{c}
\vdash X_1 \ldots X_n. Y : a
\end{array}
\]

\[
\begin{array}{c}
\vdash \lambda x_n. Y : \alpha_n \rightarrow a
\end{array}
\]

\[
\begin{array}{c}
\vdash \lambda x_1 \ldots x_n. Y : \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow a
\end{array}
\]

Here, $x_1, \ldots, x_n$ must be variables not in $\Delta$ (otherwise $\Delta$ would change at each $\rightarrow_i$ step) and so not free in $Y$. Thus to make $\lambda x_1 \ldots x_n. Y$ translatable into combinators we must have $K$ in the logic.
(ii) In Combinatory logic

\[ \vdash f_e : f \to \alpha \]

can be replaced by:

\[(I)\]

\[ \vdash y : f \quad \vdash f_e : f \to a \]

\[ \vdash f_e y : a \quad \vdash K : a \to \alpha_n \to a \]

\[ \vdash K(f_e y) : \alpha_n \to a \quad \vdash K : (\alpha_n \to a) \to \alpha_{n-1} \to \alpha_n \to a \]

\[ \vdash K(...K(f_e y)...)) : \alpha_2 \to \ldots \to \alpha_n \to a \]

\[ \vdash K(K(...K(f_e y)...)) : \alpha_1 \to \ldots \to \alpha_n \to a. \]

\[ \vdash \lambda^* y. K(K(...K(f_e y)...)) : f \to \alpha. \]

fig. 6.2

So again \( K \) is needed.

Example 6.1 :

Consider

\[(I)\]

\[ x_1 : a \to f \]

\[ \to_e \]

\[ x_1 x_2 : f \]

\[ f_e \]

\[ x_1 x_2 : b \to c \]

\[ \to_i \]

\[ \lambda x_2. x_1 x_2 : a \to b \to c \]

\[ \to_i \]

\[ \lambda x_1 x_2. x_1 x_2 : (a \to f) \to a \to b \to c \]

fig. 6.3
This can be written as:

\[
\begin{align*}
&\text{(1)} & x_1 : a \rightarrow f & x_2 : a & x_3 : b \\
&\rightarrow_e \quad \text{-------------------------} & x_1x_2 : f & f_e \quad \text{----------------} & x_1x_2 : c \\
&\rightarrow_i \quad \text{-------------------------} & (3) & \lambda x_3 : x_1x_2 : b \rightarrow c \\
&\rightarrow_i \quad \text{-------------------------} & (2) & \lambda x_2x_3 : x_1x_2 : a \rightarrow b \rightarrow c \\
&\rightarrow_i \quad \text{-------------------------} & (1) & \lambda x_1x_2x_3 : x_1x_2 : (a \rightarrow f) \rightarrow a \rightarrow b \rightarrow c
\end{align*}
\]

fig. 6.4

Thus both \( \lambda \)-terms \( \lambda x_1x_2 : x_1x_2 \) and \( \lambda x_1x_2x_3 : x_1x_2 \) represent a proof of \( \vdash \sim a \rightarrow a \rightarrow b \rightarrow c \).

They are also proofs of \( (a \rightarrow b) \rightarrow (a \rightarrow b) \) and \( (a \rightarrow c) \rightarrow a \rightarrow b \rightarrow c \), so it becomes clear that the introduction of rule \( f_e \) means that a \( \lambda \)-term no longer uniquely represents a proof.

The translation of these \( \lambda \)-terms into combinators gives I and BK (without including \( f_e \)) which are proofs of \( a \rightarrow a \) and \( (a \rightarrow c) \rightarrow a \rightarrow b \rightarrow c \).

To ensure that proofs more closely represent theorems, we will use the combinatory logic axiom \( f_e \) in the \( \lambda \)-calculus insted of rule \( f_e \). The \( \lambda \)-calculus is therefore extended by the constant \( f_e \).
Note that in Chapter 4, given an intuitionistic implicational theorem \( \tau \) we get, from the Ben-Yelles algorithm, the proof of \( \tau \) or of a theorem of which \( \tau \) is a substitution instance.

The above example now becomes:

\[
\begin{align*}
(1) & \quad x_1 : a \rightarrow f \\
(2) & \quad x_2 : a \\
(3) & \quad x_3 : b \\
\Rightarrow & \quad \text{from Ben-Yelles algorithm, the proof of } f \text{ or of } \text{a theorem of which } \tau \text{ is a substitution instance.}
\end{align*}
\]

\[\to e \quad \frac{\inferenceRule{\rightarrow e}{x_1 \land x_2 : f}}{x_1 \land x_2 : f} \]

\[\to i \quad \frac{\inferenceRule{\rightarrow i}{f : f \rightarrow b \rightarrow c}}{f(x_1x_2) : b \rightarrow c} \]

\[\to i \quad \frac{\inferenceRule{\rightarrow i}{\lambda x_2 \cdot f(x_1x_2) : a \rightarrow b \rightarrow c}}{\lambda x_1 x_2 \cdot f(x_1x_2) : (a \rightarrow f) \rightarrow a \rightarrow b \rightarrow c} \]

fig. 6.5

where \( \lambda x_1 x_2 \cdot f(x_1x_2) \equiv Bf_e \) and \( Bf_e : (a \rightarrow f) \rightarrow a \rightarrow b \)
of which the required proof is a substitution instance.

The proof using only \( f \rightarrow \text{atom} \) gives \( \lambda x_1 x_2 x_3 \cdot f(x_1x_2) \)
which can be translated into \( B(BK)(Bf_e) \), either of which is proof of
\( (a \rightarrow f) \rightarrow a \rightarrow b \rightarrow c \).

Note that the example shows that we cannot eliminate uses of \( f \rightarrow \alpha \)
where \( \alpha \) is composite in weaker logics that do not have \( K \).

Now we can modify the Long(\( \tau \)) generating algorithm for logics with
\( f_e : f \rightarrow a \) (\( a \) an atom)
as follows:
Chapter 6  Logics with Implication and Intuitionistic negation

§ 6.2 The SKfe Long(\(\tau\)) Generating Algorithm

**Aim:** To find all long \(X\)'s such that \(\text{FV}(X) \subseteq \{x_1, \ldots, x_{m-1}\}\) and for \(m \geq 1\), using the axiom \(\vdash \text{fe} : f \rightarrow a\) (\(a\) an atom),

\[x_1 : \tau_1, \ldots, x_{m-1} : \tau_{m-1} \vdash X : \tau\]  \(\text{-(A)}\)

(At this stage \(X\) in (A) is an unknown \(\lambda\) - term.)

**Method:**

Let \(\tau = \tau_m \rightarrow \ldots \rightarrow \tau_n \rightarrow a\) where \(a\) is an atom and \(n \geq m - 1\).

Then \(X\) must be of the form

\[X = \lambda x_m \ldots x_n \cdot X' \quad \text{or} \quad X = \lambda x_m \ldots x_n \cdot \text{fe}X'\]

where for \(m \leq i \leq n\), \(x_i : \tau_i\).

Thus (A) is solved if we can solve:

\[x_1 : \tau_1, \ldots, x_n : \tau_n \vdash X' : a \quad \text{or} \quad x_1 : \tau_1, \ldots, x_n : \tau_n \vdash X' : f\]  \(\text{-(B)}\)

where \(X'\) is an unknown \(\lambda\) - term no longer than \(X\).

Let \(\tau_{i_1}, \ldots, \tau_{i_p}\) be the \(\tau_i\)'s \((1 \leq i \leq n)\) that have \(\text{tail}(\tau_i) = a\) and \(\tau_{j_1}, \ldots, \tau_{j_q}\) be those with \(\text{tail}(\tau_i) = f\) and let for \(1 \leq k \leq p\) or \(q\)

\[\tau_{i_k} = \tau_{k,1} \rightarrow \ldots \rightarrow \tau_{k,r_k} \rightarrow a \quad \text{or} \quad \tau_{j_k} = \tau'_{k,1} \rightarrow \ldots \rightarrow \tau'_{k,t_k} \rightarrow f\]

Then \(X'\) must have the form, for at least one \(k\),

\[X' = x_{i_k} X_{k,1} \ldots X_{k,r_k} \quad \text{or} \quad x_{j_k} X_{k,1} \ldots X_{k,t_k}\]

and so a solution for any \(k \,(1 \leq k \leq p)\) of

\[x_1 : \tau_1, \ldots, x_n : \tau_n \vdash x_{i_k} X_{k,1} \ldots X_{k,r_k} : a \]

or

\[x_1 : \tau_1, \ldots, x_n : \tau_n \vdash x_{j_k} X_{k,1} \ldots X_{k,t_k} : f\]  \(\text{-(C)}\)

for \(X_{k,1}, \ldots, X_{k,r_k}\) or \(X_{k,1,1}\) provides a solution to (B).
Thus there is a solution to (B) and so to (A) if for any \( k, \) \((1 \leq k \leq p\) or \( q)\), we have for all \( s \) such that \( 1 \leq s \leq r_k \) or \( t_k \) solutions to

\[
x_1 : \tau_1, \ldots, x_n : \tau_n \vdash X_{k,s} : \tau_{k,s}.
\]

and (D) is solved as (A) by the same steps as above.

**Note that** this means that we have in general more (C) steps than that in the Ben-Yelles algorithm in Chapter 4.

The algorithm produces a type-inhabitant (or TI) tree as in the following figure where we use the abbreviation

\[
\Gamma_j = \{x_1 : \tau_1, \ldots, x_j : \tau_j\}.
\]

\[
\begin{array}{c}
\Gamma_n \vdash x_{i_1} X_{1,1} \ldots X_{1,r_1} : a \ldots \\
\ldots \Gamma_n \vdash x_{i_p} X_{p,1} \ldots X_{p,r_p} : a.
\end{array}
\]

\[
\begin{array}{c}
\Gamma_n \vdash x_{j_1} X_{1,1} \ldots X_{1,t_1} : f \ldots \\
\ldots \Gamma_n \vdash x_{j_q} X_{q,1} \ldots X_{q,t_q} : f (C)
\end{array}
\]

\[
\begin{array}{c}
\Gamma_n \vdash X' : f. \vdash f_e : f \rightarrow a (B') \\
\Gamma_n \vdash f_e X' : a (B)
\end{array}
\]

\[
\Gamma_m \vdash X : \tau (A)
\]

fig. 6.6
Towards an alternative algorithm

Instead of using the axiom $f_e : f \rightarrow c$ as an axiom in the proof of $\tau$ perhaps for several $c$’s, we can introduce a new variable $y_i$ as an inhabitant of each use of $f \rightarrow c_i$ in the proof of $\tau$ where $c_i$’s are type variables of $\tau$ other than $f$. This gives an SK proof of

$$y_1 : (f \rightarrow c_1), \ldots, y_k : (f \rightarrow c_{k-1}) \vdash X : \tau$$

A given $f \rightarrow c_i$ may be used zero or more times and in general we have

$$\vdash \lambda y_1 \ldots y_{k-1}. X : (f \rightarrow c_1) \rightarrow \ldots (f \rightarrow c_{k-1}) \rightarrow \tau$$

where $\lambda y_1 \ldots y_{k-1}. X$ is SK-definable but not necessarily definable in logics that do not have both K and W. We therefore have:

The second SK$f_e$ algorithm

Aim To prove $\tau$.

Step 1 Identify the distinct type variables $c_1, \ldots, c_k$ of $\tau$, where $c_k$ is $f$.

Step 2 Find $X$ such that $\vdash X : (f \rightarrow c_1) \rightarrow \ldots (f \rightarrow c_{k-1}) \rightarrow \tau$ by the SK-algorithm.

Then $Xf_ef_e \ldots f_e$ (with $(k-1)$ $f_e$’s) is a proof of $\tau$ and $X^*f_ef_e \ldots f_e$ will represent the Hilbert-style proof.
§ 6.3 Finding Proofs in SKfe

Theorem 6.2
(i) \( \vdash X : \tau \) has an SKfe solution for \( X \) iff the TI - tree with \( \vdash X : \tau \) at the root has a TMC subtree with no TR - branches.
(ii) If \( \vdash X : \tau \) has a solution for \( X \) in SKfe, then there is a TMC subtree of its TI - tree in which each branch has at most
\[
(1 \tau\| + 2\| \tau\| - 3) \| \tau\| \text{ (B) nodes.}
\]

Proof:
(i) The same as in Theorem 4.1(i).
(ii) From the second algorithm, if \( c_1, \ldots, c_n \) are the atoms of \( \tau \) other than \( f \), we have
\[
1 (f \to c_1) \to \ldots (f \to c_n) \to \tau 1 = 1 \tau + 2(\| \tau \| - 1)
\]
and
\[
\| (f \to c_1) \to \ldots (f \to c_n) \to \tau\| = \| \tau\| + 1.
\]
So, substituting into Theorem 4.1 (ii) we have that the upper bound on the number of (B) nodes is
\[
(1 \tau\| + 2\| \tau\| - 3) \| \tau\| .
\]

If \( f \) is not in \( \tau \), the SK - algorithm and the upper bound of Theorem 4.1 can be used.

Corollary:
If in the TI - tree with \( \vdash X : \tau \) as root, there is no solution for \( X \) in SKfe in any subtree without type repetition, then there is no solution at all. The maximal number of (B) nodes in any branch that needs to be checked is
\[
(1 \tau\| + 2\| \tau\| - 3) \| \tau\| .
\]
By this corollary, we have a decision procedure for a logic with implication and intuitionistic negation and we can generate a proof of any provable formula in that logic.

**Theorem 6.3:**

\[ a \rightarrow a \lor b \] is a theorem in \( \text{SKfe} \) if we define

\[ a \lor b = (a \rightarrow f) \rightarrow b. \]

**Proof:**

By Theorem 5.2 (4) \( \vdash a \rightarrow \neg a \rightarrow b \) has no proof in \( \text{SK} \) logic. But by using axiom \( f_e : f \rightarrow b \) at the second (B) node, as in fig. 6.7, we will find the proof

\[
\lambda x_1 x_2 . f_e(x_2 x_1) \quad \text{in} \quad \text{SKfe}.
\]

\[
X_2 = x_1
\]

\[
\Gamma_2 \vdash X_2 : a \quad (A) = (B)
\]

\[
\Gamma_2 \vdash x_2 X_2 : f \quad (C)
\]

\[
\Gamma_2 \vdash X_1 : f , \quad f_e : f \rightarrow b \quad (A) = (B)
\]

\[
\Gamma_2 \vdash f_e X_1 : b \quad (C)
\]

\[
x_1 : a , \quad x_2 : a \rightarrow f \vdash X' : b \quad (B)
\]

\[
\vdash X : a \rightarrow (a \rightarrow f) \rightarrow b \quad (A)
\]

fig. 6.7.
Also, by using the second algorithm and Dekker [8] we have

\[ \lambda y_1y_2y_3y_4 . y_2(y_4y_3) : (f \to a) \to (f \to b) \to a \to (a \to f) \to b \]

in SK-logic, thus

\[ (\lambda y_1y_2y_3y_4 . y_2(y_4y_3))f \circ e : a \to (a \to f) \to b \]

in SKfe, which, in this case, reduces to the same proof.

**Theorem 6.4**

The following are not types of any \( \lambda f_e \) - terms, and so not theorems of SKfe or any of its sublogics (Note Definition 5.1) :-

1. \( \vdash \sim \sim a \to a \)
2. \( \vdash (\sim a \to \sim b) \to (b \to a) \)
3. \( \vdash a \lor a \to a \)
4. \( \vdash a \& b \to a \)
5. \( \vdash a \& b \to b \)
6. \( \vdash a \land b \to a \)
7. \( \vdash a \land b \to b \)
8. \( \vdash a \lor b \to b \lor a \)
9. \( \vdash (a \to c) \& (b \to c) \to (a \lor b) \to c \)
10. \( \vdash (a \to c) \land (b \to c) \to (a \lor b) \to c \)
11. \( \vdash (a \to c) \& (b \to c) \to (a \lor_1 b) \to c \)
12. \( \vdash (a \to c) \land (b \to c) \to (a \lor_1 b) \to c \)
13. \( \vdash a \& (b \lor c) \to (a \& b) \lor c \)
14. \( \vdash a \land (b \lor c) \to (a \land b) \lor c \)
15. \( \vdash a \& (b \lor_1 c) \to (a \& b) \lor_1 c \)
16. \( \vdash a \land (b \lor_1 c) \to (a \land b) \lor_1 c \).

**Proof** : By Dekker [8] and the second SKfe Algorithm.
§ 6.4 Weaker logics with implication and intuitionistic negation

In this section we are looking for a decision procedure for some weaker logics than $SK$-logic with the extra axiom $\vdash f \rightarrow \alpha$.

Logics without $W$ but with $K$:

By Theorem 6.1, we can replace the use of $\vdash f \rightarrow \alpha$ by $\vdash f \rightarrow a$ where $a$ is an atom (tail of $\alpha$). Hence the first $SKf_e$ algorithm with the appropriate restrictions mentioned in § 4.4 gives us a $BCKf_e$ and $BB'IKf_e$ Algorithm.

We can also use a version of the second $SKf_e$ algorithm as a $BCKf_e$ or $BB'IKf_e$ Algorithm. Instead of finding just any $X$ such that:

$\vdash X : (f \rightarrow c_1) \rightarrow \ldots (f \rightarrow c_k) \rightarrow \tau$

where $c_1, \ldots, c_k$ are the atoms of $\tau$ other than $f$, or equivalently, if $\tau = \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow a$, any $X'$ such that:

$x_1 : f \rightarrow c_1, \ldots, x_k : f \rightarrow c_k, x_{k+1} : \tau_1, \ldots, x_{k+n} : \tau_n \vdash X' : a$,

we need one in which $x_{k+1}, \ldots, x_{k+n}$ appear at most once. However $x_1, \ldots, x_k$ can appear more than once (as these can be replaced by $f_e$'s).

Therefore we have the following theorem:
Theorem 6.5

If \( \vdash Y : \tau \) has a solution for \( Y \), a \( \text{BCK}^e \) or \( \text{BB'}IK^e \) - \( \lambda \) term, then there is a TMC - subtree above this root with no branches having more than \( ((\|\tau\| + 1)\|\tau\| + \|\tau\| - 3)\|\tau\| \) nodes, that provides that solution. The set of all such solutions is finite.

Proof:

By Theorem 4.1 the number of \( (B) \) nodes in a branch that need to be checked is at most \( (|\beta| - 1)\|\beta\| \) where \( \beta = (f \to c_1) \to \ldots \to (f \to c_{k-1}) \to \tau \) which is \( (|\tau| + 2\|\tau\| - 3)\|\tau\| \).

Theorem 6.5 provides us with a decision procedure, which provides proofs, for \( \text{BCK}^e \) and \( \text{BB'}IK^e \) - logics. By drawing up a tree where all branches from the root are continued until they have \( (|\tau| + 2\|\tau\| - 3)\|\tau\| \) nodes unless they terminate or die before then, all \( \text{BCK}^e \) solutions are given and that in \( \text{BB'}IK^e \) can be recognised in this finite set.
Logics with $f_e$ but without $K$ ($BCIf_e$, $BB'Ife$ and $BCIWfe$)

Note that without $K$ our earlier methods do not work, in particular, we cannot change from $f \to \alpha$ to $f \to a$.

We can extend the SK algorithm to include extra (B) steps as in figure 6.8 below:

![Figure 6.8](image_url)

In this way we have a $BCIf_e$, $BB'Ife$ and a $BCIWfe$ algorithm. The number of (B) nodes in each branch will be no more than that given by Theorem 4.1, however the branches will be longer because of the extra nodes that appear between some (A) and (B) nodes. Also there will be for more (B) nodes as illustrated above.
Theorem 6.6

\[ a \rightarrow a \lor b \] is not a theorem of BB'If, BB'IKe or BB'IWF but it is a theorem of BCIf and BCKf if we define
\[ a \lor b = (a \rightarrow f) \rightarrow b. \]

Proof:

If \( f \rightarrow \alpha \) is allowed as an axiom there is one extra branch from the root of the tree in fig. 6.7 given by:

\[
x_1 : a \vdash X' : f \\
x_1 : f \vdash X' : (a \rightarrow f) \rightarrow b \\
| X : a \rightarrow (a \rightarrow f) \rightarrow b
\]

This clearly dies.

So the proof of this theorem in fig. 6.7 is the only one. It is an SKf, a BCKf and a BCIf proof but not a BB'IIf, BB'IKe or a BB'IWF proof.

Note that this kind of negation is called minimal negation. BB'I logic with this negation is called minimal by Anderson and Belnap [1]. Elsewhere SK- logic with this negation is referred to as minimal.
CHAPTER 7

Classical Implicational Logic (with Weak Negation)

§ 7.1 Introduction

In this chapter, we extend the Ben-Yelles algorithm for finding proofs in lambda calculus from the intuitionistic implicational logic to classical implicational logic by extending the typed lambda calculus to include a new operator $\forall$, similar to $\lambda$. Later on we include weak negation as in Chapter 5. First, we extend the definition of a term.

Definition 7.1 ($\lambda \forall$-terms)

(i) Any variable is a $\lambda \forall$-term.

(ii) If $x$ is a variable and $X$ and $Y$ are $\lambda \forall$-terms, then $(XY)$, $(\lambda x . X)$ and $(\forall x . X)$ are $\lambda \forall$-terms.

We will use the same abbreviations for terms involving $\forall$ as we did for terms involving $\lambda$.

Examples of $\lambda \forall$-terms:

(i) $\lambda x_1 x_2 . \forall x_3 . x_3 (x_2 x_1)$

(ii) $\lambda x_1 x_2 x_3 . x_2 (\forall x_4 . x_3 x_4 x_1)$

(iii) $\lambda x_1 . \forall x_2 . \lambda x_3 . x_2 (x_1 x_3)$

(iv) $\forall x_1 . \lambda x_2 . \forall x_3 . x_1 (x_2 x_3)$

(v) $\lambda x_1 . \forall x_2 x_3 . x_2 (x_1 x_3)$
Now, we state the introduction rule $V_i$.

\[ \frac{\text{Rule } V_i}{X : \alpha \rightarrow B} \]
\( \vdots \)
\( Y : \alpha \)
\( \text{--} \)
\( Vx \cdot Y : \alpha \)

The propositional (as type) part of this rule is the Peirce rule of Gentzen which extends intuitionistic implicational logic to classical logic.

**Combinatory logic**

In combinatory logic, instead of $\forall$, we add a new "combinator" $P$ (Peirce) with the type

$$P : ((\alpha \rightarrow \beta) \rightarrow \alpha) \rightarrow \alpha.$$ 

The type here is the Peirce axiom of classical logic.

**Translations**

The translations of Chapter 3 are extended to cover the $\lambda \forall$ calculus and $\text{SKP - combinatory logic}$ by adding:

\( (p) \quad (Vx \cdot X)_* = P(\lambda x \cdot X)_* \)
\( (p_1) \quad P_\lambda = \lambda y \cdot (Vx \cdot yx) \)

We abbreviate $P(\lambda^* x \cdot X_*)$ by $V^* x \cdot X_*$. 
§ 7.2 Proof reduction of $V_i$ in the $\lambda \nu$ - calculus

A reduction of classical proofs allows the postponement of all uses of the Peirce rule to the end of the proof as follows:

The deduction

$$\begin{align*}
(I) & \hspace{1cm} x : \alpha \rightarrow \beta \\
D_1 & \\
X : \alpha & \\
V_i & \hspace{1cm} \text{--------------------- (1)} \\
V_x . X : \alpha & \\
D_2 & \\
Z : \gamma & \\
\text{fig. 7.1}
\end{align*}$$

reduces to:

$$\begin{align*}
(2) & \hspace{1cm} u : \alpha \\
& \hspace{1cm} [u/Vx . X ] \ D_2 \\
z : \gamma \rightarrow \beta & \hspace{1cm} [u/Vx . X ] Z : \gamma \\
\rightarrow_e & \hspace{1cm} \text{-------------------------------------------} \\
z([u/Vx . X ] Z) : \beta
\end{align*}$$

$$\begin{align*}
\rightarrow_i & \hspace{1cm} \text{------------------------------------------- (2)} \\
\lambda u . z([u/Vx . X ] Z)/x & \hspace{1cm} \alpha \rightarrow \beta \\
[\lambda u . z([u/Vx . X ] Z)/x] D_1 & \\
[\lambda u . z([u/Vx . X ] Z)/x] X : \alpha & \\
[[\lambda u . z([u/Vx . X ] Z)/x] X /Vx . X ] D_2 & \\
[[\lambda u . z([u/Vx . X ] Z)/x] X /Vx . X ] Z : \gamma
\end{align*}$$

$$\begin{align*}
V_i & \hspace{1cm} \text{------------------------------------------- (1)} \\
Vz . [\lambda u . z([u/Vx . X ] Z)/x] X /Vx . X ] Z : \gamma & \\
\text{where } u, z \notin \text{ FV(ZX)}.
\end{align*}$$

fig. 7.2
This proof reduction suggest the general rule for V - postponement as follows:

\[(V_g) \quad Z \rightarrow Vz \cdot ([\lambda u \cdot z([u/Vx \cdot X] Z)/x]) X / Vx \cdot X] Z\]

where \(u, z \notin FV(ZX)\).

The following are special cases of this rule:

\[(V_1) \quad (Vx . M) N \rightarrow Vz \cdot ([\lambda u \cdot z(Nu)/x] M) N\]

where \(u, z \notin FV(NM)\).

\[(V_2) \quad N(Vx . M) \rightarrow Vz \cdot N([\lambda u \cdot z(Nu)/x] M)\]

where \(u, z \notin FV(NM)\).

\((V_1)\) and \((V_2)\) postpone a \(V_i\) - step past a \(\rightarrow_e\) step as major or a minor premise.

Further, rules such as the following have to be added to the rules of \(\lambda\) - calculus in Chapter 2 to get a \(\lambda V\) - calculus;

\[(\alpha_v) \quad Vx . M \rightarrow Vy . [y/x] M\] where \(y \notin FV(M)\).

\[(\xi_v) \quad M > N \Rightarrow Vx . M \rightarrow Vx . N\]

There are further proof reductions which motivate other \(\lambda V\) - reduction rules (see Bunder and Hirokawa [5]).

Example 7.1:

\[(I) \quad (2)\]

\[x : \alpha \quad y : \beta \rightarrow \gamma\]

\[D_1\]

\[M : \beta\]

\[V_i \quad \underline{(2)}\]

\[Vy . M : \beta\]

\[\rightarrow_i \quad \underline{(1)}\]

\[\lambda x . Vy . M : \alpha \rightarrow \beta\]
is reduced to

(1) \[ x : \alpha \]

(2) \[ z : \beta \]

(3) \[ \lambda x . z : \alpha \rightarrow \beta \]

\[ \rightarrow \text{e} \]

\[ u : (\alpha \rightarrow \beta) \rightarrow \gamma \]

\[ \rightarrow \text{i} \]

\[ \lambda z . u(\lambda x . z) : \beta \rightarrow \gamma \]

\[ [((\lambda z . u(\lambda x . z))/y) \ D_1 \]

\[ [((\lambda z . u(\lambda x . z))/y) M : \beta \quad \text{let } [((\lambda z . u(\lambda x . z))/y) M = M' \]

\[ \rightarrow \text{i} \]

\[ \lambda x . M' : \alpha \rightarrow \beta \]

\[ \forall \text{vi} \]

\[ \forall u . \lambda x . M' : \alpha \rightarrow \beta \]

**Example 7.2:** (Proof of W with Vi last)

(1) \[ x_1 : a \rightarrow a \rightarrow b \]

(2) \[ x_2 : a \]

\[ \rightarrow \text{e} \]

\[ x_1 x_2 : a \rightarrow b \]

\[ x_3 : (a \rightarrow b) \rightarrow b \]

\[ \rightarrow \text{e} \]

\[ x_3(x_1 x_2) : b \]

\[ \rightarrow \text{i} \]

\[ \lambda x_2 . x_3(x_1 x_2) : a \rightarrow b \]

\[ \forall \text{vi} \]

\[ \forall x_3 . \lambda x_2 . x_3(x_1 x_2) : a \rightarrow b \]

\[ \rightarrow \text{i} \]

\[ \lambda x_1 . \forall x_3 . \lambda x_2 . x_3(x_1 x_2) : (a \rightarrow a \rightarrow b) \rightarrow a \rightarrow b \]
becomes:

(4)
\[ x_3 : a \rightarrow a \rightarrow b \]

(3)
\[ u : a \land b \]

\[ \rightarrow_i \quad (\lambda x_3 . u : (a \rightarrow a \rightarrow b) \rightarrow a \rightarrow b) \]

(5)
\[ z : ((a \rightarrow a \rightarrow b) \rightarrow a \rightarrow b) \rightarrow b \]

\[ \rightarrow_e \]

\[ \lambda x_3 . u : (a \rightarrow a \rightarrow b) \rightarrow a \rightarrow b \]

\[ \rightarrow_i \quad (\lambda x_3 . u) : b \]

\[ \lambda u . z(\lambda x_3 . u) : (a \rightarrow b) \rightarrow b \]

\[ \rightarrow_e \]

\[ (\lambda u . z(\lambda x_3 . u))(x_1 x_2) : b \]

\[ \rightarrow_i \quad (\lambda x_2 . (\lambda u . z(\lambda x_3 . u))(x_1 x_2)) : a \rightarrow b \]

\[ \rightarrow_i \quad (\lambda x_1 x_2 . (\lambda u . z(\lambda x_3 . u))(x_1 x_2)) : (a \rightarrow a \rightarrow b) \rightarrow a \rightarrow b \]

\[ \rightarrow_e \]

\[ \lambda x_1 x_2 . (\lambda u . z(\lambda x_3 . u))(x_1 x_2) : (a \rightarrow a \rightarrow b) \rightarrow a \rightarrow b \]

Which reduces to

\[ V z . \lambda x_1 x_2 . z(\lambda x_3 . x_1 x_2) : (a \rightarrow a \rightarrow b) \rightarrow a \rightarrow b \]

which in turn translates to:

\[ V^* z . \lambda^* x_1 x_2 . z(K x_1 x_2) : (a \rightarrow a \rightarrow b) \rightarrow a \rightarrow b \]

\[ V^* z . \lambda^* x_1 . B z(B K x_1) : (a \rightarrow a \rightarrow b) \rightarrow a \rightarrow b \]

\[ V^* z . B^*(B K)(B z) : (a \rightarrow a \rightarrow b) \rightarrow a \rightarrow b \]

\[ P(B(B^*(B K)B)) : (a \rightarrow a \rightarrow b) \rightarrow a \rightarrow b \]

Now, we will introduce the algorithm for finding proofs in SKP- logic.
§ 7.3 Algorithms for SKP

The algorithm we introduce here will search for a long normal form proof of a type $\tau$ in which all $\forall$-steps have been postponed to just after the last set of $\lambda \rightarrow$ steps. It is clear from fig 7.1 and 7.2 that this postponement can always be performed.

Let $\tau = \tau_1 \rightarrow \tau_2 \rightarrow ... \rightarrow \tau_n \rightarrow a$, then the proof will take the following form:

\[
\begin{align*}
\vdots \\
x_1 : \tau_1, ..., x_n : \tau_n, x_{n+1} : \tau \rightarrow \beta_1, ..., x_{n+p} : \tau \rightarrow \beta_p \quad | X' : a \\
\vdash x_{n+1} : \tau \rightarrow \beta_1, ..., x_{n+p} : \tau \rightarrow \beta_p \quad | \lambda x_1 ... x_n . X' : \tau \\
\vdash \forall x_{n+1} ... x_{n+p} . \lambda x_1 ... x_n . X' : \tau
\end{align*}
\]

fig. 7.3

Now, we need to discover what forms $\beta_1, \beta_2, ...$ and $\beta_p$ can take, so we have the following theorem in SKP - logic.

**Theorem 7.1**

The deduction given in fig. 7.3 can be transformed to one in which each $\beta_i$ is an atom of $\tau$. 
Proof:
Consider the first appearance of an $x_{n+i}$ on the right of a |- in the proof.

(i) If this is in the form
$$x_1: \tau_1, \ldots, x_{n+p+1}: \tau \rightarrow \beta_i, x_{n+p+1}: \tau_{n+p+1}, \ldots, x_m: \tau_m \vdash x_{n+i} X_0 \ldots X_r: c$$
where $c$ is an atom and $\beta_i = \gamma_1 \rightarrow \ldots \rightarrow \gamma_r \rightarrow c$, then, there is a simpler proof of the form:
$$x_1: \tau_1, \ldots, x_{n+i-1}: \tau \rightarrow \beta_{n+i-1}, x_{n+i}: \tau \rightarrow c, x_{n+i+1}: \tau \rightarrow \beta_{n+i+1}, \ldots, x_m: \tau_m \vdash x_{n+i} X_0: c.$$ 
The rest of the proof will have $x_{n+i} X_0$ for $x_{n+i} X_0 \ldots X_r$ and this is still a SKP-theorem term.

If the atom $c$ is not in $\tau$, it can be replaced by an atom in $\tau$.

(ii) If this is in the form
$$x_1: \tau_1, \ldots, x_{n+p}: \tau \rightarrow \beta_p, x_{n+p+1}: \tau_{n+p+1}, \ldots, x_m: \tau_m \vdash x_j X_0 \ldots X_s x_{n+i} X_{s+2} \ldots X_m: d,$$
then $j$ is not $n+1 \leq j \leq n+p$.

(For, if $n+1 \leq j \leq n+p$ then $x_j$, after the changes in (i), has a type $\tau \rightarrow d$ and this is impossible as it require $s = 0$, $m = 1$ and $x_{n+i}: \tau$).

Hence, $j < n+1$ or $j > n+p$ and $\tau_j$ is a long negative subtype of $\tau$. $\tau \rightarrow \beta_i$ the type of $x_{n+i}$, must then be a long positive subtype of $\tau_j$ and so is a long positive proper subtype of $\tau$, which is impossible.

We then have:
Theorem 7.2

If \( c_1, \ldots, c_p \) are the type variables of \( \tau \), then to derive \( \vdash Y : \tau \) using \( \forall_i, \rightarrow_i \) and \( \rightarrow_e \) it is enough to derive

\[
x_{n+1} : \tau \rightarrow c_1, \ldots, x_{n+p} : \tau \rightarrow c_p \vdash X : \tau
\]

for some term \( X \) using only \( \rightarrow_i \) and \( \rightarrow_e \).

A value for \( Y \) is then \( \forall x_{n+1} \ldots x_{n+p} \cdot X \)

The first SKP algorithm:

Aim: Given \( \tau \), to find \( Y \) such that \( \vdash Y : \tau \).

Step 1 Identify the distinct type variables \( c_1, \ldots, c_p \) of \( \tau \).

Step 2 Find using the SK algorithm a term \( X \) such that

\[
x_{n+1} : \tau \rightarrow c_1, \ldots, x_{n+p} : \tau \rightarrow c_p \vdash X : \tau
\]

Then \( Y \equiv \forall x_{n+1} \ldots x_{n+p} \cdot X \).

Towards an alternative algorithm

We can introduce a new variable \( y_i \) as an inhabitant of each use of \( \tau \rightarrow c_i \) in the proof of \( \tau \) where \( c_i \)'s are type variables of \( \tau \). This gives an SK proof of

\[
y_1 : (\tau \rightarrow c_1), \ldots, y_p : (\tau \rightarrow c_p) \vdash X : \tau
\]

A given \( \tau \rightarrow c_i \) may be used zero or more times and in general we have

\[
\vdash \lambda y_1 \ldots y_p \cdot X : (\tau \rightarrow c_1) \rightarrow \ldots (\tau \rightarrow c_p) \rightarrow \tau
\]

where \( \lambda y_1 \ldots y_p \cdot X \) is SK definable. We therefore have:
The second SKP algorithm

Aim  Given \( \tau \), to find \( X \) such that \( \vdash X : \tau \).

**Step 1** Identify the distinct type variables \( c_1, \ldots, c_p \) of \( \tau \).

**Step 2** Find \( Y \) such that \( \vdash Y : (\tau \rightarrow c_1) \rightarrow \ldots (\tau \rightarrow c_p) \rightarrow \tau \) by the SK - algorithm.

Then \( X = \forall y_1 \ldots y_p . Y y_1 \ldots y_p \) represents a proof of \( \tau \) and

\[ (\forall y_1 \ldots y_p . Y y_1 \ldots y_p) = (\forall y_1 \ldots y_p . Y^* y_1 \ldots y_p) \]

will represent the Hilbert style proof.

Now, we will find the number of (B) nodes in the SKP- algorithm which we have to consider to give a decision procedure for the SKP- logic and also to give a proof of any provable classical implicational formula.

**Theorem 7.3** (In SKP- logic)

(i) \( \vdash X : \tau \) has a solution for \( X \) iff the TI - tree with \( \vdash X : \tau \) at the root has a TMC subtree with no TR - branches.

(ii) If \( \vdash X : \tau \) has a solution for \( X \) in SKP, then there is a TMC subtree of TI - tree in which each branch has at most

\[ ((|\tau| + 1)(|\tau| + 1) - 2)|\tau| \]

of (B) nodes.

**Proof :**

(i) The same as in Theorem 4.1(i)

(ii) If \( c_1, \ldots, c_n \) are variables of \( \tau \), then

\[ \vdash (\tau \rightarrow c_1) \rightarrow \ldots (\tau \rightarrow c_n) \rightarrow \tau \]

\[ = (|\tau| + 1)(|\tau| + 1) - 1. \]

and \( |\tau| = |\tau| \)
Chapter 7 Classical Implicational Logic

So substituting into Theorem 4.1 (ii) we have that the upper bound on the number of (B) nodes is:

$$((l \tau l + 1) (l \tau l + 1) - 2 ) l \tau l$$

**Corollary:**

If in the TI - tree with l- X : τ as root, there is no solution for X in SKP in any subtree without type repetition, then there is no solution at all. The maximal number of (B) nodes in any branch that needs to be checked is $$((l \tau l + 1) (l \tau l + 1) - 2 ) l \tau l$$.

The above theorems also hold if τ contains the constant f which introduced in the definitions of the various connectives in Chapter 5. At this stage we are not assuming the f_e rule or axiom.

Below are two examples using the Peirce rule.

**Example 7.3**

To find a proof of a \lor a \rightarrow a , where a \lor b = (a \rightarrow f) \rightarrow b, we will add extra hypotheses x_1 and x_2 with types

$$(((a \rightarrow f) \rightarrow a) \rightarrow a) \rightarrow f$$ and $$(((a \rightarrow f) \rightarrow a) \rightarrow a) \rightarrow a$$

respectively at the root of the TI - tree and use the Ben - Yelles algorithm, as in Chapter 4, in figure 7.4 below where

$$\Gamma_{31} = \Gamma_3 , x_7 : (a \rightarrow f) \rightarrow a$$

$$\Gamma_{32} = \Gamma_{31} , x_8 : a$$

$$\Gamma_{33} = \Gamma_{31} , x_9 : (a \rightarrow f) \rightarrow a$$.
\[ X_3 = x_4 \]

\[ X_2 = x_4 \]

\[ \Gamma_3, x_7 : (a \rightarrow f) \rightarrow (a \rightarrow f) \rightarrow a \rightarrow X_4' : a \]

\[ \Gamma_3, x_4 : a \rightarrow X_1' : f \]

\[ \Gamma_3 \models X_4 : ((a \rightarrow f) \rightarrow a) \rightarrow a \rightarrow X_1 : a \rightarrow f \rightarrow X_2 X_4 : a \rightarrow X_3 X_1 : a \rightarrow \]

\[ x_1 : (((a \rightarrow f) \rightarrow a) \rightarrow a) \rightarrow f, x_2 : (((a \rightarrow f) \rightarrow a) \rightarrow a) \rightarrow a \rightarrow a, \]

\[ x_3 : (a \rightarrow f) \rightarrow a \rightarrow X' : a \]

\[ x_1 : (((a \rightarrow f) \rightarrow a) \rightarrow a) \rightarrow f, x_2 : (((a \rightarrow f) \rightarrow a) \rightarrow a) \rightarrow a \rightarrow a \rightarrow X : ((a \rightarrow f) \rightarrow a) \rightarrow a. \]

fig. 7.4
From figure 7.4, we can see that there is no proof of \( a \lor a \rightarrow a \) in SK-logic but by using the \( x_1 \) and \( x_2 \) hypotheses we can find proofs such as \( \forall x_1 x_2 . \lambda x_3 . x_3(\lambda x_4 . x_1(\lambda x_5 . x_4)) : ((a \rightarrow f) \rightarrow a) \rightarrow a \) by the first SKP algorithm and \( \forall x_1 x_2 . (\lambda x_1 x_2 . \lambda x_3 . x_3(\lambda x_4 . x_1(\lambda x_5 . x_4))) x_1 x_2 \) by the second algorithm. (The second proof \( \beta \) - reduces to the first).

This translates to

\[
\forall x_1 x_2 . \lambda ^* x_3 . x_3(\lambda ^* x_4 . x_1(\lambda ^* x_5 . x_4))
\]

\[
\equiv \forall x_1 x_2 . \lambda ^* x_3 . x_3(\lambda ^* x_4 . x_1(Kx_4))
\]

\[
\equiv \forall x_1 x_2 . \lambda ^* x_3 . x_3(Bx_1(\lambda ^* x_4 . Kx_4))
\]

\[
\equiv \forall x_1 x_2 . \lambda ^* x_3 . x_3(Bx_1(K))
\]

\[
\equiv \forall x_1 x_2 . C(\lambda ^* x_3 . x_3)(Bx_1K)
\]

\[
\equiv \forall x_1 x_2 . CI(Bx_1K)
\]

\[
\equiv \forall x_1 . P(\lambda ^* x_2 . CI(Bx_1K))
\]

\[
\equiv \forall x_1 . P(K(CI(Bx_1K)))
\]

\[
\equiv P(\lambda ^* x_1 . P(K(CI(Bx_1K))))
\]

\[
\equiv P(BP(\lambda ^* x_1 . K(CI(Bx_1K))))
\]

\[
\equiv P(BP(BK(\lambda ^* x_1 . CI(Bx_1K))))
\]

\[
\equiv P(BP(BK(B(CI)(\lambda ^* x_1 . Bx_1K))))
\]

\[
\equiv P(BP(BK(B(CI)(C(\lambda ^* x_1 . Bx_1)K))))
\]

\[
\equiv P(BP(BK(B(CI)(CBK))))
\]

which will represent the Hilbert style proof.
Example 7.4

To find proofs of \( a \lor b \rightarrow b \lor a \) if \( a \lor b = (a \rightarrow b) \rightarrow b \)
we add two variables \( x_1 \) and \( x_2 \) with types \(((a \rightarrow b) \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a\) and \(((a \rightarrow b) \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a \rightarrow b\) respectively at the root, as in figure 7.5 below where

\[
\begin{align*}
\Gamma_{41} &= \Gamma_4, x_{12}: (a \rightarrow b) \rightarrow b, x_{13}: b \rightarrow a. \\
\Gamma_{42} &= \Gamma_4, x_{10}: (a \rightarrow b) \rightarrow b, x_{11}: b \rightarrow a.
\end{align*}
\]

From figure 7.5, we can see that there is no proof of \( a \lor b \rightarrow b \lor a \) in \( \text{SK} \)-logic, but by using the extra hypotheses \( x_1 \) and \( x_2 \), we can find proofs such as

\[
\begin{align*}
\forall x_1 x_2 \cdot \lambda x_3 x_4. x_4(x_3(\lambda x_5. x_2(\lambda x_6 x_7. x_5))) &: (a \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a \quad \text{by the first SKP algorithm} \\
\forall x_1 x_2. (\lambda x_1 x_2 \cdot \lambda x_3 x_4. x_4(x_3(\lambda x_5. x_2(\lambda x_6 x_7. x_5)))) x_1 x_2 &: (a \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a \quad \text{by the second SKP algorithm.}
\end{align*}
\]

This translates to \( \forall^* x_1 x_2. \lambda^* x_3 x_4. x_4(x_3(\lambda^* x_5. x_2(\lambda^* x_6 x_7. x_5))) \)
which will represent the Hilbert style proof.
Chapter 7

Classical Implicational Logic

\[
\begin{align*}
X_4 &= X_5 \\
\Gamma_7, x_8: (a \rightarrow b) &\rightarrow b, x_9: b \rightarrow a \vdash X_4: a \\
\Gamma_7 &\vdash X_4: ((a \rightarrow b) \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a \\
\Gamma_7 &\vdash x_1 X_4 : a
\end{align*}
\]

\[
\begin{align*}
X_3 &= X_5 \\
\Gamma_5, x_6: (a \rightarrow b) &\rightarrow b, x_7: b \rightarrow a \vdash X_3: a \\
\Gamma_5 &\vdash X_3: ((a \rightarrow b) \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a \\
\Gamma_5 &\vdash x_2 X_3 : b
\end{align*}
\]

\[
\begin{align*}
\Gamma_4 &\vdash x_4 X_5 : a \\
\Gamma_4 &\vdash X_5: ((a \rightarrow b) \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a \\
\Gamma_4 &\vdash X_2 : b
\end{align*}
\]

\[
\begin{align*}
\Gamma_4 &\vdash x_7 X_4 : a \\
\Gamma_4 &\vdash X_4: ((a \rightarrow b) \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a \\
\Gamma_4 &\vdash x_5 : a \\
\Gamma_4 &\vdash X_2 : b
\end{align*}
\]

\[
\begin{align*}
\Gamma_4 &\vdash x_6: ((a \rightarrow b) \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a \\
\Gamma_4 &\vdash x_1 X_6 : a \\
\Gamma_4 &\vdash X_6 : ((a \rightarrow b) \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a \\
\Gamma_4 &\vdash x_4 X_1 : a
\end{align*}
\]

\[
\begin{align*}
x_1 : (((a \rightarrow b) \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a) &\rightarrow a, \\
x_2 : (((a \rightarrow b) \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a) &\rightarrow b, x_3 : (a \rightarrow b) \rightarrow b, \\
x_4 : b \rightarrow a &\vdash X_4 : a
\end{align*}
\]

\[
\begin{align*}
x_1 : (((a \rightarrow b) \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a) &\rightarrow a, x_2 : (((a \rightarrow b) \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a) \rightarrow b \\
\Gamma_4 &\rightarrow (b \rightarrow a) \rightarrow a \vdash X : ((a \rightarrow b) \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a
\end{align*}
\]

fig. 7.5
Now, by using "Brouwer 6.0" (see Dekker [8]) we will give further examples which form Theorems 7.5 and 7.6 below. In some cases fewer than the maximum number of $x_i: \tau \rightarrow c_i$ assumptions were used to save computer time.

**Weaker logics with P (or $\nu_i$) but without the $fe$ axiom:**

We add the following to the translations in § 7.1

\[(p_2) \quad (x_1, x_2, ..., x_n; P)^* = P\]

\[(p_3) \quad (x_1, x_2, ..., x_n; \forall x_{n+1}. Y)^* = (x_1, x_2, ..., x_n; P(\lambda x_{n+1}. Y))^*.\]

**Example:** If $*$ is $(i \eta b b')$ then

\[
(\lambda x_1. \forall x_3. \lambda x_2. x_3 (x_1 x_2))^* \\
\equiv \lambda^* x_1. (x_1; \forall x_3. \lambda x_2. x_3 (x_1 x_2))^* \\
\equiv \lambda^* x_1. (x_1; P(\lambda x_3. \lambda x_2. x_3 (x_1 x_2))^*) \\
\equiv \lambda^* x_1. (x_1; P)^* (x_1; \lambda x_3. \lambda x_2. x_3 (x_1 x_2))^* \\
\equiv \lambda^* x_1. P(\lambda x_3^1. (\lambda x_2^3. (x_1, x_3, x_2; x_3 (x_1 x_2))^*)) \\
\equiv \lambda^* x_1. P(\lambda x_3^1. (\lambda x_2^3. (x_1, x_3, x_2; x_3 (x_1 x_2))^*))^* \\
\equiv \lambda^* x_1. P(\lambda x_3^1. B'(\lambda x_2^3. x_1 x_2) x_3 ) \\
\equiv \lambda^* x_1. P(\lambda x_3^1. B' x_1 x_3 ) \\
\equiv \lambda^* x_1. P(\lambda x_3^1. B' x_1 ) \\
\equiv B P(\lambda^* x_1. B' x_1 ) \\
\equiv B P B'.
\]

This example shows the first part of the following theorem.
Theorem 7.4

(i) A combinator with the type of $W$ can be defined in any logic with $B$, $B'$ and $P$.

(ii) The type of $K$ can be defined from those of $B$, $P$, $C$ and $I$.

(iii) The type of $C$ can be definable from those of $B$, $W$, $B'$ and $K$.

Proof:

(i) Using $\to_i$, $\to_e$ and $\forall_i$ we can deduce that

$$BPB' : (a \to (a \to b)) \to a \to b$$

which is the same as the type of $W$.

(ii) $B(BP)(BC(CI)) : a \to b \to a$

(iii) $C = B(BW)(B(B'(BB'K))B')$.

$BPB'$ of course does not have the property of the combinator $W$ which is $Wxy \rhd xyy$. Similarly, the combinator in the proof of (ii) does not have the reduction property of $K$.

It follows that the only logics of interest are $BB'P$ and $BB'IP$. $BCIP$ and $BCKP$ are the same as SKP.

Note 1: In the proof reduction shown in fig.7.2 the hypotheses introduced in $D_2$ can only be cancelled in $D_2$ but hypotheses introduced in $D_1$ may be cancelled in $D_1$ or $D_2$. So in the normalized deduction fig. 7.2 they may be cancelled without first being introduced.
A \textit{Vi} step postponed past a \( \rightarrow_{\text{BCI}} \) step turns this into a \( \rightarrow_{\text{BCK}} \) step and hence, this proof reduction works, in general, only for logics that contain \textit{K}.

\textbf{Note 2} : Also, a \textit{Vi} step postponed after a \( \rightarrow_{i} \) step turns this into a \( \rightarrow_{iw} \) step and hence, this proof reduction works, in general, only for logics that contain \textit{W}.

Hence for the two weaker systems \textit{BB} \^\textit{P} and \textit{BB} \^\textit{IP}, our earlier methods do not work because we need both \textit{K} and \textit{W} to postpone \textit{Vi}.

\textbf{Theorem 7.5}

The following are theorems of \textit{SKP} - logic :-

(1) \( a \lor a \rightarrow a \)
(2) \((a \rightarrow b) \rightarrow a \) \( \rightarrow a \)
(3) \( a \lor_1 b \rightarrow b \lor_1 a \)
(4) \((a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow a \lor_1 b \rightarrow c \)
(5) \( a \land (b \lor_1 c) \rightarrow (a \land b) \lor_1 c \)
(6) \( a \land (b \lor_1 c) \rightarrow (a \land b) \lor_1 c \)

\textbf{Proof}:

In each case, this follows from Definition 5.1, the given \( \lambda \)-\textit{V} - proof using the second \textit{SKP} algorithm and the translations of Chapter 3 :

(1) \textit{V}x_1x_2.(\lambda x_1 x_2 x_3 . x_3(\lambda x_4 . x_2(\lambda x_5 . x_1(\lambda x_6 . x_4))))x_1x_2 :

\((a \rightarrow f) \rightarrow a \) \( \rightarrow a . \)
(2) \( \forall x_1 x_2. (\lambda x_1 x_2 x_3 . x_1(\lambda x_4 . x_3(\lambda x_5 . x_2(\lambda x_6 . x_5))))x_1 x_2 : \\
\quad ((a \to b) \to a) \to a
\)

(3) \( \forall x_1 x_2. (\lambda x_1 x_2 x_3 x_4 . x_4(x_3(\lambda x_5 . x_2(\lambda x_6 x_7 . x_1(\lambda x_8 x_9 . x_5))))))x_1 x_2 : \\
\quad ((a \to b) \to b) \to (b \to a) \to a
\)

(4) \( \forall x_1 x_2 x_3. (\lambda x_1 x_2 x_3 x_4 x_5 x_6 . x_4(x_1(\lambda x_7 x_8 x_9 . x_5(x_6(\lambda x_{10} . x_9(\lambda x_{11} . x_2 \\
\quad (\lambda x_{12} x_{13} x_{14} . x_7 x_{10})))))x_1 x_2 x_3 : \\
\quad (a \to c) \to (b \to c) \to (((a \to b) \to b) \to c)
\)

(5) \( \forall x_1 x_2 x_3 x_4 . (\lambda x_1 x_2 x_3 x_4 x_5 x_6 . x_6(\lambda x_7 . x_5(\lambda x_8 x_9 . x_4(\lambda x_{10} x_{11} . x_9(\lambda x_{12} \\
\quad . x_{11} (\lambda x_{13} . x_{10}(\lambda x_{14} x_{15} . x_{13}(x_1(\lambda x_{16} x_{17} . x_{15}(\lambda x_{18} . x_{17} (\lambda x_{19} . x_{16} \\
\quad (\lambda x_{20} x_{21} . x_7 x_{14} x_{18})))) x_{12}))))))x_1 x_2 x_3 x_4 : \\
\quad ((a \to ((b \to c) \to c) \to f) \to f) \to (((a \to b \to f) \to f) \to c) \to c.
\)

(6) \( \forall x_1 x_2 x_3 x_4 . (\lambda x_1 x_2 x_3 x_4 x_5 x_6 . x_6(\lambda x_7 . x_5(\lambda x_8 x_9 . x_4(\lambda x_{10} x_{11} . x_8 x_9 \\
\quad (\lambda x_{12} . x_{11} (\lambda x_{13} . x_{10}(\lambda x_{14} x_{15} . x_7(\lambda x_{16} . x_{12})(x_1(\lambda x_{17} x_{18} . x_{14} x_{15} (\lambda x_{19} . \\
\quad x_{18} (\lambda x_{20} . x_{17} (\lambda x_{21} x_{22} . x_{20}(\lambda x_{23} . x_{19} x_{22}))))))))))x_1 x_2 x_3 x_4 : \\
\quad (((a \to (b \to c) \to c) \to a \to f) \to f) \to (((a \to b) \to a \to f) \\
\quad \to f) \to c) \to c.
Theorem 7.6

Parts (1), (2), (3) and (4) of Theorem 7.5 are also BB'P and BB'IP theorems.

Proof:

The following terms and types demonstrate the proofs using $V_i$ (Peirce's law) :-

(1) $\lambda x_1.V x_2. x_1 x_2 : ((a \rightarrow f) \rightarrow a) \rightarrow a$ (or P)

(3) $\lambda x_1 x_2. V x_3. x_2(x_1 x_3) : ((a \rightarrow b) \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a$

(4) $\lambda x_1 x_2 x_3. V x_4. x_2(x_3(\lambda x_5. x_4(x_1 x_5))) :$

\hspace{1cm} (a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow ((a \rightarrow b) \rightarrow b) \rightarrow c$

Theorem 7.7

The following are not types of any term and so not theorems of SKP or any of its subsystems :-

(1) $\vdash \neg \neg a \rightarrow a$

(2) $\vdash (\neg a \rightarrow \neg b) \rightarrow (b \rightarrow a)$

(3) $\vdash a \& b \rightarrow a$

(4) $\vdash a \& b \rightarrow b$

(5) $\vdash a \& b \rightarrow a$
(6) \( \vdash a \land b \rightarrow b \)

(7) \( \vdash a \rightarrow a \lor b \)

(8) \( \vdash a \lor b \rightarrow b \lor a \)

(9) \( \vdash (a \rightarrow c) \land (b \rightarrow c) \rightarrow (a \lor b) \rightarrow c \)

(10) \( \vdash (a \rightarrow c) \land (b \rightarrow c) \rightarrow (a \lor b) \rightarrow c \)

(11) \( \vdash (a \rightarrow c) \land (b \rightarrow c) \rightarrow (a \lor_1 b) \rightarrow c \)

(12) \( \vdash (a \rightarrow c) \land (b \rightarrow c) \rightarrow (a \lor_1 b) \rightarrow c \)

(13) \( \vdash a \land (b \lor c) \rightarrow (a \land b) \lor c \)

(14) \( \vdash a \land (b \lor c) \rightarrow (a \land b) \lor c \).
In this chapter, we extend the Ben-Yelles Algorithm to some logics with connectives other than $\rightarrow$, using a collection of properties from Chapters 5, 6 and 7. These logics will have:

$$fe : f \rightarrow \alpha \quad \text{and} \quad P : ((\alpha \rightarrow \beta) \rightarrow \alpha) \rightarrow \alpha \quad \text{(or} \ V_i\text{)}$$

and the definitions of $\sim$, $\lor$, $\land$ of Definition 5.1.

In a proof in classical logic of a theorem $\tau$, written in terms of $\rightarrow$ and $f$ only, we assume that all $V_i$ steps are postponed to the end of the proof. So we have an $SKf_e$ proof followed by $V_i$ steps.

We use $y_i$ as an inhabitant of each use of $f \rightarrow c_i$ in the $SKf_e$ part of the proof where the $c_i$'s are the type variables of $\tau$. This gives an $SK$ proof of

$$x_1 : \tau \rightarrow c_1, \ldots, x_k : \tau \rightarrow c_k, y_1 : f \rightarrow c_1, \ldots, y_k : f \rightarrow c_k \quad \vdash X : \tau$$

So we have

$$y_1 : f \rightarrow c_1, \ldots, y_k : f \rightarrow c_k \quad \vdash \forall x_1 \ldots x_k . X : \tau$$

then

$$\vdash \lambda y_1 \ldots y_k . \forall x_1 \ldots x_k . X : (f \rightarrow c_1) \rightarrow \ldots (f \rightarrow c_k) \rightarrow \tau$$

The classical proof is then given by

$$\vdash [fe/y_1, \ldots, fe/y_k] \forall x_1 \ldots x_k . X : \tau$$

or

$$\vdash \forall x_1 \ldots x_k . [fe/y_1, \ldots, fe/y_k] X : \tau.$$
Alternatively, if

\[
\begin{align*}
 l- Y : (\tau \to c_1) \to \ldots \to (\tau \to c_k) \to (f \to c_1) \to \ldots \to (f \to c_k) \to \tau
\end{align*}
\]

then \( x_1 : \tau \to c_1, \ldots, x_k : \tau \to c_k \mid l- Y x_1 \ldots x_k f_e \ldots f_e : \tau \)

and so \( l- \forall x_1 \ldots x_k. Y x_1 \ldots x_k f_e \ldots f_e : \tau \)

We therefore have:

The SKPfe algorithm

**Aim** To find \( X \) such that \( l- X : \tau \).

**Step 1** Identify the distinct type variables \( c_1 , \ldots, c_k \) of \( \tau \), where \( c_k \) is \( f \).

**Step 2** Find \( Y \) using the SK - algorithm such that

\[
\begin{align*}
 l- Y : (\tau \to c_1) \to \ldots (\tau \to c_k) \to (f \to c_1) \to \ldots (f \to c_{k-1}) \to \tau
\end{align*}
\]

**Step 3** Then in SKPfe:

\[
\begin{align*}
 l- \forall x_1 \ldots x_k. Y x_1 \ldots x_k f_e \ldots f_e : \tau.
\end{align*}
\]

Now, we extend Theorem 4.1 to find the maximum number of (B) nodes that need to be considered in the above algorithm to find proofs in SKPfe.
Theorem 8.1 In SKPfe-logic:

(i) If \( \vdash X: \tau \) has a solution for \( X \) iff the TI-tree with

\[
\vdash Y: (\tau \rightarrow c_1) \rightarrow ... (\tau \rightarrow c_n) \rightarrow (f \rightarrow c_1) \rightarrow ... (f \rightarrow c_n) \rightarrow \tau
\]

at the root has a TMC subtree with no TR-branches.

(ii) If \( \vdash X: \tau \) has a solution for \( X \), then there is a TMC subtree of the TI-tree in (i) in which each branch has at most

\[
((\| \tau \| + 1)(\| \tau \| + 3) - 4) \| \tau \| \text{ (B) nodes.}
\]

Proof:

(i) As for Theorem 4.1(i).

(ii) If \( c_1, ..., c_n \) are the variables of \( \tau \), then

\[
\vdash (\tau \rightarrow c_1) \rightarrow ... (\tau \rightarrow c_n) \rightarrow (f \rightarrow c_1) \rightarrow ... (f \rightarrow c_n) \rightarrow \tau
\]

\[
= 3 \| \tau \| + (\| \tau \| + 1) \| \tau \|
\]

\[
= (\| \tau \| + 1)(\| \tau \| + 3) - 3
\]

and, assuming \( f \) is in \( \tau \),

\[
\| (\tau \rightarrow c_1) \rightarrow ... (\tau \rightarrow c_n) \rightarrow (f \rightarrow c_1) \rightarrow ... (f \rightarrow c_n) \rightarrow \tau \| = \| \tau \|
\]

So substituting into Theorem 4.1 (ii) we have that the upper bound on the number of (B) nodes is:

\[
((\| \tau \| + 1)(\| \tau \| + 3) - 4) \| \tau \|
\]

Corollary:

If in the TI-tree with \( \vdash X: \tau \) as root, there is no solution for \( X \) in SKPfe in any subtree without type repetition, then there is no solution at all. The maximal number of (B) nodes in any branch that needs to be checked is

\[
((\| \tau \| + 1)(\| \tau \| + 3) - 4) \| \tau \|.
\]
Example 8.1

To find a proof of \( a \lor b \rightarrow b \lor a \) using \( f_e \) and \( P \)

where \( a \lor b = (a \rightarrow f) \rightarrow b \)

We will add five extra hypotheses \( \Gamma_5 = \{ x_1 : (((a \rightarrow f) \rightarrow b) \rightarrow (b \rightarrow f) \rightarrow a) \rightarrow a, x_2 : (((a \rightarrow f) \rightarrow b) \rightarrow (b \rightarrow f) \rightarrow a) \rightarrow b, x_3 : (((a \rightarrow f) \rightarrow b) \rightarrow (b \rightarrow f) \rightarrow a) \rightarrow f, x_4 : f \rightarrow a \) and \( x_5 : f \rightarrow b \)

at the root of a TI-tree for \( ((a \rightarrow f) \rightarrow b) \rightarrow (b \rightarrow f) \rightarrow a \) and proceed with the SK-algorithm as in Chapter 4 as in figure 8.1.

Note that some branches are omitted.
\[ X_8 = x_{10} \]

\[ \Gamma_{10}, x_{11} : (a \rightarrow f) \rightarrow b, x_{12} : b \rightarrow f \vdash X_8 : a \]

\[ \Gamma_{10} \vdash X_8 : ((a \rightarrow f) \rightarrow b) \rightarrow (b \rightarrow f) \rightarrow a \]

\[ \Gamma_{10} \vdash x_2 X_8 : b \]

\[ \Gamma_{10} \vdash X_7 : b \]

\[ \Gamma_{10} \vdash x_7 X_7 : f \]

\[ \Gamma_9, x_{10} : a \vdash X_6 : f \]

\[ \Gamma_9 \vdash X_6 : a \rightarrow f \]

\[ \Gamma_9 \vdash x_6 X_6 : b \]

\[ \Gamma_9 \vdash X_3 : b \]

\[ \Gamma_9 \vdash x_9 X_3 : f \quad \Gamma_9 \vdash x_3 X_4 : f \quad \Gamma_9 \vdash x_7 X_5 : f \quad \vdash \quad \vdash \quad \vdash \quad \vdash \]

\[ \Gamma_9 \vdash X_2 : f \quad \Gamma_7 \vdash X_{10} : ((a \rightarrow f) \rightarrow b) \rightarrow (b \rightarrow f) \rightarrow a \quad \Gamma_7 \vdash X_{11} : b \]

\[ \Gamma_9 \vdash x_4 X_2 : a \quad \Gamma_7 \vdash x_3 X_{10} : f \quad \Gamma_7 \vdash x_7 X_{11} : f \quad (C) \]

\[ \Gamma_7, x_8 : (a \rightarrow f) \rightarrow b, x_9 : b \rightarrow f \vdash X_1 : a \quad \Gamma_7 \vdash X_9 : f \quad (B) \]

\[ \Gamma_7 \vdash X_1 : ((a \rightarrow f) \rightarrow b) \rightarrow (b \rightarrow f) \rightarrow a \quad \Gamma_7 \vdash X_9 : f \quad (A) \]

\[ \Gamma_7 \vdash x_1 X_1 : a \quad \Gamma_7 \vdash x_4 X_9 : a \quad (C) \]

\[ \Gamma_5, x_6 : (a \rightarrow f) \rightarrow b, x_7 : b \rightarrow f \vdash X' : a \quad (B) \]

\[ \Gamma_5 \vdash X : ((a \rightarrow f) \rightarrow b) \rightarrow (b \rightarrow f) \rightarrow a \quad (A) \]

\[ \text{fig 8.1} \]
From figure 8.1, we can find a proof such as

\[ \forall x_1 x_2 x_3 . (\lambda x_1 x_2 x_3 x_4 x_5 x_6 x_7 . x_1 (\lambda x_8 x_9 . x_4 (x_9 (x_6 (\lambda x_{10} . x_7 (x_2 (\lambda x_{11} x_12 . x_{10}))))))) x_1 x_2 x_3 \text{efe : } ((a \rightarrow f) \rightarrow b) \rightarrow (b \rightarrow f) \rightarrow a \text{ in SKPfe.} \]

Of course all theorems of classical logic are theorems of SKPfe. Some more examples using "Brouwer 6.0" (see Dekker [8]) appear in Theorem 8.2 below.

**Note that** we need both K and W for postponement, so the earlier methods do not work for the weaker systems BB'Pfe and BB'IPfe.

**Theorem 8.2**

The following are theorems in the SKPfe - logic :- (Consider Definition 5.1)

1. \( \neg \neg a \rightarrow a \)
2. \( (\neg a \rightarrow \neg b) \rightarrow (b \rightarrow a) \)
3. \( a \& b \rightarrow a \)
4. \( a \& b \rightarrow b \)
5. \( a \land b \rightarrow a \)
6. \( a \land b \rightarrow b \)
7. \( (a \rightarrow c) \& (b \rightarrow c) \rightarrow (a \lor b) \rightarrow c \)
8. \( (a \rightarrow c) \land (b \rightarrow c) \rightarrow (a \lor b) \rightarrow c \)
9. \( (a \rightarrow c) \& (b \rightarrow c) \rightarrow (a \lor_1 b) \rightarrow c \)
10. \( (a \rightarrow c) \land (b \rightarrow c) \rightarrow (a \lor_1 b) \rightarrow c \)
(11) \( a \land (b \lor c) \rightarrow (a \land b) \lor c \)

(12) \( a \land (b \lor c) \rightarrow (a \land b) \lor c \)

**Proof:**

In each case, this follows from the given \( \lambda \lor \rightarrow \) proofs and the translations of Chapter 3:

(1) \( \forall x_1 x_2 . (\lambda x_1 x_2 x_3 x_4 . x_3(x_4(\lambda x_5 . x_2(\lambda x_6 . x_5))))x_1 x_2 f e : ((a \rightarrow f) \rightarrow f) \rightarrow a \)

(2) \( \forall x_1 x_2 x_3 . (\lambda x_1 x_2 x_3 x_4 x_5 x_6 x_7 . x_1(\lambda x_8 x_9 . x_4(x_6(\lambda x_{10} . x_8(\lambda x_{11} . x_3(\lambda x_{12} x_{13} . x_{11}))(x_2(\lambda x_{14} x_{15} . x_{10}))))x_7)))x_1 x_2 x_3 f e e : ((a \rightarrow f) \rightarrow b \rightarrow f) \rightarrow b \rightarrow a \)

(3) \( \forall x_1 x_2 x_3 . (\lambda x_1 x_2 x_3 x_4 x_5 x_6 . x_1(\lambda x_7 . x_4(x_6(\lambda x_8 x_9 . x_7(\lambda x_{10} x_{11} . x_3(\lambda x_{12} . x_{10}))))))x_1 x_2 x_3 f e e : ((a \rightarrow b \rightarrow f) \rightarrow f) \rightarrow a \)

(4) \( \forall x_1 x_2 x_3 . (\lambda x_1 x_2 x_3 x_4 x_5 x_6 . x_2(\lambda x_7 . x_5(x_7(\lambda x_8 x_9 . x_6(\lambda x_{10} x_{11} . x_3(\lambda x_{12} . x_9))))))x_1 x_2 x_3 f e e : ((a \rightarrow b \rightarrow f) \rightarrow f) \rightarrow b \)

(5) \( \forall x_1 x_2 x_3 . (\lambda x_1 x_2 x_3 x_4 x_5 x_6 . x_1(\lambda x_7 . x_4(x_7(\lambda x_8 x_9 . x_6(\lambda x_{10} x_{11} . x_3(\lambda x_{12} . x_{11}))))))x_1 x_2 x_3 f e e : ((a \rightarrow b) \rightarrow a \rightarrow f) \rightarrow f) \rightarrow a \)
Chapter 8  Classical Logic

(6) \( \forall x_1x_2x_3 . (\lambda x_1x_2x_3x_4x_5x_6 . x_2(\lambda x_7 . x_4(x_6(\lambda x_8x_9 . x_7(\lambda x_{10}x_{11} . x_3(\lambda x_{12} x_{10}(x_1(\lambda x_{13} . x_8x_9))))))))x_1x_2x_3fefe : ( ((a \rightarrow b) \rightarrow a \rightarrow f) \rightarrow f \rightarrow b )

(7) \( \forall x_1x_2x_3x_4 . (\lambda x_1x_2x_3x_4x_5x_6x_7x_8x_9 . x_7(x_8(\lambda x_{10}x_{11} . x_4(\lambda x_{12}x_{13} . x_{11} x_9(\lambda x_{14} . x_4(\lambda x_{15}x_{16} . x_{10}x_{14}))))))))x_1x_2x_3x_4fefe : 
\(( ((a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow f) \rightarrow f \rightarrow ((a \rightarrow f) \rightarrow b) \rightarrow c)\)

(8) \( \forall x_1x_2x_3x_4 . (\lambda x_1x_2x_3x_4x_5x_6x_7x_8x_9 . x_7(x_8(\lambda x_{10}x_{11} . x_4(\lambda x_{12}x_{13} . x_{10}x_{11} x_9(\lambda x_{14} . x_4(\lambda x_{15}x_{16} . x_{11}x_{14}))))))))x_1x_2x_3x_4fefe : 
\(( ((a \rightarrow c) \rightarrow b \rightarrow c) \rightarrow (a \rightarrow c) \rightarrow f) \rightarrow f \rightarrow ((a \rightarrow f) \rightarrow b) \rightarrow c)\)

(9) \( \forall x_1x_2x_3x_4 . (\lambda x_1x_2x_3x_4x_5x_6x_7x_8x_9 . x_7(x_8(\lambda x_{10}x_{11} . x_4(\lambda x_{12}x_{13} . x_{11} x_9(\lambda x_{14} . x_9(\lambda x_{15} . x_2(\lambda x_{16}x_{17} . x_{10}x_{14}))))))))x_1x_2x_3x_4fefe : 
\(( ((a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow f) \rightarrow f \rightarrow ((a \rightarrow b) \rightarrow b) \rightarrow c)\)

(10) \( \forall x_1x_2x_3x_4 . (\lambda x_1x_2x_3x_4x_5x_6x_7x_8x_9 . x_7(x_8(\lambda x_{10}x_{11} . x_4(\lambda x_{12}x_{13} . x_{10}x_{11} x_9(\lambda x_{14} . x_6(x_4(\lambda x_{15}x_{16} . x_{11}x_{14}))))))))x_1x_2x_3x_4fefe : 
\(( ((a \rightarrow c) \rightarrow b \rightarrow c) \rightarrow (a \rightarrow c) \rightarrow f) \rightarrow f \rightarrow ((a \rightarrow b) \rightarrow b) \rightarrow c)\)

(11) \( \forall x_1x_2x_3x_4 . (\lambda x_1x_2x_3x_4x_5x_6x_7x_8x_9 . x_7(x_8(\lambda x_{10}x_{11} . x_4(\lambda x_{12}x_{13} . x_{11} x_9(\lambda x_{14} . x_9(\lambda x_{15} . x_{15}x_{10}x_{14}))))))))x_1x_2x_3x_4fefe : 
\(( (a \rightarrow ((b \rightarrow f) \rightarrow c) \rightarrow f) \rightarrow f \rightarrow (((a \rightarrow b) \rightarrow f) \rightarrow f) \rightarrow f) \rightarrow c)\)
Chapter 8  Classical Logic

(12) $\forall x_1 x_2 x_3 x_4 \cdot (\lambda x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 \cdot x_7(x_8(\lambda x_{10} x_{11} \cdot x_9(\lambda x_{12} x_{13} \cdot x_6(x_4(\lambda x_{14} x_{15} \cdot x_{10} x_{13}(\lambda x_{16} \cdot x_{14}(\lambda x_{17} x_{18} \cdot x_{15}(\lambda x_{19} x_{19}(\lambda x_{20} \cdot x_{16} x_{20}))))))))) x_{11})) x_{1} x_{2} x_{3} x_{4} f e f e f e :$

$(((a \to (b \to f) \to c) \to a \to f) \to f) \to (((a \to b) \to a \to f) \to f) \to f) \to c.$

Theorem 8.3

Parts (1) and (6) of Theorem 8.2 are also a $BB'Pf_e$ and $BB'IPf_e$ theorems.

Proof:

The following terms and types demonstrate the proofs using $f_e$ and $V_i$:

(1) $\lambda x_1 . \forall x_2 . f_e(x_1(\lambda x_3 . x_2 x_3)) :$

$((a \to f) \to f) \to a$

(6) $\lambda x_1 . \forall x_2 . f_e(x_1(\lambda x_3 x_4 . x_2(x_3 x_4))) :$

$(((a \to b) \to a \to f) \to f) \to b$

==================================
Conclusions

Given a formula $\tau$ of implicational logic, the Long ($\tau$) Generating Algorithm of Bunder determines the (possibly empty) set of $\lambda$ - terms which have $\tau$ as a type and so, via the formulas as types isomorphism, the set of its proofs can be determined. Also using the translations in Chapter 3, the corresponding Hilbert - style proofs can be obtained. This algorithm applies to $SK$ - logic (intuitionistic implicational logic) and has been extended to some weaker logics such as $BCK$, $BCI$, $BCIW$, $BB'I$ and $BB'IK$. Some of these have been implemented by Dekker in Brouwer 6.0.

In this thesis we prove that we can use these algorithms to cover some logics with connectives other than $\rightarrow$.

By defining $\neg a$ by $a \rightarrow f$ where $f$ is a type constant and defining $\&$, $\land$ and $\lor$ in terms of $f$ and $\rightarrow$, we proved some properties of these connectives.

In logics with the extra axiom $f_e : f \rightarrow \alpha$, we converted a formula $\tau$ to one $\tau'$. The proof of $\tau'$ in $SK$ - logic using the earlier algorithms led to the proof of $\tau$ using the extra axiom. We did the same for logics with the Peirce axiom, with or without $f_e : f \rightarrow \alpha$. 
The theorem prover Brouwer 6.0 therefore now extends to full classical propositional logic and to a large number of sublogics such as $\mathbf{BCKf}_e$, $\mathbf{BCIf}_e$, $\mathbf{BB'lf}_e$, $\mathbf{BB'IKf}_e$ and $\mathbf{BCIWf}_e$. Not all of our techniques applied to some weaker logics such as $\mathbf{BB'IP}$ and $\mathbf{BB'IPf}_e$, we proved some results in these logics without Brouwer 6.0.
REFERENCES


[5] Bunder, M.W. & Hirokawa, S. "Classical Formulas as Types of \( \lambda v \) - terms" - University of Wollongong - Department of Mathematics.


===============
