Implementing $\chi$

Jayshan Raghunandand
jr200@doc.ic.ac.uk

June 2004

Project Supervisor: Dr. Steffen van Bakel
Abstract

Calculi are used to reason about functional languages from a theoretical perspective. They allows us to make bold statements about programs based on sound and correct deductive methods.

Methods of visualization allow us to gain a better understanding of the communication and interaction involved between processes. Often, graphs are used because of the level of intuition they can bring to a system. A specific type of graph, called a ‘Term-Graph’ is one method of implementing functional programming languages. It has the added benefit of increasing the efficiency of computations through a concept known as sharing.

In this report we present the design and implementation of a term-graph rewriting system that implements a higher-order calculus. We evaluate the tool and present the main conclusions of the project.
Acknowledgements

I would like to thank the following people who have helped throughout the project.

First and foremost, Dr. Steffen van Bakel, my supervisor, who not only gave great help and excellent guidance throughout the course of the project, but also kept me motivated through continued enthusiasm and positive feedback on my progress.

Professor Chris Hankin, for taking the time to explain concepts I was unsure of, and giving guidance on where to research certain aspects of the project.

Alex Summers, for testing the many implementations of the project (and giving insightful feedback).

Paul Jolly and Jonathan Hayman for answering the hundred \LaTeX{}questions I had when writing the report.

Carlos Costa-Rausa for proof reading and general feedback throughout the entire project.
# Contents

1 Introduction 1

2 Background 3
   2.1 The Lambda Calculus, λ .......................... 3
      2.1.1 Syntax ........................................ 3
      2.1.2 Free and Bound Variables ..................... 4
      2.1.3 β-substitution ................................. 4
      2.1.4 α-conversion ................................. 4
      2.1.5 Confluence .................................... 5
   2.2 Type Systems ..................................... 6
   2.3 Reduction Strategies .............................. 7
   2.4 The Curry-Howard Isomorphism ................... 8
   2.5 The Curien-Herbelin Calculus, λμ̂ ................ 8
   2.6 “The Mother of all Calculi” - χ .................... 9
      2.6.1 Syntax ........................................ 9
      2.6.2 Free and Bound Connectors .................... 9
      2.6.3 Reduction Rules ............................... 10
      2.6.4 Flagged Cuts ................................. 10
      2.6.5 Basic Properties ............................. 12
      2.6.6 Curry Type Assignment ......................... 14
   2.7 Implementing Functional Languages .............. 15
      2.7.1 Term Rewriting Systems ....................... 15
      2.7.2 Graph Rewriting Systems ..................... 16
   2.8 Applications currently available ................ 19
   2.9 Lexers and Parsers ............................... 20
   2.10 Browser Support ................................ 21

3 Understanding χ’ 22
   3.1 χ-Terms .......................................... 22
   3.2 Interaction of Terms ............................. 23

4 Implementation 26
   4.0.1 Lexical Analysis and Parsing ................... 26
   4.0.2 Abstract Data-Types ........................... 27
   4.0.3 Iterators ....................................... 32
   4.0.4 Interpreting the rules ......................... 33
   4.1 Rewriting .......................................... 35
      4.1.1 Rule Matching ............................... 35
Chapter 1

Introduction

Alan Turing’s attempt to formalise the notion of computability gave birth to an idealised computing device known as the Turing Machine. While this model provided an architecture upon which the development of computers would be based, it was too mechanical to allow for effective reasoning of computability.

In 1941, Alonzo Church published his work on a new model to reason about computations, the Lambda Calculus.

The following decades saw the calculus extended by different theorists, in order to reason about different aspects of computing. In 1965, Curry and Howard defined a correspondence between the Lambda Calculus and Intuitionistic Logic, known as the Curry-Howard Isomorphism. For some time now, theorists have been searching for a ‘Curry-Howard’ correspondence with Classical Logic. A recent development in the field is the work of Steffen van Bakel, Stéphane Lengrand and Pierre Lescanne; the formalisation of a calculus which provides this correspondence, named χ.

In this report we present an implementation of the calculus in the form of a term-graph rewriting system. Chapter 2 presents a concise background to the field, and Chapter 3 gives a metaphor for the calculus. Chapters 4 and 5 detail the design and implementation decisions made and Chapter 6 evaluates the success of the project and looks at further work which can be done to extend the tool.
Chapter 2

Background

This chapter presents the origins and motivations of the $\chi$-calculus together with a detailed description of its syntax and semantics. Since $\chi$ is a fairly recent development, there are no existing implementations of interpreters for this calculus. We look at similar calculi and approaches used to efficiently implement reduction systems. We begin with an overview of the $\lambda$-calculus to introduce some key concepts that will be required when implementing $\chi$.

2.1 The Lambda Calculus, $\lambda$

The $\lambda$-calculus is a formal mathematical system devised by Alonzo Church to investigate functions through function application and function application.

2.1.1 Syntax

The syntax of $\lambda$ is short and simple. The set $\Lambda$, of $\lambda$-terms ranged over $M$ can be constructed from the context-free grammar below, given in Backus-Naur form.

$$M ::= x \mid \lambda x. M \mid M_1 M_2$$

The letter $x$ is a member of the set of term variables, $\mathcal{V}\{x, y, z, x_1, x_2, x_3, \ldots \}$.

$\lambda x. M$ denotes function definition or abstraction, which is analogous to the mathematical notation $f(x)=M$ and $x$ is read as a formal parameter of the function. $M_1 M_2$ denotes function application, in which $M_1$ is a function and $M_2$ is the operand. This operation produces a new term, the application of $M_1$ to $M_2$.

To avoid ambiguity when parenthesis are omitted, abstraction associates to the right and application associates to the left.
2.1.2 Free and Bound Variables

Two notions are used to classify variables in an expression. An occurrence of a variable is bound in a λ-term if it is in the scope of any abstraction of a variable with the same name. Otherwise the variable is said to be free. The set of bound and free variables for the λ-calculus are defined by the following functions.

\[ BV : \lambda \rightarrow \mathcal{P}(V) \]

\[
BV(x) = \emptyset \\
BV(\lambda x.M) = BV(M) \cup \{x\} \\
BV(M_1 M_2) = (BV M_1) \cup (BV M_2)
\]

\[ FV : \lambda \rightarrow \mathcal{P}(V) \]

\[
FV(x) = \{x\} \\
FV(\lambda x.M) = FV(M) - \{x\} \\
FV(M_1 M_2) = (FV M_1) \cup (FV M_2)
\]

A closed term is one where \( FV M = \emptyset \), or every variable occurrence in the term is bound.

2.1.3 \(\beta\)-substitution

The rule of execution that tells us how to calculate function application is known as \(\beta\)-reduction. It states that if an abstraction \(\lambda x.M\) is applied to a term \(N\), then the result will be a new \(\lambda\)-term with the body of the abstraction \(M\) altered such that all free occurrences of the formal parameter \(x\) are replaced with \(N\).

\[
(\lambda x.M)N \xrightarrow{\beta} M[N/x]
\]

2.1.4 \(\alpha\)-conversion

If we define the terms \(M = \lambda x\lambda y.x\) and \(N = \lambda x.y\), then consider the result of the function application \(MN\):

\[
MN = (\lambda x\lambda y.x)(\lambda x.y) \\
\xrightarrow{\beta} \lambda y.(\lambda x.y)
\]

We arrive at a reduced term whose occurrence of \(y\) that was initially free is now bound. The problem is known as variable capture, and to solve it we introduce an equality relation which states that the \(\lambda\)-term \(M_1\) is syntactically equal or
α-congruent to the λ-term \( M_2 \) if \( M_2 \) results from \( M_1 \) by a series of renaming bound variables. The example now becomes:

\[
MN = (\lambda x \lambda y.x)(\lambda x.y) \\
\equiv_\alpha (\lambda x \lambda u.x)(\lambda x.y) \\
\longrightarrow^\beta \lambda u.(\lambda x.y)
\]

### 2.1.5 Confluence

When no more \( \beta \)-reductions are possible from an initial sequence of reductions, we say the term has been reduced to *normal form*. The following example shows that not every term will have a normal form.

\[
\Omega = (\lambda x.xx)(\lambda x.xx) \\
\longrightarrow^b (\lambda x.xx)(\lambda x.xx) \\
\longrightarrow^b \ldots
\]

The term \( \Omega \) can always be reduced, but only onto itself and therefore has no normal form.

We may also arrive at an expression where several \( \beta \)-reductions are possible at the same time. This is not a problem since the λ-calculus is a confluent one and any sequence of reductions will reduce to the same single normal form. This is expressed in the theorem of Church and Rosser given below.

**Theorem 2.1** If a term \( M \) can be reduced to terms \( N \) and \( P \), then there exists a term \( Q \) to which both \( N \) and \( P \) can be reduced. (\( \rightarrow \) represents a sequence of zero or more \( \beta \)-reductions).
2.2 Type Systems

A type will give us information on the kinds of arguments and results a term will accept and produce. This information is essential to the compiler if it is to generate efficient machine code.

Before looking at the type system for the Lambda Calculus, a brief overview of Sequent calculus is given as it is often used to describe type systems.

Sequent Calculus

In sequent calculus, conclusions and premises are treated in the same way as Natural Deduction models. However, the proof constructs judgements rather than conclusions.

We introduce the notion of a multi-set, which is a set that can distinguish how often an element occurs in it - an ‘orderless’ list. A sequent is an object of the form:

$$\Gamma \vdash \Delta$$

This is read as “whenever all of the $\Gamma$ are true, then at least one of the $\Delta$ is true.”

Curry Type Assignment

Curry’s system for type assignment expresses function abstraction and function application for the $\lambda$-calculus and makes the following problem decidable. Given a term $M$, is there a basis (a set of statements with only distinct variables as subjects) $B,\sigma$ such that $B \vdash_C M:\sigma$.

1. The following terms will be used when defining the Curry type assignment system.

   (a) The set of types, $T_C$, ranged over $\{\sigma, \tau, \ldots\}$ is defined over a set of type-variables $\Phi$=\{\(\varphi_1, \varphi_2, \ldots\}\} by:

      $$\sigma ::= \varphi \mid \sigma \rightarrow \tau$$

   (b) A statement is an expression of the form $M : \sigma$, where $M$ is a $\lambda$-term $\sigma \in T_C$. We call $M$ the subject and $\sigma$ the predicate of $M : \sigma$.

2. A natural deduction system, below, defines Curry-type assignment and derivation. Note that $(Ax)$ is the axiom, $(\rightarrow I)$ is arrow introduction and
$(\to E)$ is arrow elimination.

$$(Ax) : \frac{\dfrac{B \vdash_C \tau}{\therefore B \vdash_C x : \alpha}}{\therefore (x : \alpha \in B)}$$

$$(\to I) : \frac{B, x : \sigma \vdash_C M : \tau}{\therefore B \vdash_C \lambda x : \tau. M : \sigma \to \tau}$$

$$(\to E) : \frac{B \vdash_C M_1 : \sigma \to \tau \quad B \vdash_C M_2 : \sigma}{\therefore B \vdash_C M_1 M_2 : \tau}$$

3. The property of subject reduction will ensure that types are preserved under reduction.

The principal type property states that if a term is ‘typeable’ (there exists a basis $B, \sigma$ such that $B \vdash M : \sigma$) then there is a principal typing for this term, i.e. a principal basis and principal type $P, \pi$ such that $P \vdash M : \pi$ and for all other basis $B', \pi'$ such that $B' \vdash M : \sigma$ and there exists an operation $O$ that does the following mapping $O((P, \pi)) = (B', \sigma)$.

### 2.3 Reduction Strategies

If we have a complex $\lambda$-term with several sub-terms that could be evaluated next, we need a strategy of choosing the sequence in which to apply reductions. This section outlines some strategies often used when implementing functional languages.

1. Call-By-Name
   The call-by-name strategy allows for lazy evaluation. It only evaluates parameters when needed, which saves time if the parameter is never needed. Intuitively, we can see that function calls will also be less expensive due to the late binding of parameters.

2. Call-By-Value Reduction
   This reduction strategy is a restriction where $\beta$-reduction only occurs when the operand of an application is a value or an abstraction.
2.4 The Curry-Howard Isomorphism

The original Curry-Howard Isomorphism reveals the exact correspondence between the type system of the lambda calculus and the natural deduction formulation of proof systems\(^1\). We can see that typing a \(\lambda\)-derivation gives us the corresponding proof in Intuitionistic Logic.

2.5 The Curien-Herbelin Calculus, \(\lambda\mu\tilde{\mu}\)

This calculus, presented in [CH00], is derived from a key classical logical system in proof theory known as implicational Gentzen’s sequent calculus LK. Its main goal was to develop a sequent calculus version of the call-by-value \(\lambda\)-calculus. It also provided a ‘Curry-Howard’ correspondence with classical logic. [Gir89] formulates classical logic through the following BNF definition:

\[
s ::= 1 | 0 | s \land s | s \lor s | s \rightarrow s
\]

The rules of classical logic are the axiom, \((\rightarrow R), (\rightarrow L)\) and cut.

\(^1\)A formalisation of Intuitionistic Logic, a logic that does not accept the law of excluded middle, \(\forall A. A \lor \neg A\)
2.6 “The Mother of all Calculi” - $\chi$

The main goal of $\chi$ was to get a Curry-Howard Isomorphism for Classical Logic. $\chi$ is defined as a sub/super calculus of $\lambda\mu\tilde{\mu}$ (Curien-Herbelin). $\chi$ is a sub-syntax of $\lambda\mu\tilde{\mu}$ and a superset of $\lambda\mu\tilde{\mu}$ because it can map all $\lambda\mu\tilde{\mu}$ terms to terms in $\chi$ without loss.

2.6.1 Syntax

This section gives the syntax and explanation of the calculus as presented in [vBLL03].

The letter $x$ is a member of the set of sockets or inputs to terms. Sockets are represented by Roman characters. The letters $\alpha$ and $\beta$ are members of the set of plugs or outputs of terms. Plugs are represented by Greek characters. The hat, $\hat{}$, symbolises the connector (socket or plug) is bound in the term. Pure $\chi$-terms are constructed from the following grammar:

$$M ::= \langle x, \alpha \rangle$$
$$\quad | \ M \hat{\beta} \cdot \alpha$$
$$\quad | \ M_1 \hat{\beta} [y] \hat{x} M_2$$
$$\quad | \ M_1 \hat{\alpha} \upharpoonright \hat{x} M_2$$

$\langle x, \alpha \rangle$ is the axiom (capsule), $\hat{\beta} M \cdot \alpha$ represents right introduction, $M_1 \hat{\beta} [y] \hat{x} M_2$ is left introduction and $M_1 \hat{\alpha} \upharpoonright \hat{x} M_2$ is the cut. There also exists a diagrammatic representation for the calculus. See Figure 2.1.

2.6.2 Free and Bound Connectors

The free sockets in a term are:

$$fs(\langle x, \alpha \rangle) = \{x\}$$
$$fs(\hat{x} M \hat{\beta} \cdot \alpha) = fs(M) \setminus \{x\}$$
$$fs(M_1 \hat{\beta} [y] \hat{x} M_2) = fs(M_1) \cup \{y\} \cup fs(M_2) \setminus \{x\}$$
$$fs(M_1 \hat{\alpha} \upharpoonright \hat{x} M_2) = fs(M_1) \cup fs(M_2) \setminus \{x\}$$

The free plugs in a term are:

$$fp(\langle x, \alpha \rangle) = \{\alpha\}$$
$$fp(\hat{x} M \hat{\beta} \cdot \alpha) = (fp(M) \setminus \{\beta\}) \cup \{\alpha\}$$
$$fp(M_1 \hat{\beta} [y] \hat{x} M_2) = (fp(M_1) \setminus \{\alpha\}) \cup fp(M_2)$$
$$fp(M_1 \hat{\alpha} \upharpoonright \hat{x} M_2) = (fp(M_1) \setminus \{\alpha\}) \cup fp(M_2)$$
The bound connectors of a term are those which are not free, i.e. \( fc(M) \cap bc(M) = \emptyset \). The same is true for bound names.

We also introduce the notion of available variables. These are variables that are present during each reduction step and will appear in the normal form for the term and is defined as follows:

\[
\begin{align*}
av((x, \alpha)) &= \{x\} \\
av(\widehat{x}M \cdot \beta) &= av(M) \setminus \{x\} \\
av(M_1[\beta][y] \widehat{x}M_2) &= av(M_1) \cup \{y\} \cup av(M_2) \setminus \{x\} \\
\end{align*}
\]

\[
\begin{align*}
\text{av}(M_1[\alpha] \upharpoonright \widehat{x}M_2) &= \begin{cases} 
\text{av}(M_1) \cup \text{av}(M_2) \setminus \{x\} & x \in \text{av}(M_2) \\
\text{av}(M_2) & x \notin \text{av}(M_2)
\end{cases}
\end{align*}
\]

2.6.3 Reduction Rules

The logical rules for the calculus are as follows:

\[
\begin{align*}
\text{(var)} : & \quad \langle y, \alpha \rangle \alpha \upharpoonright \widehat{x} \langle x, \beta \rangle \rightarrow \langle y, \beta \rangle \\
\text{(exp)} : & \quad (\widehat{y}M \cdot \alpha) \alpha \upharpoonright \widehat{x} \langle x, \gamma \rangle \rightarrow \widehat{y}M \cdot \gamma & \alpha \notin fs(M) \\
\text{(med)} : & \quad \langle y, \alpha \rangle \alpha \upharpoonright \widehat{x} (N[\beta][x] \widehat{\alpha}P) \rightarrow \widehat{N}[\gamma][x] \widehat{\alpha}P & x \notin fs(N, P) \\
\text{(ins)} : & \quad (\widehat{y}M \cdot \beta \cdot \alpha) \alpha \upharpoonright \widehat{x} (N[\gamma][x] \widehat{\alpha}P) \rightarrow \\
& \quad \text{either} \begin{cases} 
N[\gamma] \upharpoonright \widehat{y}(M[\beta] \upharpoonright \widehat{\gamma}P) & \alpha \notin fs(M), x \notin fs(N, P) \\
(N[\gamma] \upharpoonright \widehat{y}M)[\beta] \upharpoonright \widehat{\gamma}P
\end{cases}
\end{align*}
\]

These rules also have a diagrammatic representation. See Figure 2.2.

2.6.4 Flagged Cuts

The syntax is now extended to include flagged cuts. (Diagrammatically, they are represented by the cut figure).

\[
M ::= \ldots \\
\mid M_1 \alpha \not\upharpoonright \widehat{x}M_2 \\
\mid M_1 \alpha \setminus \widehat{x}M_2
\]

We introduce left and right propagation and two rules to activate cuts.
Left Propagation

\[(dL) : \langle y,\alpha\rangle\alpha \not\vdash \hat{x}M \rightarrow \langle y,\alpha\rangle\alpha \vdash \hat{x}M\]
\[(L1) : \langle y,\beta\rangle\alpha \not\vdash \hat{x}M \rightarrow \langle y,\beta\rangle\]
\[(L2) : (\hat{g}M'\beta \cdot \alpha)\alpha \not\vdash \hat{x}M \rightarrow (\hat{g}(M'\alpha \not\vdash \hat{x}M)\beta \cdot \gamma)\gamma \vdash \hat{x}M, \quad \gamma \text{ fresh}\]
\[(L3) : (\hat{g}M'\beta \cdot \gamma)\alpha \not\vdash \hat{x}M \rightarrow \hat{g}(M'\alpha \not\vdash \hat{x}M)\beta \cdot \gamma, \gamma \neq \alpha\]
\[(L4) : (N\beta [z] \hat{g}P)\alpha \not\vdash \hat{x}M \rightarrow (N\alpha \not\vdash \hat{x}M)\beta [z] \hat{g}(P\alpha \not\vdash \hat{x}M)\]
\[(L5) : (N\beta \vdash \hat{g}P)\alpha \not\vdash \hat{x}M \rightarrow (N\alpha \not\vdash \hat{x}M)\beta \vdash \hat{g}(P\alpha \not\vdash \hat{x}M)\]

Right Propagation

\[(dR) : M\alpha \not\vdash \hat{x}(x,\beta) \rightarrow M\alpha \vdash \hat{x}(x,\beta)\]
\[(R1) : M\alpha \not\vdash \hat{x}(y,\beta) \rightarrow \langle y,\beta\rangle\quad y \neq x\]
\[(R2) : M\alpha \not\vdash \hat{x}(\hat{g}M'\beta \cdot \gamma) \rightarrow \hat{g}(M\alpha \not\vdash \hat{x}M')\beta \cdot \gamma\]
\[(R3) : M\alpha \not\vdash \hat{x}(N\beta [x] \hat{g}P) \rightarrow M\alpha \vdash \hat{x}(N\alpha \not\vdash \hat{x}N)\beta [z] \hat{g}(M\alpha \not\vdash \hat{x}P), \quad z \text{ fresh}\]
\[(R4) : M\alpha \not\vdash \hat{x}(N\beta [z] \hat{g}P) \rightarrow (M\alpha \not\vdash \hat{x}N)\beta [z] \hat{g}(M\alpha \not\vdash \hat{x}P), \quad z \neq x\]
\[(R5) : M\alpha \not\vdash \hat{x}(N\beta \vdash \hat{g}P) \rightarrow (M\alpha \not\vdash \hat{x}N)\beta \vdash \hat{g}(M\alpha \not\vdash \hat{x}P)\]

Activating Cuts

\[(act-L) : M\alpha \vdash \hat{x}N \rightarrow M\alpha \not\vdash \hat{x}N \quad \text{If } M \text{ does not introduce } \alpha\]
\[(act-R) : M\alpha \vdash \hat{x}N \rightarrow M\alpha \not\vdash \hat{x}N \quad \text{If } N \text{ does not introduce } x\]

Where (M introduces x), either M = P\alpha [x] \hat{g}Q, and x does not occur free in P,Q or M = \langle x,\delta \rangle. Where (M introduces \delta), either M = \hat{x}N\beta \cdot \delta, and \delta does not occur free in N or M = \langle x,\delta \rangle.

We say a term is connectable if either of the sub-terms of the cut output or input from the channels provided by the cut. The calculus is non-confluent since situations may arise when the cut can be activated in more than one way. An expression can have different normal forms depending on how the cut is activated.

Reduction Strategies

1. The Call-by-Value strategy forbids the activation of cut by (act-R), when the latter could be activated by (act-L), and we write M \rightarrow_{V} N, if M \rightarrow_{A} N, and only those cuts are activated that are permitted by the CBV strategy.
2. The Call-by-Name strategy consists in forbidding the activation of a cut by (act-L) when the latter could be activated by (act-R); like above, we write $M \rightarrow^*_N N$.

2.6.5 Basic Properties

Below are a list of Lemmas provable from the set of logical terms, propagations and activation rules.

1. $M\alpha \not\rightarrow^*_A N \rightarrow^*_A M$ if $\alpha$ does not appear in $M$ and $M$ is pure.
2. $M\alpha \uparrow^*_A N \rightarrow^*_A M$ if $\alpha$ does not appear in $M$ and $M$ is pure.
3. $M\alpha \backslash^*_A N \rightarrow^*_A N$ if $x$ does not appear in $N$ and $N$ is pure.
4. $M\alpha \uparrow^*_A N \rightarrow^*_A N$ if $x$ does not appear in $N$ and $N$ is pure.
5. $M\delta \not\rightarrow^*_A \exists z.\alpha \rightarrow^*_A M[\alpha/\delta]$ if $M$ is pure.
6. $M\delta \uparrow^*_A \exists z.\alpha \rightarrow^*_A M[\alpha/\delta]$ if $M$ is pure.
7. $\langle z.\alpha\rangle\alpha \not\rightarrow^*_A \exists zM \rightarrow^*_A M[z/x]$ if $M$ is pure.
8. $\langle z.\alpha\rangle\alpha \uparrow^*_A \exists zM \rightarrow^*_A M[z/x]$ if $M$ is pure.
Figure 2.1: Axiom (cap), Right Introduction, Left Introduction and Cut for $\chi$

\[
\begin{array}{c}
x, \alpha \quad y, [M] \beta \quad [M_1] \xi[y] \quad [M_2] \\
\frac{\alpha \rightarrow x}{\gamma} \quad \frac{\gamma \rightarrow \beta}{y} \quad \frac{\alpha \rightarrow \beta}{\gamma}
\end{array}
\]

Figure 2.2: The Logical Rules: Var, Exp, Med and Ins for $\chi$

\[
\begin{array}{c}
y, \alpha \quad \alpha \rightarrow x \quad \beta \quad \rightarrow \gamma
\end{array}
\]

\[
\begin{array}{c}
y, \alpha \quad \alpha \rightarrow x \quad [N \xi \frac{M \beta \gamma \rightarrow [P]}{y}]
\end{array}
\]

\[
\begin{array}{c}
y, \alpha \quad \alpha \rightarrow x \quad [N \frac{M \beta \gamma \rightarrow [P]}{y}]
\end{array}
\]

\[
\begin{array}{c}
y, \alpha \quad \alpha \rightarrow x \quad [N \frac{M \beta \gamma \rightarrow [P]}{y}]
\end{array}
\]

or
\[
\begin{array}{c}
y, \alpha \quad \alpha \rightarrow x \quad [N \frac{M \beta \gamma \rightarrow [P]}{y}]
\end{array}
\]

or
\[
\begin{array}{c}
y, \alpha \quad \alpha \rightarrow x \quad [N \frac{M \beta \gamma \rightarrow [P]}{y}]
\end{array}
\]

13
2.6.6 Curry Type Assignment

The types for the $\chi$-calculus are defined by the grammar:

$$A, B :: \varphi | A \rightarrow B$$

1. $\varphi$ is the base type open for substitution.

2. The context of variables, $\Gamma$ is a mapping from variables to types, denoted by a finite set of statements $x : A$ such that the subjects of the statements, $x$ are distinct. We write $\Gamma, x : A$ for the context of variables defined by:

$$\Gamma, x : A = \Gamma \cup \{x : A\} \text{ if } 'G \text{ is not defined on } x$$

$$= \Gamma \quad \text{otherwise}$$

3. The context of names is written $\alpha : B, \Delta$.

4. Type judgements are expressed by the relate $M : \Gamma \vdash \Delta$ where $M$, the witness of this judgement is a $\chi$-term, $\Gamma$ is the context of variables and $\Delta$ is the context of names.

5. Type assignment is defined by the following sequent calculus.

\[
\begin{align*}
\text{(cap)} : & \quad \langle y, \alpha \rangle : \Gamma, y : A \vdash \alpha : A, \Delta \\
\text{(exp)} : & \quad M : \Gamma, x : A \vdash \alpha : B, \Delta \\
& \quad \bar{x}M \tilde{\alpha} : \beta : \Gamma, y : A \vdash \alpha : A \rightarrow B, \Delta \\
\text{(med)} : & \quad M : \Gamma \vdash \alpha : A, \Delta \\
& \quad N : \Gamma, x : B \vdash \Delta \\
& \quad M\tilde{\alpha} [y] \bar{x}N : \Gamma, y : A \rightarrow B \vdash \Delta \\
\text{(cut)} : & \quad M : \Gamma \vdash \alpha : A, \Delta \\
& \quad N : \Gamma, x : B \vdash \Delta \\
& \quad M\tilde{\alpha} \vdash \bar{x}N : \Gamma \vdash \Delta
\end{align*}
\]

6. We can name a derivation $D$ as, $D :: M : \Gamma \vdash \Delta$
2.7 Implementing Functional Languages

Originally, computing was done at the lowest level based on the primitive language of the machine. It progressed with the development of low-level then high-level languages which were then improved by functional and logical programming languages. There are a number of models to implement functional languages including term rewriting systems, graph rewriting systems, the three instruction machine and the infamous G-Machine. This section concentrates on Term Rewriting Systems and Graph Rewriting System.

2.7.1 Term Rewriting Systems

Definition

[Klo92] defines a TRS as the pair $(\Sigma, R)$, where $\Sigma$ is the signature and $R$ is a set of rewrite or reduction rules.

1. The signature consists of:
   (a) A countably infinite set of variables $x_1, x_2, x, y, x', y', \ldots$
   (b) A non-empty set of function symbols $F, G, \ldots$ each equip with an arity specifying the number of arguments it can take. 0-ary function symbols are called constant symbols.

2. The set of reduction terms ranged over $\Sigma$ is $\text{Ter}(\Sigma)$ and is defined inductively:
   (a) $x, y, z \in \text{Ter}(\Sigma)$
   (b) if $F$ is an $n$-ary function symbol with arity $\geq 0$, and $t_1, \ldots, t_n$ in $\text{Ter}(\Sigma)$, then $F(t_1, \ldots, t_n) \in \text{Ter}(\Sigma)$

3. Terms not containing a variable are called ground or closed terms denoted by $T_0$. Terms with no duplicate occurrences of variables are called linear.

4. A rewrite rule, $r \in R$ is a pair $(l, r)$ of terms $\in \text{Ter}(\Sigma)$. It will be written $r : l \rightarrow r$.

Confluence and Termination

In general, the process of term rewriting will continue until no further rules can be applied. This point is called a normal form. Confluent TRSs are composed of rewrite rules that will ensure a unique normal form.

A terminating system is one which does not allow infinite reductions. In [HL78], Huet and Lankford reduced the uniform halting problem for Turing machines
to the termination problem for rewriting systems. It is also well-known that outermost rewriting strategies have better termination than innermost rewriting strategies [HL91].

Rewriting Strategies

A term may contain many redexes. The rewriting strategy determines the next rule to be applied. A well-known strategy is the rightmost innermost, which chooses the rightmost redex that does not contain another redex. Alternatively, we can apply a partial ordering on the rewrite rules as seen in priority rewrite systems (PRSs) in [BBKW89]. The downfall of such systems is that they are often expensive and come with problematic operational semantics. Lazy evaluation is also an option when choosing a rewrite strategy.

2.7.2 Graph Rewriting Systems

The use of graphs to represent complex situations often leads to an intuitive understanding of the situation. We can use graphs to represent functional expressions, and evaluate these expressions through rule-based graph transformations. They allow for optimisations such as sharing of common subexpressions. [Plu98] presents a good introduction to term graph rewriting systems, and defines them as follows.

Definition

From now on, the term graph refers to a labelled, rooted, ordered, acyclic directed set of nodes.

1. Formally, a term graph \( G \) is defined as the tuple \( (N, lab, succ, r) \) over a signature \( \Sigma = (F, V) \) divided into function symbols \( F \), variables \( V \) where:
   
   (a) \( N \) is a finite set of nodes, where each node is the result node of a uniques edge.
   
   (b) \( lab \) is the labelling function \( N \rightarrow \Sigma \). At most, there is one label per node.
   
   (c) \( succ \) is the successor function \( N \rightarrow N^* \), and \( N^* \) is the ordered set \( \{n_1, n_2, \ldots, n_k\} \) corresponding to the successors or arguments of node \( n \). \( k \) is the arity of \( n \). The \( i^{th} \) component of \( succ(n) \) is written \( succ(n)_i \).
   
   (d) \( r \in N \) is the root node. We do not require that every other node in the graph is reachable. \( G|r \) is read as the subgraph rooted at \( r \).

2. An open graph is one where \( lab \) and \( succ \) are only required to be partial functions on \( N \). A node on which \( lab \) and \( succ \) are undefined, denoted by
the symbol $\bot$, is called an empty node. A closed graph is one without empty nodes.

3. Graph substitution, more commonly referred to as homomorphism is a function that makes two graphs identical by replacing variables in one graph with any graph valid in the GRS. Some examples are given below.

Formally, a homomorphism is defined as the mapping $h : N_1 \rightarrow N_2$ between two graphs, $G_1 = (N_1, lab_1, succ_1, r_1)$ and $G_2 = (N_2, lab_2, succ_2, r_2)$ that has properties such that structure is preserved. That is $\forall n \in N_1$:

(a) Labels are preserved, $lab_2(h(n)) = lab_1(n)$ where $lab_1(n) \not\in V$
(b) Edges are preserved, $succ_2(h(n)) = h(succ_1(n))$

4. An isomorphism is any homomorphism $h$ which has an inverse homomorphism, $h'$: $G_2 = h(G_1)$ and $G_1 = h'(G_2)$. Also, the identity homomorphism is defined as $G = id(G)$ for any $G$.

5. A rooted homomorphism obeys $r_2 = h(r_1)$, and a rooted isomorphism is one where both forward and inverse homomorphisms are rooted.

6. A rooted isomorphism defines equivalence between two graphs. The notion of graph equivalence is defined as rooted isomorphism, written as $g_1 \simeq g_2$.

Basic Rules of GRS

[Ari93a] identifies a set of GRS terms whose differences can be regarded as syntactic noise, similar to the notion of $\alpha$-equivalence in the $\lambda$-calculus.

1. Only constants and variables can be freely substituted.

$$
\frac{X = V}{X \rightarrow V} \quad \frac{X = Y}{X \rightarrow Y} (X \neq A)
$$

2. With bindings such as $x = x$, we can bind $x$ with the symbol $\circ$, where the symbol $\circ$ behaves as a constant value which can be freely substituted.

$$X = X \rightarrow X = \circ$$
3. The block flattening rule renames all bound variables in an internal block to avoid name clashes with variables in the surrounding scope.

4. The commutativity rule states that the order of bindings in a block does not affect a term.

5. We say a term is in canonical form if all the substitutions, detection of degenerate cycles, flattening of blocks and bindings of the from $x = y$ and $x = v$ have been deleted.

6. Two terms are $\alpha$-equivalent if their canonical forms are the same up to renaming of bound variables and commutativity of bindings.

Graph Rewriting Rules

[McB93] defines graph rewriting as follows:

1. A graph rewrite rule is a triple $(G_r, r_h, r_b)$, conventionally written as head $\rightarrow$ body, where:
   (a) $G_r$ is an open labelled graph.
   (b) $r_h$, the head node is the left root of the rewrite rule.
   (c) $r_b$, the body node is the right root of the rewrite rule.

2. A redex in a graph $G_0$, is the pair $\Delta = (R, h)$, where $R$ is a graph rewrite rule $(G_r, r_h, r_b)$ and $h$ is an occurrence of $R$, or the homomorphism from $G_r|r_h$ to $G_0$. In other words, the redex of the graph is $h(r_h)$ and the reduct (contractum) is $h(r_b)$.

3. The application of a graph rewrite rule to a graph, $G_0$, has four steps:
   (a) Matching: find a node $n$ and homomorphism $h$ such that $G_0|n = h(r_h)$ and $r_b - r_h = h(r_b - r_h)$.
   (b) Building: make an isomorphic copy of $g(r_b - r_h)$ and add this to the graph $G_0$, forming the new graph $G_1$.
   (c) Redirection: update the succ function and root of $G_1$ forming graph $G_2$ as follows:
      i. $\forall n, i$, if $\text{succ}_1(n)_i = h(r_h)$ then $\text{succ}_2(n)_i = h(r_h)$ else $\text{succ}_2(n)_i = \text{succ}_1(n)_i$
      ii. if $r_1 = h(r_h)$ then $r_2 = h(h_b)$ else $r_2 = r_1$
   (d) Garbage Collection: remove any nodes $n \in N_2$ for which no path exists from $r_2$ to $n$. This final graph represents the rewrite of the original graph $G_0$. 

18
2.8 Applications currently available

(There are currently no existing $\chi$ interpreters to evaluate).

In this section we look at the features of existing theorem provers, in particular those presented with graphical user interfaces. Additionally, we give an overview of an existing functional programming language interpreters based on term-graph rewriting.

Coq and CtCoq

Coq is a proof system which handles calculus assertions, and mechanically checks proofs of these assertions. It allows one to write formal specifications, programs and verifies programs against their specification. Using the Curry-Howard Isomorphism, it formalises programs, and properties of proofs to the typed $\lambda$-calculus.

A mode of interaction is provided, allowing the user to develop theories step-by-step, back-tracking if needed. An output file provides a compact representation of its input. [BB99] gives a breakdown of the basic features one would find in a good interactive theorem prover, taking CtCoq as an example. These features are listed below:

1. **Structured Editing**: the syntactic structure of the language is made more apparent by representing data as trees. Fonts and colours are used to highlight special notations. The system also offers a conventional text-editing interface.

2. **Pretty Printing and Notations**: each rule in the language is given an extensible layout description such as colour, font, size etc. Layout is computed incrementally to allow selections of sub-expressions.

3. **Multiple Window Organisation**: more than one proof can be open at one time. These proofs can be used collaboratively to attempt to reach a common goal.

Some more advanced features are also discussed such as:

1. **Proof by pointing**: the proof starts by stating the goal. The user can then apply commands to reduce this goal to simpler sub-goals. The user can point to sub-formulae in the goals and elaborate on them. This pointing is done with a device such as a mouse.

2. **Script Management**: the user is allowed to define some strategies to attempt to reach a goal.

3. **Textual Explanation of Proofs**: an English description of the proof is given.
The Prototype Verification System (PVS)

PVS is based on higher-order sequent calculus for classical logic. PVS consists of a specification language - higher order and recursive definitions, and a rich type system. Its theorem prover consists of powerful automatic components such as linear arithmetic. It is interesting to note that it holds a library of existing specifications/theories.

PVS uses Gnu or X Emacs to provide an integrated interface to its prover. Commands are selected by pull-down menus or by programmable extended Emacs commands. Help, status-reporting and browsing tools are also available, as well as the ability to generate typeset specifications (in user-defined notation) using LaTeX. Proof trees and theory hierarchies can be displayed graphically using Tcl/Tk.

Clean

Clean was originally designed to be an experimental intermediate language to study new concepts and strategies of functional programming implementation. To name a few: term graph rewriting, lazy copying, abstract reduction and uniqueness typing were implemented without too many issues. A successor of Clean is Concurrent Clean which features term graph rewriting semantics and lazy evaluation. It allows one to define sharing and cyclic structures and is based on the Milner/Hindley type assignment system.

Clean, being a term graph rewriting system, works on graphs rather than terms; often the programmer is unaware of this. [Pla00] explains the main difference between Clean and other languages is that “when a variable occurs more than once in a function body, the semantics prescribe the actual argument is shared”. Essentially, it is modelled on the definitions given in Section 2.7.2. The reduction strategy chosen (by default) is the functional reduction strategy which corresponds to the normal order reduction in the $\lambda$-calculus.

2.9 Lexers and Parsers

Lexical Analysis is the first stage of processing a language. The input to the interpreter is simply a stream of characters. A lexer *recognises* a stream of input characters and groups them into the reserved words, constants, identifiers and symbols that are defined in the language. These “tokens” are then passed to the parser.

A number of different lexical analysers exist for Java. JFlex appears to be the most popular choice of lexer, and is still being updated. It can be downloaded from [http://jflex.de/](http://jflex.de/).
The parser (a.k.a. the syntactic analyser) examines the source and ensures the
tokens are arranged in valid grammatical constructs as defined by the language.
The output of the parser is a parse-tree.

JFlex works well with the LALR Parser Generator, CUP downloadable from http://www.cs.princeton.edu/appel/modern/java/CUP/.

2.10 Browser Support

The intention is to implement the interpreter as a Java Applet - viewable in the
user’s Internet browser or downloaded and run as a standalone application.

Due to different standards being followed by the mainstream Internet browsers
during their development, there are some outstanding issues concerning the
display of some Unicode characters - including the Greek symbol set.

[W3C] provides a summary on when to use Greek characters and symbols. To
summarize, Greek letters can be represented on a webpage in the following ways:

1. Greek characters may be represented by glyphs in the Adobe font "Symbol".

2. There exists an entity set to display modern Greek characters. They can
be referred to by their entity reference, or the numerical equivalent for the
entity reference. For example, the Greek letter alpha, α can be represented
by the entity reference \textalpha;.

(A list of entity references is also available from [W3C]).
Chapter 3

Understanding $\lambda$

The $\lambda$-calculus models functional programming languages in terms of function abstraction and function application. Function abstraction defines a function in terms of its body and its arguments. In the LC, a step of computation is expressed via $\beta$-reduction, which expresses the idea of function application. A large portion of the project was spent trying to understand what is happening with terms in $\lambda$ and how they interact. In this section, we describe a metaphor that may be useful when trying to understand $\lambda$.

3.1 $\lambda$-Terms

The proposed analogy of thinking about $\lambda$-terms is to imagine the terms as groups of people trying to communicate with each other, see Figure 3.1.

![Figure 3.1: Metaphor for cap, exp, med, cut](image)

The capsule is a pair of components: a socket which acts as the input channel, and the plug, which acts as the output channel. The two components are linked
together, in the sense that the capsule reads from the input channel and writes to the output channel. (Sockets are reminiscent of variables, and plugs are reminiscent of continuations).

The export term can be seen to describe a function. The illustration shows a group of people, some of which are ‘supervised’ by the two actors (with hats) on either side. The left-actor reads input and passes it to members of the group for which it is the supervisor. The group members can be thought to perform some function on the input. The right-supervisor receives output from members of the group which it supervises. Communication can only take place if the input/output channel colours match up. We can retrieve the entire function described by the supervisors and group from a separate output channel. This output channel also introduces the arrow type in to the system.

The mediator term describes the following scenario. Two groups exist and have designated a spokesman to speak for them (the actors with hats). The groups cannot communicate directly, as they speak different languages. A place is specified in the scenario where a translator can be inserted in order to facilitate communication between the two groups.

The cut term simply tries to join two groups together so they may communicate directly, and bypass the spokesman.

### 3.2 Interaction of Terms

A well-connected term is one where two terms communicate via compatible input and output channels. The logical rules for the calculus, describe how well-connected terms reduce. Figure 3.2 illustrates the interaction of two capsules, an export with a capsule and a capsule with a mediator. If we look at these terms closely, we can clearly see a simple renaming has taken place. A plug/socket in the original term has been replaced with a plug/socket from the capsule.

The renaming rules do not describe the execution step of the calculus. This is illustrated in Figure 3.3. An export term exists that provides a well-connected translation service such that two groups in the mediator scenario can communicate. The translation service is drawn between the two mediator groups allows the two groups to communicate.
Figure 3.2: *Logical Rules I*

Figure 3.3: *Logical Rules II*
Figure 3.4 shows two scenarios where groups attempt to communicate but are not well-connected. The first shows the situation where one group is not communicating via the channel provided by the cut. The second shows the situation where one group is not communicating at all via the channel provided by the cut. In these situations, the cut ‘activates’ itself. The purpose of this is to allow itself to be replicated and propagated within terms. The activated cut propagates in the direction of the non-communicating term in an attempt to communicate with all members of the group directly, bypassing the spokesman of the initial group (if there was one).

Note that a cut can be activated and propagated in two ways, depending on which group is not communicating on the cut. This introduces non-confluence in the calculus.
Chapter 4

Implementation

The functionality of the TGRS and its implementation is outlined in this section. We begin by discussing an implementation of a basic-working TGRS before considering optimisations that could be made to the system. Following this, optimisations are considered and implementation methods detailed.

4.0.1 Lexical Analysis and Parsing

All valid $\chi$-expressions are built from the abstract grammar defined in [vBLL03].

\[
M \ ::= \ (x, \alpha) \quad \text{(cap)} \\
| \ yM_2 \cdot \alpha \quad \text{(exp)} \\
| \ M_1 \beta \ [y] \ xM_2 \quad \text{(med)} \\
| \ M_1 \alpha \ \hat{\cdot} \ xM_2 \quad \text{(cut)} \\
| \ M_1 \alpha \ \hat{\cdot} \ xM_2 \quad \text{(cutL)} \\
| \ M_1 \alpha \ \hat{\cdot} \ xM_2 \quad \text{(cutR)}
\]

Some conventions were chosen to allow easy input into the interpreter. Greek characters are represented by preceding its Roman equivalent by the grave accent character (Unicode[0x0060h]: `). For example, $\alpha$ would be represented as `$a$'. Additionally, the cut symbols: $\hat{\cdot}$, $\hat{\cdot}$, $\hat{\cdot}$ are input as $+$, $+L$ and $+R$ respectively. Extra notation for bound terms (connectors with the $\hat{\cdot}$ symbol) is not necessary as the syntax for terms is fixed and there can be no confusion possible.

With this information, JFlex is used to generate a lexical analyser from which JavaCUP can recognise valid tokens. JavaCUP is a bottom-up or shift/reduce parser. It shifts recognised tokens onto a stack until the top of the stack can be
reduced by matching it with a defined production rule. The lexical definition
for a connector is as follows:

\[
\begin{align*}
\text{ALPHA} & = [a-z] \\
\text{GREEK} & = "\`"[a-z] \\
\text{INDEX} & = 0 | [1-9][0-9]* \\
\text{NAMEDTERM} & = [A-Z][A-Za-z0-9]* \\
\text{SOCKET} & = \{\text{ALPHA}\}|\{\text{ALPHA}\}\{\text{INDEX}\}+ \\
\text{PLUG} & = \{\text{GREEK}\}|\{\text{GREEK}\}\{\text{INDEX}\}+ \\
\text{CONNECTOR} & = \text{PLUG} \mid \text{SOCKET}
\end{align*}
\]

Two types of conflicts exist with bottom up parsers. Shift/reduce conflicts
occur when it is possible to perform a shift or a reduce action in a given state.
It may also be possible to reduce by more than one production. This is known
as a reduce/reduce conflict. The bracketing around the capsule can be ignored
without introducing ambiguity into the grammar, any other bracketing is made
explicit. The ASCII definition of the grammar is given below.

\[
M ::= <x.\ 'a/> | (yM\ 'b.\ 'a) | (M\ 'b[y]\ xN) | (M\ 'a+xN)
\]

Extending this with flagged cuts we have:

\[
M ::= ... \mid (M\ 'a+LxN) \mid (M\ 'a+RxN)
\]

### 4.0.2 Abstract Data-Types

In Section 2.7.2 we presented an overview of the definition for a graph struc-
ture suitable to be used in the context of a TGRS. A graph, \( \mathcal{G} \), is the tuple
\((N, \text{lab, succ, } r)\) over a signature \( \Sigma = (F, V) \) of function symbols and variables.
The following relates these definitions to \( \chi \).

**Definition 4.1** The signature, \( \Sigma = (F, V) \), of the TGRS is:

\[
\begin{align*}
F & = \{\text{cap, cut, med, exp, } \perp\} \\
V & = \{P, S\}
\end{align*}
\]

Where \( \perp \) is defined to be any node in the set \( F \), \( P \) is the plug connector and \( S \)
is the Socket connector.
Representation of Terms

Before continuing with the definitions, it is necessary to determine the level of sharing that we will use when building the graphs. There are a number of different representations of $\chi$-expressions as graphs.

The function terms $\text{cut}$, $\text{exp}$ and $\text{med}$ bind succeeding function symbols over connectors. We could choose to represent this information in the graph representation for these functions (see Figure 4.2). Here we introduce the intermediary nodes $\text{BP}$, $\text{BS}$ and $\text{BD}$ which binds terms to plugs, sockets or both connectors respectively. This is in agreement with graph representations for the $\lambda$-calculus, where an abstraction $\lambda x.M$ is represented as shown in Figure 4.1.

$$
\begin{array}{c}
\text{r} \\
\downarrow \\
\lambda \\
\downarrow \\
M \\
\downarrow \\
x
\end{array}
$$

\textbf{Figure 4.1: Term-Graph for function abstraction in the $\lambda$-calculus}

If we consider the successors individually, and do not discriminate between bound and unbound nodes, Figure 4.3 shows another possible representation for the terms. Note, the definition for the $\text{cap}$ graph stays the same since it has no bound terms and its only successors are graph variables.

\textbf{Figure 4.2: Graph representations for Cut, Exp, Med}

The initial implementation chosen was the first representation where intermediary nodes captured the essence of binding. However, it was thought the intermediary bindings nodes introduced unnecessary complexity in to the graph representation and the idea leaned too much on the $\lambda$-calculus notion.

The second approach of structuring the graphs was also preferred in the implementation, since it allowed a clearer representation of the terms, (Figure 4.3). This approach is additionally more memory-efficient and requires less objects to be created per term-graph, so should also be faster aswell.

\textbf{Definition 4.2} \textit{The successor function can now be derived from the agreed graph structure, where $\text{succ} :: \Sigma \rightarrow N^*$.}
The $i^{th}$ successor of a node, $n$, is written as $\text{succ}(n)_i$.

Figure 4.3 also shows this information in graphical form. It is necessary to maintain the ordering of the successor function, otherwise we would have to introduce extra information to determine left terms from right terms.

The traditional approach to implementing a graph data-type is to keep a record of the nodes and edges in the graph. Nodes are instances of function symbols or variables, and edges are tuples that relates nodes to their immediate successors. When attempting to apply this implementation to our TGRS we lose some important information in the graph.

If we represent the mediator structure (shown in Figure 4.3) in this format, the following sets are obtained.

Nodes $= \{\text{med, } \bot_1, \bot_2, P, S_1, S_2\}$
Edges $= \{(\text{med, } \bot_1), (\text{med, } \bot_2), (\text{med, } P), (\text{med, } S_1), (\text{med, } S_2)\}$.

Using this information alone, we have the following inconsistencies:

\[
\begin{align*}
\text{succ(med)}_0 &= \bot_1 \quad \text{or} \quad \text{succ(med)}_0 = \bot_2 \\
\text{succ(med)}_2 &= S_1 \quad \text{or} \quad \text{succ(med)}_2 = S_2 \\
\text{succ(med)}_3 &= S_1 \quad \text{or} \quad \text{succ(med)}_3 = S_2 \\
\text{succ(med)}_4 &= \bot_1 \quad \text{or} \quad \text{succ(med)}_4 = \bot_2
\end{align*}
\]
We either need to extend the edge tuple to convey the position of the node in the successor function, or explore an alternative way of implementing the data-types. During the design phase of the project, the following alternatives were considered.

1. Include the successor information in the edges tuple.

   An entry in the edges set would be a tuple \((F \times \text{succ\_position} \times \Sigma)\).

   For example, the edges set for the mediator rule would be:
   \[
   \text{Edges} = \{(\text{med},0,\bot_1),(\text{med},1,\bot_2), (\text{med},2,P), (\text{med},3,S_1), (\text{med},4,S_2)\}.
   \]

2. Grouping of the Successors

   Associate an ordered list of successors with each physical node in the graph. The grouping for the successor nodes in our mediator example would become \((\text{med},[\bot_1,P,S_1,S_2,\bot_2])\).

In Java, object creation is expensive in terms of time and memory. The first approach creates more objects per node than the second. Furthermore, the second approach lends itself more easily towards an object-oriented implementation, since we only need to associate a list of successors with each node.

Figure 4.6 shows the decided implementation for the node structure. Abstract classes are shown in red, concrete classes in blue. Each \texttt{Node} maintains a record of the number of nodes pointing to it which will be useful when implementing garbage collection. The \texttt{Connector} records its name, and \texttt{FunctionSymbols} record an ordered list of their successors. Also, a field is associated with the \texttt{Cut} class which determines whether it is an activated cut or not. Possible values for this field are given by the enumeration \{\texttt{CUT\_NORMAL=0, CUT\_LEFT=1, CUT\_RIGHT=2}\}. The labelling function for nodes, defined in Section 2.7.2, is obtained using the \texttt{instanceof} Java keyword.

**Fresh Variables**

Rules \(L_2\) and \(R_3\) of the calculus introduce the notion of fresh connectors into the system. This is a new connector whose name does not appear in the expression, or in graph terms, it is a connector that appears only in the body of the rule graph. During execution of a term, the names for these connectors must be dynamically generated. A simplistic approach for obtaining fresh variable names is to maintain a set or “variable universe” of unused plugs and sockets in the graph. While this approach seems expensive at first, it does have its advantages.

When an expression requires a fresh variable, it retrieves and removes a name from the variable universe. Conversely, when an execution of an expression causes a variable to disappear from a term, its name can be recycled and so it is reintegrated with the variable universe. We do not have to iterate through the set of nodes in the graph to determine the next fresh variable, an expensive operation when the graph structure becomes large.
The overhead is that we are potentially reserving more memory than may be required by the execution. The variable universe exposes the following method: `nextPlug()`, `nextSocket()`, `addPlug(Plug p)`, `addSocket(Socket s)`.

We can now define a graph and rule-graph data type as in Listing 4.1.

```java
public class Graph {
    protected Set nodeSet;
    protected VariableUniverse names;
    protected Node root;
}

public class RuleGraph extends Graph {
    protected String ruleName;
    protected Node body;
}
```

Listing 4.1: Graph and RuleGraph Structures
4.0.3 Iterators

The graph structure stores its set of associated nodes in the `java.util.Set` data type. The default `Set` iterator has no specific order for node traversal. Since there are likely to exist nodes in the node-set that have yet to be garbage collected, time is wasted checking these nodes. Traversing the nodes in the graph based on the successor information ensures we only consider nodes that are still referenced in the term-graph. The implementation of two graph traversal algorithms are detailed here and are used later by the rewriting phase of the TGRS.

Breadth-First Graph Traversal

The `java.util.Iterator` interface requires implementation of three methods:

- **public boolean hasNext()** : checks if there remain unvisited nodes
- **public Object next()** : returns the next node to be visited by the traversal algorithm
- **public void remove();** : skips the next node to be visited by the traversal algorithm

The breadth-first traversal class maintains an ordered work-list of nodes to be traversed. The algorithm terminates when the work-list is empty, otherwise nodes are selected from the front of the list.

As a node is removed from the `workList`, its children are added to the end. In the case that the `Node` is an instance of a `Connector`, the `getArity()` function returns zero and no extra work is added; (see Listing 4.2).

```java
public Object next(){
    Node result = (Node)workList.get(0);
    workList.remove(0);

    for (int i=result.getArity();i>=0;--i){
        workList.add(result.getNode(i));
    }

    return result;
}
```

Listing 4.2: Breadth-first traversal
Pre-Order Graph Traversal

The iterator maintains a work-list in the form of a stack. The algorithm terminates when the stack is empty, and nodes are removed from the top of the stack. As a node is popped from the stack, its children are pushed to the top of the stack. This ensures each node is visited before any of its children.

```java
public Object next(){
    Node result = (Node)stack.pop();
    for (int i=result.getArity();i>=0;--i)
        stack.push(result.getNode(i));
    return result;
}
```

Listing 4.3: Pre-order traversal

We ensure left-most nodes are traversed first by cycling through the children in reverse order. This is a requirement when implementing functional strategies such as innermost and outermost reduction.

4.0.4 Interpreting the rules

Each of the rules in Section 2.6.3 must be interpreted and represented as a graph rewrite rule. Recall that a rewrite rule is a triple \((G_r, r_h, r_b)\) where \(G_r\) is an open labelled graph, \(r_h\) the left root of the rewrite rule and \(r_b\) the right root of the rewrite rule. If we take the rule \((L2)\), we can construct a ‘rule graph’ for the term by following the process:

1. Determine the set nodes present in the head of the rule. Binding connectors are shared locally with free connectors in the sub-term of the same name, as are free plugs and sockets in the term.
2. Build the head of the rule by connecting the edges according to the successor function for each node.
3. Determine the set of nodes introduced by the body of the rule.
4. Add associations between each new node in the body of the rule graph and its successors.

This is shown graphically in Figure 4.5, and the complete set of rule-graphs for the TGRS is listed in Appendix A.
Figure 4.5: Rule Generation Steps
Storage of Rules

The implementation of the TGRS stores the set of graph rewrite rules in a structure called RuleList. The class encapsulates a java.util.List object containing the set of RuleGraph objects in the system, and provides handy accessor functions to retrieve a rule by its name or index into the list.

4.1 Rewriting

The rewriting component is responsible for searching and applying each rule to the redex. This functionality is grouped together in a single class called ReductionEngine, and performs the following fours key tasks:

1. Rule matching and homomorphism creation
2. Building
3. Graph redirection
4. Garbage collection

4.1.1 Rule Matching

Once the system has been initialized and the rules loaded, the user enters an expression into the tool. The lexical analyser and parser builds a graph representation of the term. This expression must then be reduced by applying matching rules to the expression.

In a pure term-graph rewriting system, this test of matching is done entirely on structure. Our system is a conditional term-graph rewriting system, meaning the rules in the system have side conditions which must also be matched.

Structural Matching

The definition given in Section 2.7.2 specifies a match between a rule \((r_h \rightarrow r_b)\) and a node, \(n\), of the graph, written \(G|n\) by defining a homomorphism. A homomorphism is a correspondence between the nodes in a term graph and the nodes in a rewrite rule, that respects the structure of graphs. The structural matching function by itself does not require a homomorphism to be built, it is purely an equality check on node types and node successors. However, if a match is found, the homomorphism is needed in the redirection phase of rewriting.

It is possible to combine the homomorphism building phase with the matching phase. If we have a number of rules in the system, or the rules are large,
then this may not be desired, since there is a cost associated with building homomorphisms for graphs that only partially match a rule structure.

When building the homomorphism in Java, an object to store the relation must be created for each successful match until either the entire rule has been matched or a we reach a node in the expression that is not a match. In the latter case, the morphism structure must be reinitialised before the next rule can be matched to avoid conflict.

Since we are dealing with a conditional graph rewriting system, homomorphism generation during the structural matching phase is a necessity, since we need these mappings when checking side conditions of rules.

A Morphism structure is created and the Node class is extended to include a matching function (Listing 4.4). Recalling the class hierarchy for the Node structure, the post-conditions for the matching function given in Definition 4.3.

### Listing 4.4: Implementing Structural Matching

```java
class Node {  
    public boolean matches(Node ruleNode, Morphism m) {  
    }
}
class Morphism {  
    private Set map;

    public Node lookup(Node ruleNode) {  
    }
    public Node add(Node ruleNode, Node graphNode) {  
    }
    public void clear() {  
    }
}
```

**Definition 4.3** Post Conditions for Node.matches(Node ruleNode, Morphism m)

\[
\text{this} \text{.} \text{matches} \left( \text{ruleNode}, \text{m} \right) \iff \left\{ \begin{array}{l}
\text{this} \text{.} \text{getClass()}.\text{equals}(\text{ruleNode} \text{.} \text{getClass()}) \land
\forall i : \text{Int} \quad \left( 0 \leq i \leq \text{this} \text{.} \text{getArity()} \land \\
\text{succ(this)}_i \text{.} \text{matches}(\text{succ(ruleNode)}_i) \right)
\end{array} \right.
\]

**ruleNode instanceof Cut:**

\[
\text{this} \text{.} \text{matches} \left( \text{ruleNode}, \text{m} \right) \iff \text{this} \text{.} \text{cutType} = \text{ruleNode} \text{.} \text{cutType}
\]

*If a match is successful, the tuple (ruleNode, this) is added to the set of morphisms. i.e.*

\[
\text{this} \text{.} \text{matches} \left( \text{ruleNode}, \text{m} \right) \rightarrow (\text{ruleNode}, \text{this}) \in \text{m}
\]
Conditional Matching

In our system, there are four distinct conditional tests based on equality, purity, ‘connectability’ and bound variables. It is possible to inline the conditional tests immediately after a homomorphism has been stored for a node in the structural matching phase, however an initial implementation separates the two for simplicity.

Before an implementation is specified, it is important to distinguish between the rule instance and the matching instance of a conditional. A List of rule-conditions is stored with each rule in the calculus.

We introduce the interface RuleCondition and implementing classes Connectable, Equals, FreeVariable, Pure which define the algorithms for testing the conditions.

Figure 4.6: Conditional Matching Scheme

1. Equals

Tests for equality are used by rules \{L1, L3, R1, R4\}. The test involves checking whether two Connectors are the same object. This is essentially the same as checking that the names of the two connectors are equal with the added flexibility that there may arise a situation when two different connectors have the same name. In this case, we would want the equality test to fail.

When a rule with the equality condition is saved, an Equals object is created storing the two rule nodes that are related. When checking whether a rule condition holds, the corresponding graph nodes are retrieved from the morphism and the equality test applied to these nodes.

The negation of this rule is found by applying the boolean NOT operator to the result.
2. Pure
Tests for purity are used by rules \{gc-L, gc-R, ren-L, ren-R\}. A term is pure if it contains no active cuts. i.e.

\[
\text{Pure}(M) \iff \neg \exists c : \text{Cut} \left\{ \begin{array}{l}
    c \in \sim \text{succ}(M) \\
    \land (c.\text{cutType} = \text{Cut}.\text{CUT}_\text{LEFT} \lor \\
    c.\text{cutType} = \text{Cut}.\text{CUT}_\text{RIGHT})
\end{array} \right.
\]

Note that \( \sim \text{succ}(M) \) is defined as the reflexive transitive closure of \( M \). The implementation for this rule in the system is to check whether a \text{Node} is an \text{instanceof} a \text{Cut}, and whether this \text{Cut} is activated, based on the \text{Cut.cutType} attribute for the node.

3. Free Variable
Section 2.6.3 gives the formal definition for free plugs and free sockets in \( \chi \). A variable is free in a term if it occurs in the term \text{and} there exists a path from that term to the variable without meeting any sub-terms whose binders also point to that variable.

The algorithm is implemented through the use of a work-list. The work-list contains only sub-terms that do not bind the variable in question. The algorithm terminates on finding \text{any} free occurrence of the variable. If the variable is not bound at a particular term, its sub-terms must be checked. We cannot assume the variable is free at this point since the variable may not occur in the term at all.

If a variable occurs bound in a function-symbol with more than one sub-term (\text{Cut} and \text{Med}), we must still check whether the variable occurs free in the opposite term. For example, if a variable is bound in the left hand side of a Cut, we no longer need to check the left hand side for occurrences since any occurrence will be bound. However, we must still check whether it is free in the right hand side, with the exception when the left and right hand side point to the same sub-term (see Figure 3).

\[
\begin{array}{c}
\Downarrow_1 \quad \alpha \quad \Downarrow_2 \\
\end{array}
\]

\[
\begin{array}{c}
\alpha \quad \Downarrow \\
\end{array}
\]

\textbf{Figure 4.7: Free Variable Work-list Algorithm}

The negation of this rule is found by applying the \text{boolean NOT} operator to the result.

4. Connectable
In Section 2.6.4 we defined the concept of connectable terms, and in Section 3 an explanation was given on why we need to check whether a term is connectable. Since the definition uses the \text{FreeVariable} condition, the FreeVariable class is extended with a static method to check if a variable is free in a term.
(med.getNode(Med.ADAPTOR_SOCKET)==v) &&
(FreeVariable.check(med.getNode(Med.LEFT_TERM))) &&
(FreeVariable.check(med.getNode(Med.RIGHTTERM)))

(exp.getNode(Exp.ADAPTOR_SOCKET)==v) &&
(FreeVariable.check(Exp.getNode(Exp.TERM)))

((cap.getNode(Cap.SOCKET)==v)||(cap.getNode(Cap.PLUG)==v) &&
(FreeVariable.check(Exp.getNode(Exp.TERM)))

4.1.2 Graph Building and Redirection

Once a match is identified between the head of a rule and a node in the term-
graph, an isomorphic copy of the nodes introduced by the body of the rule must
be created and added to the set of term-graph nodes, nodeSet. The set of nodes
that appear in the body and not in the head is called C. We create this by first
adding all nodes that appear in the rule to it, then removing all nodes reached
from the body root by using one of the iterators defined in Section 4.0.3.

For each node remaining in C, a new Node object of the same type is created
and added to the term-graph. If a Cut is created, the cutType attribute is set
accordingly. If the Node is a Connector type it can be identified as a fresh
variable. In this case, a fresh name is obtained from the variable universe. The
morphism set is updated with a mapping from the rule instance of the node and
the newly created node. The newly created node has no parents and so has a
reference count of zero.

The purpose of the redirection phase is to update the parents and children of
each node introduced by the body according to the rule. We iterate through
the set of newly created nodes, C initializing its successor function by using the
homomorphism. Listing 4.5 is applied to each node in C.
Node newGraphNode = morphism.lookup(introducedRuleNode);

for(int i=0;i<introducedRuleNode.getArity();i++){
    //get next child of rule node
    Node ruleNodeChild = introducedRuleNode.getNode(i);

    //retrieve corresponding child in newGraphNode
    Node graphNodeChild = morphism.lookup(ruleNodeChild);

    //increase reference count of the new child
    graphNodeChild.addReference();

    //set the child to be the successor of the node
    graphNode.setNode(i, graphNodeChild);
}

Listing 4.5: Graph Redirection

The final step is to redirect all nodes from the head of the match, to the newly created body. We cannot interleave node building with node redirection, since node redirection relies on there being a complete morphism for the rule.

If we were using an object oriented language with pointers such as C++, this step would be straightforward since we have direct access to the memory location we wish to update. However, since Java does not facilitate pointer manipulation an alternative method of redirection must be implemented. The initial goal of the project was to get a working system, and so the simplest method of redirection is adopted. To redirect from Node A to Node B, we iterate through the set of graph nodes. For each FunctionSymbol in the set, we replace any child node that is A, with B. Section 4.2.3 looks at alternative methods of implementing redirection in Java.

The node we redirected from is not physically removed from the set immediately, since object destruction is an expensive operation. If the redirection phase has worked correctly, its reference count should be zero, meaning the node has been marked for removal. We decrease the reference count of each child of the marked node, since essentially they have one less object pointing to them. If any of reference counts of the children become zero, we again have to propagate the result to the child nodes. This algorithm is easily implemented recursively, however an iterative solution is preferred since we will be able to make further optimisations on this later (see Section 4.2.1 for details of implementation).

4.1.3 Garbage Collection

The garbage collector uses the standard set iterator to remove nodes in the graph whose reference count is zero. On finding such a node, it is removed from the term-graph. Assuming no other Java objects point to this node, its memory
will be reclaimed when Java’s own garbage collector is invoked. We have no control on calling Java’s garbage collector, but we can control when to invoke our own. Two options are available:

1. Invoke Garbage Collector every \( n \) reduction steps
   A counter is incremented after executing each term. When the counter reaches the threshold, the garbage collector is invoked.

2. Invoke Garbage Collector when low memory
   After each execution step, we use Java’s \texttt{Runtime.getRuntime().freeMemory()} function to determine the amount of free memory in the system. If this value is less than a threshold, the garbage collector is invoked.

Our TGRS implements the second method.

### 4.2 Extending the TGRS

There are currently two modes of reduction for the TGRS. An interactive \emph{trace} mode which allows the user to select which cut to reduce from a list of all possible cuts in the term-graph and a \emph{match-by-rule} mode, which tries to match each rule with each node in the graph in a brute-force manner.

The two presumably confluent strategies for the calculus: ‘call-by-name’ and ‘call-by-value’ are to be implemented, as well as some common functional strategies such as outermost reduction and innermost reduction.

Tests were performed to determine the bottle-necks of the system and try to implement solutions to them. On running initial reductions of \( \chi \)-expressions, it was thought the bottleneck of the system was the GUI, detailed in Chapter 5. After a sufficiently large term-graph was built the rewriting system slowed down severely. Profiling the code, it was clear the majority of time was spent finding the next match to reduce. One possible solution that was investigated was the implementation of a data structure which allowed easy access to reducible nodes in the term-graph.

#### 4.2.1 Cut-Pointer Stack

Initially, matching involved a breadth-first search of the graph for the first Cut. Following this, each rule in the rule list would attempt to be matched against the cut. On finding a match, the cut was executed, and the process restarted. If the cut could not be matched against a rule, the next cut in the graph was found.

When the term-graph is small, this is an apt implementation for the outer-most reduction strategy. However, as the size of the term-graph increases, it is evi-
dent that this soon becomes the bottleneck of the system. The implementation
also offers poor support for inner-most reduction, since to find the inner-most
reducible cut, an entire parse of the graph would have to be performed.

*The Cut node is the only reducible expression in the calculus.* In any valid χ-
expression, the minimum proportion of nodes to cut nodes is 4:1. In our initial
implementation, this means at least 75% of the time traversing nodes is wasted.

Keeping a separate list of cuts is an optimisation. In this way, we have direct
‘pointers’ to reducible points in the graph. Storing meta-information, such as the
*level* of the cut in the term-graph, will be useful when implementing functional
strategies. Figure 4.8 gives an overview of the datastructure.

A level is a horizontal row of nodes in the term-graph. Level zero marks the
top-most or outermost level in the graph (the root node) and the level index
increases as we move down the graph. For a reliable and correct implementation
of functional strategies, it is essential the level information is kept valid during
all stages of the reduction.

**Implementation Details**

The *Cut* node is extended with a private field, *level*, which records the level
relative to the root of the term-graph. After the user enters an expression into
the lexer/parser, an initial pass of the graph sets the *level* attribute of each *Cut*
node.

Each stored rule for the calculus is also extended to record the levels of each cut
in the body of the rule *relative to the outermost node*. (See Figure 4.9). This

![Figure 4.8: Cut-Pointer Stack with Level Information](image-url)
information is used to set levels of new cuts introduced into the expression by the rule body.

The building phase of the reduction engine is modified to incorporate the level information. When an isomorphic copy of a cut is made, it has its level attribute set to the index of the cut selected for reduction, plus the relative index stored in the rule body. This step maintains the validity of the level information.

Finally, we associate with each term-graph, a `CutStack` object. The structure is defined as in Figure 5.7 and Listing 4.6.

```java
public class CutStack {
    private int highestLevel;
    private int count;
    private Object[] stack;
}

public List getOuterMostCut(int start);
public List getInnerMostCut(int start);
public void add(Cut c);
public void remove(Cut c);
public void decreaseCount();
public int getCutIndex(Cut c);
public Iterator iterator();
```

Listing 4.6: Structure of the `CutStack` class

This is an array of pointers to lists, where the index of the array is the tree-level and the list pointed to by the index is the set of Cuts at that level. If there are no cuts at a particular level, the index points to `null`. Two additional fields are stored with the array.

1. Highest Cut Level
   When implementing innermost reduction, it is useful to know the highest level of the cut in the graph. This way, we can immediately return the
list of cuts at a particular level without having to cycle through the entire structure. When adding a Cut to the graph, its level is checked against the current highest-level. If it is greater, the field is updated. If we are removing the last cut from the list of cuts at the highest level, then we update the field to point to the next highest level. This is done via a loop from the old highest level to an entry in the stack that is not null.

2. Number of Cuts in the Graph

On initialization of the cut-stack structure, the array size is set to an initial capacity, e.g. 256. This figure represents the maximum depth of the term graph. If more levels are required, a new array of larger capacity is created and the structure copied across.

We will later detail the implementation of an iterator for the cut-stack which returns the inner or outermost nodes in the graph. Rather than cycling through the stack array 0→255, we keep a count of how many cuts are in the stack. As we return each cut to the user via the iterator’s next() method a local counter is incremented. When this local counter is equal to count we know we have returned all the cuts. This field is updated in the add and remove methods of the class.

The add and remove methods maintain the List structures at each index of the stack. On adding a cut to a level which contains no other cuts (i.e. stack[level] == null), a new List is created for the level and the cut inserted into the List. The number of cuts in the stack is increased. Removing the last cut from a List sets the List entry for the level to null and decreases the count of cuts in the stack. The highest level field is updated if this was the highest level.
Functional Strategies

Two public methods are exposed to cater for functional strategies.

1. **public List getOuterMostCut(int start);**
   Cycles from the index, `start`, downwards (increasing level number) until it encounters a List of cuts. These will be the outermost cuts relative to the starting position.

2. **public List getInnerMostCut(int start);**
   This function is similar to the outermost cut, except it cycles upwards (decreasing level number) from the starting point. This List of cuts will be the inner most cuts relative to the starting position.

Shifting Levels

When a cut is executed, it may cause the level information to become corrupted. Figure 4.11 shows an example of this. The inactive cut is shifted up a level. Additionally, if \( \bot_2 \) and \( \bot_3 \) contain cuts, these are also shifted up a level. The implemented solution is to include shifting information in the head of the rule. This is the number of levels each cut increases or decreases by when executed. The information is also associated with each \( \bot \) in the rule head and is generated automatically when a rule-graph is saved.

After an execution, any cut nodes in the head of the rule whose level attribute is non-zero modifies the corresponding cut in the term-graph. In the case of \( \bot \) terms, this information is propagated to the set of nodes in its reflexive transitive closure. The cut-stack is relatively small compared with the entire graph, so the operation is negligible.

![Figure 4.11: The effect of shifting levels due to gc-L](image)

level

<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Shared Cuts

One final point to consider is if a cut is shared and so appears in more than one level of the term-graph. A convention is arbitrarily chosen to set its level to the innermost. It is left for further work to investigate whether this scheme is preferable over choosing an outermost convention.

4.2.2 Prioritizing Rules

The set of RuleGraph objects for the calculus are stored in the singleton instance of the RuleList object. With outermost or innermost reduction, a Cut is selected from the top or bottom of the term-graph. Each RuleGraph in the RuleList structure is then cycled through in order of index for matches. If more than one match is found, the current implementation allows the user to choose which rule to apply.

Call-by-name and call-by-value can be implemented if a strict first-match policy is adopted. This reduces the problem to a List ordering implementation. For call-by-value the (act-L) rule is placed above the (act-R) rule in the RuleList, and for call-by-name the (act-R) rule is placed above the (act-L) rule. In the scenario that a cut could be activated in either way, the priority system ensures the first rule encountered is applied.

To add flexibility to the system, a scheme is introduced to allow custom orderings of rules. Two new structures are implemented: Strategy and StrategyList. A strategy is a List which holds the names of the rules in the desired order. Since more than one strategy may be desired, a strategy-list holds the different strategies in the system. When a new strategy is selected, the system RuleList instance is reordered accordingly. In Prompt-mode the tool ignores the priority system and prompts all matches applicable to the cut.

The convention chosen is to work on active cuts first. The CutStack class is modified to keep two counters for cuts: one for flagged cuts in the term, and the other for non-flagged cuts. The cut-stack iterator is modified to return flagged cuts before non-flagged cuts in the term-graph.

4.2.3 Efficient Redirection of Nodes

In Section 4.1.2 a quick, but expensive algorithm for the redirection mechanism was implemented. A discussion follows of implementation alternatives for redirection in an OO-language without pointer manipulation.
Back ‘Pointers’

Each Node is extended with a List containing references to its parents. The list is maintained when nodes are added, removed or redirected in the system. During the redirection stage, the node in the term-graph that corresponds to the head of the rule updates each of its parents to point to the node corresponding to the node body. The outline of the procedure is given in Listing 4.7.

```java
public void redirect(Node from, Node to)
{
    List parents = from.getParents();
    for (int i=0;i<parents.size();i++)
    {
        Node p = (Node)parents.get(i);
        for (int j=0;j<p.getArity();j++)
        {
            if (p.getNode(j)==from)
                p.setNode(j,to);
        }
    }
}
```

Listing 4.7: Redirection using back-pointers

Object Wrappers

Each Node object in the system is encapsulated by a NodeWrapper object. The successors of each node are redirected to point to the NodeWrappers rather than the Node objects themselves. Now the redirection phase consists of updating the node pointed to by the node wrapper. Since it is the node-wrapper that is shared, this update is reflected globally with respect to the term-graph.

This approach amounts to emulating a pointer manipulation mechanism in Java, but with extra costs. The size of the node set for each term-graph is doubled. Additionally, a level of indirection is introduced.

Our implementation preferred the back-pointer elimination since it created less objects, and less time would be spent iterating through intermediary wrapper nodes.

4.2.4 Renaming Rules

The cut of a pure-term with a capsule bound to that cut amounts to renaming all free occurrences of the binding connector in pure term. i.e.

\[ (\text{ren-L}) : M \overline{\delta} \uparrow \overline{\exists} (z.a) \rightarrow M[\alpha/\delta] \text{ if } M \text{ is pure} \]
\[ (\text{ren-R}) : (z.a)\overline{\alpha} \uparrow \overline{x}M \rightarrow M[z/x] \text{ if } M \text{ is pure} \]
The effect of this is actually $\alpha$-conversion, since we must rename all occurrences of the binding connector throughout the whole term. If we did not, taking the case of ren-$\lambda$, we would have terms referring to $M$ with binder $\delta$. This is clearly wrong since all occurrences of $\delta$ in $M$ have been renamed to $\alpha$.

To implement this, it should be a simple case of pointer redirection. However, since Java does not allow for pointer manipulation, we must iterate through the set of nodes and redirect all function-symbols who’s children are $\delta$ to $\alpha$.

### 4.2.5 Block Flattening

The block-flattening rule of term-graph rewriting systems was explained in Section 2.7.2. Essentially, it renames bound variables to avoid name clashes (this is essentially $\alpha$-conversion). It is interesting to see that the term-graph representation of expressions do not need $\alpha$ conversion. The reason for this is the connectors are identified as objects rather than the String values they encapsulate.

We update our `toString()` and `toUnicode()` methods to rename these name clashes. This is done by propagating a List of binders who must be renamed. The renaming consists of adding an index to the string representation of the connector.

If we consider the example of $\hat{y}(\hat{y}(y,\beta)\beta \cdot \alpha)\beta \cdot \gamma$, then the List of binders that gets propagated to sub-terms is $[y,\beta]$. On arrival of the inner export term, the flattening procedure will see one $y$ socket and one $\beta$ binder already exists in outer terms. This causes the inner term representations to be suffixed with the index `one`. Had there been two occurrences of $y$ in the set, the index would be two.

Our example gets renamed to: $\hat{y}(\hat{y}_1(y_1,\beta_1)\beta_1 \cdot \alpha)\beta \cdot \gamma$
4.3 Interpreting $\lambda x$

The definition for interpreting $\lambda x$ terms into $\chi$, given in [vBL03] is:

$$
\begin{align*}
\llbracket x & \rrbracket_\alpha^\chi = (x, \alpha) \\
\llbracket \lambda x.M & \rrbracket_\alpha^\chi = \llbracket M \rrbracket_{\alpha\beta}^\gamma \cdot \alpha \\
\llbracket M \cdot N & \rrbracket_\alpha^\chi = \llbracket M \rrbracket_{\alpha\beta}^\gamma \cdot \llbracket N \rrbracket_{\beta\gamma}^\delta \cdot [x \cdot g(y, \alpha)] \\
\llbracket M \cdot x = N & \rrbracket_\alpha^\chi = \llbracket N \rrbracket_{\beta\gamma}^\delta \cdot \llbracket M \rrbracket_{\alpha\beta}^\eta \cdot [x \cdot g(y, \alpha)]
\end{align*}
$$

The lambda-terms are interpreted over output channels (plugs). Since the lambda-calculus contains no concept of plugs, these should be automatically generated by the calculus, although it should also be possible to specify the output channel for the entire term, if these terms are to be integrated with $\chi$-terms. The lexical analyser and parser must be extended to recognise lambda expressions. Once the parser can do this the generation of $\chi$-terms is straightforward.

In the $\lambda$-calculus, function application is left associative and function abstraction is right associative. Additionally, the parser should recognise sugared abstractions (e.g. $\lambda xy.x = \lambda x.(\lambda y.x)$).

The lexical analyser is extended with the following tokens: $\{\lambda, =, [\cdot], \cdot\}$. The SOCKET and PLUG terms are terminal-symbols defined by the lexical analyser as in Section ???. The standard BNF definition is rewritten in order to avoid conflicts, and is as follows:

\[
\begin{align*}
\text{\textbf{\lambda-term}} & := \text{app} \mid \text{abs} \mid \text{atom} \mid \text{xterm} \mid (\text{\textbf{\lambda-term}}) \\
\text{app} & := \lambda\text{-term} \mid \text{\lambda-term} \mid \text{app} \mid \text{\lambda-term} \\
\text{abs} & := \lambda \text{abs\_body} \\
\text{abs\_body} & := \text{SOCKET} \text{abs\_body} \mid . \text{\lambda-term} \\
\text{atom} & := \text{SOCKET} \\
\text{xterm} & := \lambda\text{-term} \langle \text{SOCKET} = \lambda\text{-term} \rangle
\end{align*}
\]

The definition of $\text{term}$ is then extended to:

\[
\text{term} := \chi\text{-term} \mid [[\lambda\text{-term}]] \mid [[\lambda\text{-term} \cdot \text{PLUG}]]
\]

JavaCUP is a bottom-up parser, meaning the innermost terms will be interpreted first. These will always be atoms which have the same format as socket tokens. However, there is a subtlety in the generation of $\chi$-terms from $\lambda$-terms. The output-plug of inner terms are reused in the outer terms.

To cater for this, the parser is extended with an internal stack ($\text{PlugStack}$), which is used to keep record of the inner-term plugs that are used by outer-
terms. One can think about the terms to be interpreted, i.e. $[[\cdot]]_\alpha$, as operations on the plug-stack.

Referring to the interpretation of the $\lambda$-calculus, the constructs on the left-hand side refers to the plug pushed on to the stack, and constructs on the right-hand side refer to plugs popped from the stack. For example, the function application interpretation rule reads $\gamma$ and $\beta$ from the plug-stack and outputs $\alpha$ on to the plug-stack. Consider the interpretation of $(\lambda x.x)y$ shown below.

<table>
<thead>
<tr>
<th>ParserStack (right is top)</th>
<th>Token PlugStack Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>shift</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>shift</td>
</tr>
<tr>
<td>$x$</td>
<td>shift</td>
</tr>
<tr>
<td>$(\lambda x)$</td>
<td>shift</td>
</tr>
<tr>
<td>$x$</td>
<td>shift</td>
</tr>
<tr>
<td>$(\lambda x.x)$</td>
<td>)</td>
</tr>
<tr>
<td>$(\lambda x$.{.x.$\delta$})</td>
<td>) $\delta$</td>
</tr>
<tr>
<td>$(\lambda x$.{.x.$\delta$})</td>
<td>) $\delta$</td>
</tr>
<tr>
<td>$({z$.$\langle x.\delta \cdot \beta \rangle$)</td>
<td>) $\beta$</td>
</tr>
<tr>
<td>$({z$.$\langle x.\delta \cdot \beta \rangle$)</td>
<td>) $\beta$</td>
</tr>
<tr>
<td>${z$.$\langle x.\delta \cdot \beta \rangle$ y</td>
<td>$\beta$</td>
</tr>
<tr>
<td>${z$.$\langle x.\delta \cdot \beta \rangle$ y</td>
<td>$\beta$</td>
</tr>
<tr>
<td>${z$.$\langle x.\delta \cdot \beta \rangle${.y.$\gamma$}</td>
<td>$\beta, \gamma$</td>
</tr>
<tr>
<td>$({z$.$\langle x.\delta \cdot \beta \rangle${.y.$\gamma$}</td>
<td>$\beta, \gamma$</td>
</tr>
<tr>
<td>$({z$.$\langle x.\delta \cdot \beta \rangle${.y.$\gamma$}</td>
<td>$\beta, \gamma$</td>
</tr>
<tr>
<td>$({z$.$\langle x.\delta \cdot \beta \rangle${.y.$\gamma$}</td>
<td>$\beta, \gamma$</td>
</tr>
<tr>
<td>$({z$.$\langle x.\delta \cdot \beta \rangle${.y.$\gamma$}</td>
<td>$\beta, \gamma$</td>
</tr>
<tr>
<td>$({z$.$\langle x.\delta \cdot \beta \rangle${.y.$\gamma$}</td>
<td>$\beta, \gamma$</td>
</tr>
</tbody>
</table>

The highlighted plugs match plugs from inner terms with their use in the outer terms.
4.3.1 Let Expressions

The user may wish to input a term which already exhibits sharing of terms. The lexical analyser is extended with the following tokens:

\[
\begin{align*}
\text{NAME} & = [A-Z][A-Za-z0-9]* \\
\text{IN} & = "in" \\
\text{LET} & = "let" \\
\text{EQUALS} & = "="
\end{align*}
\]

Using our previous definition of ‘term’ we extend the parser with the following BNF for valid input terms.

\[
\text{InputTerm ::= term | LET NAME EQUALS InputTerm IN term}
\]

All references to \text{NAME} in ‘term’ are shared and so point to the same node.
Chapter 5

GUI Design

5.1 Special Characters

The alphabet of the calculus involves the following special character sets:

- Cut Symbols: $\hat{t}$, $\check{r}$, $\mathring{x}$
- Bound Sockets: $\hat{a}\hat{b}\hat{c}...\hat{x}\hat{y}\hat{z}$
- Plugs: $\alpha\beta\chi...\xi\nu\zeta$
- Bound Plugs: $\acute{\alpha}\acute{\beta}\hat{\chi}...\hat{\xi}\hat{\nu}\hat{\zeta}$

Various methods exist for displaying these special characters on-screen in Java. Three approaches were attempted and implemented.

Image Mapping

Each glyph (visualisation of the special character) is drawn to an image.

\[
\hat{\alpha} | \check{\beta} | \mathring{\chi} | \mathring{\delta} | \check{\epsilon} | \check{\phi} | \check{\gamma} | \check{\eta} | \check{\iota} | \check{\rho} | \check{\sigma} | \check{\tau} | \check{\theta} | \check{\lambda} | \check{\nu} | \check{\pi} | \check{\rho} | \check{\sigma} | \check{\tau} | \check{\phi} | \check{\omega} | \check{\xi} | \check{\psi} | \check{\varphi} | \check{\chi} | \check{\psi} | \check{\varphi} | \check{\chi} | \check{\psi} | \check{\varphi} | \check{\chi} | \check{\psi} | \check{\varphi} | \check{\chi} | \check{\psi} | \check{\varphi}
\]

**Figure 5.1:** *Image-map of accented Greek characters*

This is read into memory as an image. The image-map is created such that each character is equally spaced horizontally and vertically, and each character is of the same proportions. A particular character can be obtained from its offsets (determined by its index) and bounding-box size.
After implementation, the drawbacks for this solution were evident. Rasterized images do not scale well, and the small size of the glyphs meant some were obscured beyond recognition after scaling.

Creating a True-Type Font

Java 1.3+ includes support for loading custom True-Type fonts. A font creation package (e.g. FontLab) cab glyphs can be created in vector format

![Figure 5.2: FontLab design of Greek Delta](image)

The special characters were designed by extending a standard template that included vectors for Greek and Roman characters. This implementation method worked well and scaling was not a problem, since the glyphs were stored in vector format. An extra-cost of using this implementation is the requirement that the Java Applet be signed. Java's font-loader writes a temporary file to disk when loading the font. Writes to disk are outside the sandbox security model for Java Applets.

Unicode Fonts

The Unicode standard provides a unique number for every character no matter what the platform, no matter what the program, no matter what the language. Greek characters are included in the Unicode range [0x03b1 - 0x03c9]. The standard defines a number of different types of characters including base characters and combining characters. Unicode 4.0, Chapter 3 section 6 [page 70] defines them as:

- D13 Base Character: A character that does not graphically combine with preceding characters, and that is neither a control nor a format character.
- D14 Combining character: A character that graphically combines with a preceding base character. The combining character is said to apply to that base character.

This allows combination of the Greek base characters (α–ζ) with the ‘combining circumflex accent’ character or ‘hat’ to create the characters (ā – ĺ). In Java this is represented as shown in Listing 5.1.
Using this method allows for representation of any of the special characters, without the need to step up the security model. It also promotes the Unicode standard and provides a more compliant implementation. For the cut-left and cut-right symbols (\(\checkmark\), \(\times\)), the standard dagger symbol can be rotated using Java2D’s `AffineTransform` method on the original glyph.

### 5.2 System Design

When researching the state-of-the-art of reduction tools, it was found that the majority were based on text commands. The project is based on term-graph rewriting systems, and so should contain graphical elements. The motivation when designing the interface was concentrated on ease of use through intuition. This primarily involved heavy use of drag-and-drop mechanisms, although it also catered for a natural keyboard interface.

Java was chosen as the implementation language since a cross-platform tool was required. The GUI is written entirely using Java’s Swing API, which is also 100% Java. Since there are no native calls, the GUI should be compatible with any platform that provides a JVM implementation.

The main components of the GUI, outlined in the following sections, are:

- Virtual Desktop
- Document
- Diags
- Graphs
- Rule Editor
- Rule List
- Point-and-click Interface
5.2.1 Virtual Desktop

It is foreseeable that the user will want to work on different reductions at any one time. The user may wish to compare results of different terms by perhaps viewing different representations of them or walking through a reduction.

At the heart of the TGRS GUI is a virtual-desktop environment that caters for such a multiple-document interface. Documents are created in this environment and can interact with each other when required. A toolbar and menu-bar provide easy access to commonly used functions.

The toolbar contains a text input panel that accepts a $\chi$-term in ASCII form. This is the main interface for inputting terms into the system. However, a graph-editor is also provided since the user may wish to evaluate a term which has a specific sharing configuration.

The menu-bar contains standard entries found in applications such as exporting a reduction to $\LaTeX$, importing user-defined strategies and rules, and accessing configuration dialogs.

5.2.2 $\chi$-Document

A document is created for each term-graph the user wishes to evaluate. Its purpose is to track the history of reduction sequence, together with the rules applied at each stage. The document-frame itself consists of a table object with three visible column. The first records the step number in the execution, the second is a Unicode representation of the term and the third records the rule that was applied at the step. As each rule is applied, a new row is added to the table.

![Figure 5.3: ChiDocument Frame](image)

Three interaction buttons are also associated with the document. A start button begins an iterative application of rules to the term-graph according to the selected strategy. The process stops when the term is in normal form. If the term has no normal form, and is an infinite reduction, a stop button is provided to control execution of the term.

55
The third button invokes a trace mode for reduction. The user is permitted to select which cut to execute from a list of all applicable rules to the term at each stage of the reduction.

5.2.3 \( \chi \)-Diagrams

In Section 2.6.3 we looked at the diagrammatic representation for the calculus. The user may wish to view this representation for any of the expressions in the \( \chi \)-document. A new type of document is created to display the diagram. For larger diagrams which do not fit in a frame, two methods of scrolling are implemented: click-and-drag and use of the scroll bars.

A canvas is defined onto which the diagram will be drawn. The component is implemented by subclassing a `JComponent`, which is the base-class for all Swing components. Five methods are defined to draw each term of the calculus, including the bottom term. Spacing is important for aesthetic display of the diagrams. The inner-terms are drawn first since we need to know their width and height before drawing the enclosing outer terms.

5.2.4 \( \chi \)-Graphs

The user would like the option to view the term-graph which has been generated for a particular reduction step. We already have the capabilities of generating a graph from a \( \chi \)-term in ASCII format. However, this graph may not be the same graph as one that would be generated during the reduction sequence. An example is shown in Figure 5.5

The first graph in the figure is one that would be generated by the parser. The second is a possible graph that could be generated from a user, or perhaps a reduction. The fact that these graphs differ rules out the possibility of dynamically generating the graph for display from the stored ASCII string. Two
possible ways in solving this are presented.

1. Cloning and Storing the Graph

We cannot simply store a *pointer* to the graph at each level of the table, since the term graph being evaluated is a volatile/mutable object. In the reduction, a string representation of the graph, and not the graph itself, is created, then stored. A clone or ‘deep copy’ would have to be made of the state of the Graph object, and this is stored at each level. This involves walking the tree from the root node, (or an iteration of the *nodeset*) and cloning nodes with a reference count more than zero. A *List* of Nodes in the graph could then be created and these stored at each row of the document.

The drawbacks of this implementation are points on efficiency. Object cloning and storing the entire structure of the tree at each reduction step is memory-inefficient and also time consuming.

2. Storing a non-volatile copy during Garbage Collection

The garbage collector can be set to run after every *n* reduction steps. Immediately after garbage collection, a deep copy of the object can be made and stored. When the user wishes to view a graph for a particular expression, the system rolls back to the last garbage collection step and sequentially applies the rules to the graph until the expression to be viewed is reached.

In order to do this, a pointer to the cut that was reduced in the expression should be stored at each row in the model. Although it is valid to do this, it would mean each cut stored would *never* be removed from memory (while the document is open), since there would always be an active reference to it even after the TGRS garbage collector has run. It would be better to store the *index* of the cut into the graph at each level. This index can be calculated from the row and column of the cut in the cut-stack. On reconstruction of the graph, the index of the cut can be retrieved from the cut-stack and the rule matched with its stored name.

The process of applying the successive rules from the garbage collection step
to the expression should be significantly faster than that when the reduction-
process was run initially. The reason for this is the GUI slows down the system
severely. When recording the row information in a document for the current
graph, an entire pass must be made by the `toUnicode()` method. A new row
must be created in the `JTable` and the table repainted. Using the second method
presented, it is known exactly which rule to apply and where, so the rule search-
ing/matching phase is skipped.

It is foreseeable, that the user will wish to see graphs of the succeeding steps
from the chosen expression. An optimisation is to store a clone of the graph at
the selected reduction step also.

**dot and Grappa**

The graph generated at a particular step needs to be laid out on-screen for
display. This is a hard problem, since the term-graphs themselves are not planar.
In other words, the term-graphs cannot be drawn in two-dimensions unless the
edges are allowed to intersect (crossover) in some manner.

There exists an open-source community called GraphViz whose research area
is graph drawing. `dot` is a preprocessor for drawing directed graphs. It reads
attributed graph files and outputs their graphical representation. In particular
its algorithms can be configured to generate a graph with the least amount of
cross-overs.

Grappa is a Java graph drawing package that provides methods for building,
manipulating and displaying graphs. The input is a graph written as text in the
`dot` language. The class of graph that will be used in this project is the `digraph
or ‘directed graph’. Nodes can be configured to different sizes and fonts.

Using a breadth first traversal of the graph, a `dot`-language representation of
the graph is created, and this passed to the Grappa interface. The graph is
generated and shown on screen. Figure 5.6 and accompanying Listing 5.2 display
an example of a generated term-graph.
digraph g{
    cut [label="cut"]; 
    cap2 [label="cap"]; 
    cap1 [label="cap"]; 
    a [label="a", color=lightgrey, style=filled]; 
    x [label="x", color=lightgrey, style=filled]; 
    b [label="b", color=lightgrey, style=filled]; 
    y [label="y", color=lightgrey, style=filled]; 
    "cap1" -> "x"; 
    "cap1" -> "a"; 
    "cap2" -> "y"; 
    "cap2" -> "b"; 
    "cut" -> "cap1"; 
    "cut" -> "a"; 
    "cut" -> "y"; 
    "cut" -> "cap2"; 
}

Listing 5.2: dot specification for Figure 5.6

Figure 5.6: ChiGraph Frame
5.2.5 Rule-Editor

Two forms of creating rules are provided. The first, covered in Section 5.3 allows a user to edit the configuration files for the TGRS and type in the rule head and body by hand. The conditions for the rules can also be set up in this way.

A second method of rule-creation is also provided. The user can add new rules graphically using a graph-editor tool. This is also useful as a learning tool, since it allows the user to see how rule-graphs are built. Three components to the rule-editor exist.

![Figure 5.7: The RuleEditor Canvas](image)

The GraphCanvas object extends a JPanel component, and defines a MouseListener and MouseMotionListener. A toolbar is provided to create different types of nodes on the canvas, or to enter different modes of operation such as delete and un-sharing mode. Since complicated rule graphs can get confusing, a highlighting mechanism is provided which identifies successor nodes of the selected parent.

The properties panel has two components. The first allows the user to change the names associated with the nodes, since connector names are automatically generated, and the default label for bottom terms is the symbol ↓.

The second component of the properties panel is a small input-box which allows the user to specify side-conditions for the rule. A lexer/parser is implemented to constrain the input to valid terms. The grammar for the conditions is defined as:
Condition ::= \(fv\ (\text{NAMED\_TERM}, \text{CONNECTOR})\)  
| \(pure\ (\text{NAMED\_TERM})\)  
| \(equals\ (\text{PLUG}, \text{PLUG})\)  
| \(equals\ (\text{SOCKET}, \text{SOCKET})\)  
| \(connectable\ (\text{NAMED\_TERM}, \text{CONNECTOR})\)  
| \(not\ (\text{Condition})\)

To use the \(fv\), \(pure\) or \(connectable\) conditions, a valid named term must be specified. This means renaming any bottom term, \(\bot\) to conform to the lexical token format for \text{NAMED\_TERM} given in Section 4.0.1.

Checking is done to ensure the named terms and connectors entered into the side condition input-box do exist in the rule graph.

**Sharing and Un-sharing**

![Diagram of RuleEditor Nodes for Cut, Med, Exp, Cap](image)

**Figure 5.8:** RuleEditor Nodes for Cut, Med, Exp, Cap

We introduce temporary function placeholder nodes \(L, R\) and \(T\) that are created when a node is first placed on the canvas (Figure 5.8). Their purpose is to create an easy sharing mechanism for sub-terms. The method of sharing in the rule editor is as follows:

1. Hold the left mouse button down on the function-placeholder or connector.
2. Drag the mouse to the Node with which it is to be shared.
3. Release the mouse button.

The result is the old connector or function-placeholder node disappears from the graph canvas, and an edge is drawn from the parent to the newly shared node. If the user changes his/her mind about a sharing, an ‘Unshare’ button
is provided. To use it, first left-click the parent of the node to be unshared, then left-click the mouse on the node to be unshared. A new unshared node (connector or function-placeholder) is automatically created. Newly created connectors obtain their name from the variable universe.

### 5.2.6 Rule-Strategy Dialog

This window (Figure 5.9) has two purposes. It allows the user to navigate through the rules of the calculus and view their graphical representation. It also serves as the rule priority configuration dialog. The rules stored in the system are ordered as specified by the current loaded ordering. This dialog can be used to define new orderings of rules using the up and down operations on the table. It is also the place where the user selects which strategy to use when reducing: innermost, outermost or first rule reduction.

![Figure 5.9: The RuleList Dialog](image)

### 5.2.7 Point-and-Click Reduction

Easy interaction is a goal in the design of the system. When evaluating larger $\chi$-terms, listing the rules that are applicable to each expression is often confusing for the user. We already know the executable places in a term are the Cut nodes. A more direct method of reducing terms would be to allow the user to ‘point and click’ on the cut that they wish to reduce. If more than one rule is applicable to the Cut, a prompt box should be presented, from which the user can select the rule he/she wishes to apply.

So far, we have described mechanisms for converting $\chi$-term-graphs to ASCII Strings, to Unicode and to $\LaTeX$. The key requirement for adding a point-and-click mechanism to $\chi$-terms is a correspondence between cuts in the string...
and the physical Cut nodes in the term-graph. This way, clicking on a Cut will identify the node to be reduced in the graph.

The JLabel Swing class is an object that can display text or images. This is ideal for representing terms on screen since the user-interface is implemented in Swing also. We extend the Node class with the abstract method toPointAndClick(List panels) which generates the representation of a term-graph using JLabel objects. The method is invoked on the root node of a term-graph. This adds items to the panels list in the order specified by the successor function for the node, recursing where necessary.

In order to associate an instance of a Cut node with a JLabel, we extend the JLabel class with a private field and necessary accessor functions. (See Listing 5.3). When a call is made to a cut-node, a CutLabel class is created and associated with the instance of the cut. Additionally, a MouseListener is added to the CutLabel, making it ‘clickable’. Different colours are used to represent flagged and non-flagged cuts.

```java
public class CutLabel extends JLabel{
    private Cut pointer;
    CutLabel(String s){ super(s); }
    public void setCut(Cut c){ pointer = c; }
    public Cut getCut(){ return pointer; }
}
```

Listing 5.3: Associating Cut instances with JLabels

When creating the CutLabel, a List of matches with rules are also created. The tool-tip for the CutLabel is set to the list of applicable matches, allowing the user to hover the mouse over a cut and view which rules are applicable.

![Figure 5.10: Point-and-click Reduction Frame](image)

It is possible that the string of a χ-expression will be longer than the screen width. In this case, scroll-buttons are placed to the left and right of the point-and-click panel. A small threading mechanism is used to implement a two-speed scrollable interface. Hovering the mouse over the button causes the term to shift left or right. Pressing the scroll-button causes the term to shift faster.

In continuing with the drag-and-drop theme of the system, the user also has the ability scroll via dragging the long χ-term.
5.2.8 Exporting to \LaTeX

At any time in the reduction of an expression, the user may wish to export the document to \LaTeX. The \texttt{Node} class is extend with a \texttt{toLaTeX()} method which generates a \LaTeX math-array of the document. The implementation is very similar to the toUnicode, toString and toPointAndClick methods, except the output is formatted differently. Although \LaTeX already has good support for displaying Greek and special characters, macros\footnote{Thanks to Steffen van Bakel} are used to define the output. The \texttt{qsymbols} package is also required.

The default output array generated is of the form shown in Listing 5.4, although the user has the option to modify the output generated.

\begin{verbatim}
\begin{array}{lll}
1. & \text{Cut}\{\text{Cap}\{x\}\{\{a\}\}\{y\}\{\text{Cap}\{y\}\{\{b\}\}\} & \\arrow& (\text{var})\llbracket1mm\rrbracket
2. & \text{Cap}\{x\}\{\{b\}\} & \&\\llbracket1mm\rrbracket
\end{array}
\end{verbatim}

\textbf{Listing 5.4: Generated \LaTeX output}

5.3 Configuration Files

The rules for the calculus are not hard-coded into the system. To increase maintainability and flexibility of the system, they are listed in XML configuration files which are loaded when the tool is initialized.

Since no manipulation of the XML is required, a SAX XML parser is used. NanoXML\footnote{by Marc De Scheemaecker, http://nanoxml.sourceforge.net/}, a non-validating parser for Java distributed under zlib/libpng license, is ideal for reading and writing XML due to its small size (6k).

The root of the XML-document is the \texttt{(RuleList)} node, which contains a set of \texttt{(Rule)} nodes as children. Each rule node describes the structure and conditions of the rule. The rule structure has a head and a body element, whose values (in string format) represent rooted term-graphs. The \texttt{(Conditions)} element contains \texttt{(Condition)} elements for the node whose value specify \texttt{RuleConditions} in the format specified in Section 5.2.5.

A \texttt{RuleImporter} class reads the head and body of the rule structure, and generates a \texttt{RuleGraph} object. It automatically shares nodes that appear in the right hand side with nodes that appear in the left hand side. This is done by structurally matching sub-terms in the body, with sub-terms in the head. An example of the XML for the exp-rule is shown in Listing 5.5.
Listing 5.5: Rule-List XML Configuration File

Listing 5.6: Strategy-List XML Configuration File

The second configuration file used in the system is for storing user-preferences on the ordering of rules, which can implement different strategies. This is simply a sequence of rule names, where each rule in the order is defined in the Rule-List XML configuration file. An example of the format of the file is given in Listing 5.6.

5.4 Deployment

The initial design treated the application as an applet embedded in a HTML page. During project development, testing was done to see which platforms supported the applet. The tool functioned in different ways depending on which JVM implementation was used. Also, it became clear that Microsoft were to discontinue support for their flavor of the JVM, and so Windows users would have to run the Sun implementation. As the program grew in size the applet took increasingly longer times to launch. While the facility also existed for the user to download the code for themselves and run the tool as a standard Java program, a more elegant solution was sought out.

Java WebStart is a mechanism for delivering Java-based applications through a standard web-server. Applications launched in this way are downloaded to the local machine and stored in a cache; options for creating desktop icons for the application are possible. The application does not need to be downloaded every time the user wishes to run it. It checks a file on the web-server to see if an updated version is available. If there is a newer version, then it is downloaded and the cache updated. Otherwise the application is run from the cache.
Configuring the web-page to support Java WebStart required writing an XML-configuration file specifying the security requirements of the application, the required JAR files and entry-point of the application.
Chapter 6

Evaluation

Evaluation and testing was ongoing throughout the project. During the implementation phase, the tool was refactored several times, and due to the short project-time available, the continuous JUnit-testing approach (as intended in the specification) was abandoned. This final chapter looks at points that were missed or delayed for discussion during the implementation stages.

6.1 Input Convention

The validity of the term-graphs produced will only ever be good as the validity of the terms input in to the system. While testing, the system accepted the following input without error: $(\langle x, \alpha \rangle \mu \uparrow \tilde{z}(y, \beta)) \tilde{a} \uparrow \tilde{g}(\langle x, \alpha \rangle \mu \uparrow \tilde{z}(y, \beta))$. A reduction followed and the term was reducible as shown in Figure 6.1. The second step is generated by the sequence $\{act-l1, dl1\}$ or $\{act-r1, dr1\}$ on the innermost cut. Examining it more closely, we can see it has connectors both free and bound in the same term. This is a result of the input term that was entered in to the TGRS. To avoid this anomaly, it should be made clear that the input term entered was invalid either by having the system check the term for errors or stating the variable convention to the user. An automated solution to the problem could be to automatically rename to avoid such variable clashes.

6.2 Suitability of Java

The purpose of this project was to deliver a tool that could be downloaded by academics (and others interested in the field) to learn about $\chi$. Presently, Java is the only widely available cross-platform programming language available. This fact alone means the tool is available to the widest audience given current methods of implementation. The drawbacks of using Java to implement the
Figure 6.1: Execution of \((\langle x, \alpha \rangle \mu \uparrow \tilde{z}(y, \beta))\hat{\alpha} \uparrow \hat{y}(\langle x, \alpha \rangle \mu \uparrow \tilde{z}(y, \beta))\)
TGRS presented themselves throughout the project. The most costly barrier was the need to implement a back-pointer mechanism for redirection. In the renaming rule for the calculus, the problem presented itself again.

It is also questionable whether the garbage collector implemented in our TGRS functions as intended. If we had set the nodes to \texttt{null} as soon as the reference count became zero, the object would not have been destroyed until Java’s own garbage collector was run. Presumably this would be at a time when the free memory of the system was low, meaning our garbage collector does not free memory when we require it. Rather, it allows objects to be collected when the JVM decides on a suitable time. Tests could be performed to find the optimal thresholds that ensure the most efficient garbage collection. Ideally, we want to run the TGRS garbage collection before Java’s own garbage collector. This will ensure the most amount memory is recovered.

6.3 Loss of Binding Information

During the preliminary testing phase of the \(\lambda\)-calculus interpreter, various \(\lambda\)-terms were evaluated and checked for correctness. In the \(\lambda\)-calculus, applying the identity to itself will always result in the identity. The second level produced

\[
(\lambda x.x)(\lambda y.y) = (\lambda y.y)(\lambda y.y) = (\lambda y.y)
\]

Interpreting the term to \(\chi\)-calculus,

\[
[[\lambda x.x](\lambda y.y)]_\alpha = [[\lambda x.x]]_\gamma \tilde{\eta} \tilde{k}([[\lambda y.y]]_\beta [k] \tilde{c}(c.\alpha))
\]

\[
[[\lambda y.y]]_\beta = \tilde{y}(y.\omega)\tilde{\omega} \cdot \beta
\]

\[
[[\lambda x.x]]_\gamma = \tilde{x}[x]_\delta \tilde{\gamma} \cdot \gamma
\]

\[
[[x x]]_\delta = (x.\mu)\tilde{\mu} [x] \tilde{z}(z.\delta)
\]

and so,

\[
[[((\lambda x.x)(\lambda y.y))]_\alpha = (\tilde{x}((x.\mu)\tilde{\mu} [x] \tilde{z}(z.\delta))_\delta \cdot \gamma \tilde{\eta} \tilde{k}((\tilde{y}(y.\omega)\tilde{\omega} \cdot \beta) [k] \tilde{c}(c.\alpha))
\]

If the cuts are chosen for reduction in a particular order, then the term normalises to \(\tilde{y}(y.\alpha)\tilde{\omega} \cdot \alpha\), and not \(\tilde{y}(y.\omega)\tilde{\omega} \cdot \alpha\), (\(\|\lambda y.y\|_\alpha\)) as expected. Appendix ?? lists the full reduction sequence. The problem occurring is clearly highlighted in step 19. A graphical representation of this is given in Figure 6.2.

The error arises as a result of the free variables in the capsule, \((y.\omega)\), being bound by two separate binders in different terms \((\tilde{y}(y.\omega)\tilde{\omega} \cdot \eta)\) and \((\tilde{y}(y.\omega)\tilde{\omega} \cdot \delta)\). Figure 6.3 explains this in the more familiar setting of the \(\lambda\)-calculus. We use the notation \([\ldots]\_M\) to describe a context.
Figure 6.2: Graphical Representation of $(\bar{g}(y.\omega)\bar{\omega} \cdot \eta)\bar{\eta} \upharpoonright \bar{q}(\bar{g}(y.\omega)\bar{\omega} \cdot \delta)\bar{\delta} [q] \bar{h}(h.\psi))$

Figure 6.3: Duplicate binders referring to the same term
After the application of Q to $(\lambda x[\ldots]_A)$, the occurrences of $x$ in the context are substituted with Q. However, $[\ldots]_B$ also had references to $x$, which has now become Q. The result is the binder $x$ in $\lambda x[\ldots]_B$ no longer binds with any occurrences in $[\ldots]_B$ since they have already been substituted with Q.

### 6.3.1 Implemented Solution

In Section 4.0.2 we looked at two alternative representation of term graphs for $\chi$. One captured the essence of binders whilst the other treated them the same as nodes. If we look at the representation for step 19 in Appendix B using the alternative representation of terms we can see the problem is no longer present (Figure 6.4) since all accesses to the capsule $(y,\beta)$ are made through the same binders.

![Figure 6.4: Alternate Graphical Representation of $(\tilde{g}(y,\omega)\tilde{\omega} \cdot \eta)\tilde{h} \doteq \tilde{q}(\tilde{g}(y,\omega)\tilde{\omega} \cdot \delta)\tilde{h} (h,\psi)$](image)

Since occurrences of variables can only be referenced through their binders, the problem described is bypassed. To implement the bound-variable system, we first extend our signature to include the intermediary nodes BP, BS and DB, which stand for bound plug, bound socket and double binding respectively. We also revise our previous definition of successors to that of Definition 6.1.
Definition 6.1 Revised definition of the successor function for the TGRS.

\[
\begin{align*}
succ \text{ cap} & ::= \text{socket plug} \\
succ \text{ exp} & ::= \text{DB plug} \\
succ \text{ cut} & ::= \text{BP BS} \\
succ \text{ med} & ::= \text{BP socket BS} \\
succ \text{ BP} & ::= \downarrow \text{ plug} \\
succ \text{ BS} & ::= \text{socket } \downarrow \\
succ \text{ BD} & ::= \text{socket } \downarrow \text{ plug} \\
succ \text{ plug} & ::= \emptyset \\
succ \text{ socket} & ::= \emptyset
\end{align*}
\]

Then \( \downarrow \) is defined as any element of the set \{\text{cap, exp, cut, med}\}.

Constructors for function symbols are updated to reflect their new structure. The toString(), toUnicode(), toLatex() and any function which makes recursive calls on successors are revised. Listing 6.1 shows a comparison of the old toUnicode() for the Exp class with the updated version.

```java
/*
   public String toUnicode(){
       StringBuffer sb = new StringBuffer(ChiToolkit.L_BRACKET);
       sb.append(succ.get(Exp.SOCKET).toUnicodeHat());
       sb.append(succ.get(Exp.TERM).toUnicode());
       sb.append(succ.get(Exp.CONNECTOR_PLUG).toUnicodeHat());
       sb.append(ChiToolkit.UNICODE_DOT);
       sb.append(succ.get(Exp.CONNECTOR_PLUG).toUnicode());
       sb.append(ChiToolkit.R_BRACKET);
       return sb.toString();
   }
*/

public String toUnicode(){
    StringBuffer sb = new StringBuffer(ChiToolkit.L_BRACKET);
    sb.append(succ.get(Exp.DOUBLE_BOUND).toUnicode());
    sb.append(ChiToolkit.UNICODE_DOT);
    sb.append(succ.get(Exp.CONNECTOR_PLUG).toUnicode());
    sb.append(ChiToolkit.R_BRACKET);
    return sb.toString();
}
```

**Listing 6.1: Updating the Exp.toUnicode() method**

The rule-editor is extended to function with the bound socket, plug and double binding terms and each rewrite rule is changed to the new format. Figure 6.5 shows a comparison of the previous and updated version of the dL rewrite rule.

We update the node-building phase in the reduction engine component to allow
isomorphic copies of the new nodes to be made. Finally, the parser and \( \lambda x \) interpreter are changed to create graphs using the updated function-symbol constructors. Evaluating the same expression using the new approach however did not lead to a correct reduction of the term.

If we consider the graphical representation of the \( ins \)-rule in the updated system, we can see the double-binding term is broken and new binders created in the body of the rule. This introduces the same error in to the system, and such results were seen in several places during further testing. A solution to this would be to introduce “rebinding-nodes” in to the TGRS which rename variables that produce conflicts in the term. Further details are given in [vBR04].

Figure 6.5: Rule Graph for \( dL \)
6.4 Summary and Further Work

The result of the project was the implementation of the calculus $\chi$ via term-graph rewriting methods, allowing $\chi$-terms to be executed and viewed as graphs. We defined and implemented a new data-structure, the cut-pointer stack and used it as a means for optimizing the term-graph rewriting process. We designed an intuitive point-and-click interface to allow easy reduction of terms in the calculus and finally we evaluated the tool, considering points that had been omitted during the implementation. We conclude with a list of further improvements and work that could be added to the project.

6.4.1 Type System

A type system for the calculus is presented in [vBLL03]. Additionally, Steffen van Bakel, an author of the paper has developed a principal typing algorithm for the calculus. This could be implemented and then the types displayed on the edges of the generated term-graph diagrams.

6.4.2 Improved GUI

The GUI was implemented using Java Swing (which supersedes Java AWT) and the Java2D API. During testing, it was seen that the GUI became unresponsive when several terms were being executed on-screen simultaneously. Additionally, a portion of time was spent debugging code to make it compatible with Apple’s implementation of the JVM.

Other GUI development packages should be investigated. One such package is the SWT (Standard Widget Toolkit) from IBM. It consists of a lightweight wrapper around native widgets, meaning no drawing is done in Java. It has been shown to be faster, richer and more stable than Swing, although it has its disadvantages such as differing behaviors on different platforms. There is also the issue of deployment since different packages are needed due to the native system calls.

6.4.3 Visualizing $\chi$-terms

The pictorial view of the term-graphs generated by the tools reveals a different method of visualizing terms in the calculus. The system could be extended with animation, so the user can see the rewrite rules being applied and transforming the $\chi$-expressions. The HOPS (Higher Order Programming System) does exactly this. It is a graphically interactive tool based on term-graphs with a feature that allows one to view a term being reduced as an animation. Implementing this as part of our tool could involve overlaying rule-body nodes,
redirecting then garbage collecting all in-front of the users eyes.

6.4.4 Recursive Let Expressions

So far, a simple naming mechanism has been implemented in the $\chi$-interpreter. The initial motivation was to aide entering terms with duplicate names and also to allow the user to specify sharing without using the graphical editor. The system could be extended to cater for recursive expressions like:

\[
\text{let } M = yM'b.'a \text{ in } M'a+x<x.'a>
\]

6.4.5 A Learning Tool

The tool should make sense to those who are already familiar to the calculus. However, to those learning about $\chi$ it can often be daunting. A simple English explanation of each reduction step could be presented to the user, allowing him/her to gain a better understanding of interacting terms.

6.4.6 Soundness and Correctness

The soundness and correctness of the system remains to be verified. This will require analysing the implementation from a theoretical perspective.
Appendix A

Graph Rewrite Rules

\((\text{var}) : (y, \alpha) \tilde{\alpha} \uparrow \tilde{x}(x, \beta) \rightarrow (y, \beta)\)

\[
\begin{align*}
&\text{r}_h \rightarrow \text{r}_b \\
&\text{y} \rightarrow \alpha \rightarrow \text{x} \rightarrow \beta \\
&\text{exp} \rightarrow \text{cap}
\end{align*}
\]

\((\text{exp}) : (\tilde{y}M\tilde{\beta} \cdot \alpha) \tilde{\alpha} \uparrow \tilde{x}(x, \gamma) \rightarrow \tilde{y}M\tilde{\beta} \cdot \gamma \alpha \notin \text{fs}(M)\)

\[
\begin{align*}
&\text{r}_h \rightarrow \text{r}_b \\
&\text{exp} \rightarrow \text{exp} \\
&\text{y} \rightarrow \beta \rightarrow \alpha \rightarrow \text{x} \rightarrow \gamma \\
&\text{cap} \rightarrow \text{cap}
\end{align*}
\]
\((\text{med}) : (y, \alpha) \tilde{\alpha} + \tilde{x}(N \tilde{\beta} [x] \exists P) \rightarrow N \tilde{\beta} [y] \exists P x \notin fs(N, P)\)
(insL) : \((\hat{\gamma} M \beta \cdot \alpha) \hat{\alpha} \vdash \hat{x}(N \hat{\gamma} [x] \hat{\gamma} P) \rightarrow N \hat{\gamma} \vdash \hat{y}(M \hat{\beta} \hat{\gamma} P) \alpha \not\in fs(M), x \not\in fs(N, P)\)

\[\text{Diagram Image}\]

(\text{insR}) : \((\hat{\gamma} M \beta \cdot \alpha) \hat{\alpha} \vdash \hat{x}(N \hat{\gamma} [x] \hat{\gamma} P) \rightarrow (N \hat{\gamma} + \hat{\gamma} M) \beta \hat{\gamma} P \alpha \not\in fs(M), x \not\in fs(N, P)\)

\[\text{Diagram Image}\]
(L1) : $(y, \beta) \bar{\alpha} \not\vdash \bar{xM} \rightarrow (y, \beta) \beta \neq \alpha$

(R1) : $M\bar{\alpha} \not\vdash \bar{x(y, \beta)} \rightarrow (y, \beta) \ y \neq x$
(L2) : $(\hat{y}M' \hat{\beta} \cdot \alpha) \hat{\alpha} \not\vdash \hat{x}M \rightarrow (\hat{y}(M' \hat{\alpha} \not\vdash \hat{x}M)\hat{\beta} \cdot \gamma) \gamma \vdash \hat{x}M, \gamma \text{ fresh}$

(R2) : $M\hat{\alpha} \not\vdash \hat{x}(\hat{y}M' \hat{\beta} \cdot \gamma) \rightarrow \hat{y}(M\hat{\alpha} \not\vdash \hat{x}M')\hat{\beta} \cdot \gamma$
(13) \( (\bar{\gamma}M'\bar{\beta} \cdot \gamma)\bar{\alpha} \not\in \bar{\pi}M \rightarrow \bar{\gamma}(M'\bar{\alpha} \not\in \bar{\pi}M)\bar{\beta} \cdot \gamma, \gamma \neq \alpha \)

(r3) \( M\bar{\alpha} \setminus \bar{x}(N\bar{\beta} [x] \bar{\gamma}P) \rightarrow M\bar{\alpha} \dagger \bar{\pi}(\bar{\pi}(M\bar{\alpha} \setminus \bar{x}N)\bar{\beta} [z] \bar{\gamma}(M\bar{\alpha} \setminus \bar{x}P)) \)
(L4) : \( (N \tilde{\beta} [z] \tilde{y} P) \tilde{\alpha} \not\vdash \tilde{x} M \rightarrow (N \tilde{\alpha} \not\vdash \tilde{x} M) \tilde{\beta} [z] \tilde{y} (P \tilde{\alpha} \not\vdash \tilde{x} M) \)

\[ \begin{array}{c}
\text{med} \downarrow \\
\text{med} \downarrow \\
\beta \downarrow \alpha \downarrow \gamma \downarrow \delta \\
\end{array} \]

(r4) : \( M \tilde{\alpha} \not\vdash \tilde{x} (N \tilde{\beta} [z] \tilde{y} P) \rightarrow (M \tilde{\alpha} \not\vdash \tilde{x} N) \tilde{\beta} [z] \tilde{y} (M \tilde{\alpha} \not\vdash \tilde{x} P) \not\vdash x \)

\[ \begin{array}{c}
\text{med} \downarrow \\
\text{med} \downarrow \\
\beta \downarrow \alpha \downarrow \gamma \downarrow \delta \\
\end{array} \]
(1.5) : \((N\widehat{\beta} \uparrow \widehat{\gamma} \overline{P})\overline{\alpha} \not\vdash \overline{x} \overline{M} \rightarrow (N\overline{\alpha} \not\vdash \overline{x} \overline{M})\widehat{\beta} \uparrow \widehat{\gamma}(P\overline{\alpha} \not\vdash \overline{x} \overline{M})\)

(r5) : \(M\overline{\alpha} \not\vdash \overline{x}(N\widehat{\beta} \uparrow \widehat{\gamma} \overline{P}) \rightarrow (M\overline{\alpha} \not\vdash \overline{x} N)\widehat{\beta} \uparrow \widehat{\gamma}(M\overline{\alpha} \not\vdash \overline{x} P)\)
\[(act-L) : M\hat{\alpha} \vdash \hat{x}N \rightarrow M\hat{\alpha} \not\vdash \hat{x}N \text{ if } M \text{ does not introduce } \alpha\]

\[(act-R) : M\hat{\alpha} \vdash \hat{x}N \rightarrow M\hat{\alpha} \not\vdash \hat{x}N \text{ if } N \text{ does not introduce } x\]
(d.) : \( \langle y.\alpha \rangle \hat{\alpha} \not\in \hat{x} M \rightarrow \langle y.\alpha \rangle \hat{\alpha} \uparrow \hat{x} M \)

(dh) : \( M\hat{\alpha} \land \hat{x}\langle x.\beta \rangle \rightarrow M\hat{\alpha} \uparrow \hat{x}\langle x.\beta \rangle \)
(ren-L): $M\tilde{\delta} \uparrow \tilde{\exists}(z, \alpha) \rightarrow M[\alpha/\delta]$

(ren-R): $(z, \alpha)\tilde{\alpha} \uparrow \tilde{\exists}M \rightarrow M[z/x]$
(gc-L): $M\tilde{\alpha} \not\vdash \hat{x}N \rightarrow M$ if $\alpha \notin fp(M)$

(ge-R): $M\tilde{\alpha} \setminus \hat{x}N \rightarrow N$ if $x \notin fs(N)$
Appendix B

Erroneous Reduction of

\((\lambda x.xx)(\lambda y.y)\)

1. \((\hat{x}((x.x)\hat{\xi} \uparrow \hat{k}((x.x)\hat{\delta} [k] \hat{h}(h,\mu))\hat{\mu} \cdot \tau) \uparrow \hat{x}((\hat{y}(y.\omega)\hat{\omega} \cdot \eta)\hat{\eta} [c] \hat{p}(p.\psi))) \rightarrow (insL)

2. \((\hat{y}(y.\omega)\hat{\omega} \cdot \eta)\hat{\eta} \uparrow \hat{x}(((x.x)\hat{\xi} \uparrow \hat{k}((x.x)\hat{\delta} [k] \hat{h}(h,\mu))\hat{\mu} \uparrow \hat{\mu}(p.\psi))) \rightarrow (act - L)

3. \((\hat{y}(y.\omega)\hat{\omega} \cdot \eta)\hat{\eta} \uparrow \hat{x}(((x.x)\hat{\xi} \uparrow \hat{k}((x.x)\hat{\delta} [k] \hat{h}(h,\mu))\hat{\mu} \uparrow \hat{\mu}(p.\psi))) \rightarrow (L5)

4. \((\hat{y}(y.\omega)\hat{\omega} \cdot \eta)\hat{\eta} \uparrow \hat{x}(((x.x)\hat{\xi} \uparrow \hat{\mu}(p.\psi))\hat{\xi} \uparrow \hat{k}((x.x)\hat{\delta} [k] \hat{h}(h,\mu))\hat{\mu} \uparrow \hat{\mu}(p.\psi))) \rightarrow (L4)

5. \((\hat{y}(y.\omega)\hat{\omega} \cdot \eta)\hat{\eta} \uparrow \hat{\mu}(p.\psi)) \rightarrow (L1)

6. \((\hat{y}(y.\omega)\hat{\omega} \cdot \eta)\hat{\eta} \uparrow \hat{\mu}(p.\psi)) \rightarrow (act - L)

7. \((\hat{y}(y.\omega)\hat{\omega} \cdot \eta)\hat{\eta} \uparrow \hat{\mu}(p.\psi)) \rightarrow (L1)

8. \((\hat{y}(y.\omega)\hat{\omega} \cdot \eta)\hat{\eta} \uparrow \hat{\mu}(p.\psi)) \rightarrow (dL)

9. \((\hat{y}(y.\omega)\hat{\omega} \cdot \eta)\hat{\eta} \uparrow \hat{\mu}(p.\psi)) \rightarrow (med)

10. \((\hat{y}(y.\omega)\hat{\omega} \cdot \eta)\hat{\eta} \uparrow \hat{\mu}(p.\psi)) \rightarrow (dL)

11. \((\hat{y}(y.\omega)\hat{\omega} \cdot \eta)\hat{\eta} \uparrow \hat{\mu}(p.\psi)) \rightarrow (act - R)

12. \((\hat{y}(y.\omega)\hat{\omega} \cdot \eta)\hat{\eta} \uparrow \hat{\mu}(p.\psi)) \rightarrow (R3)

13. \((\hat{y}(y.\omega)\hat{\omega} \cdot \eta)\hat{\eta} \uparrow \hat{\mu}(p.\psi)) \rightarrow (R5)

14. \((\hat{y}(y.\omega)\hat{\omega} \cdot \eta)\hat{\eta} \uparrow \hat{\mu}(p.\psi)) \rightarrow (R1)
15. \((\vec{g}(y,\omega)\vec{\omega} \cdot \eta) \uparrow \vec{q} \)
   \(((\vec{g}(y,\omega)\vec{\omega} \cdot \eta) \uparrow \vec{x}(x,\delta)) \uparrow [q] \vec{h}(h,\mu) \uparrow \vec{p}((\vec{g}(y,\omega)\vec{\omega} \cdot \eta) \uparrow \vec{x}(x,\delta)) \rightarrow (R1)\)

16. \((\vec{g}(y,\omega)\vec{\omega} \cdot \eta) \uparrow \vec{q}((\vec{g}(y,\omega)\vec{\omega} \cdot \eta) \uparrow \vec{x}(x,\delta)) \uparrow [q] \vec{h}(h,\mu) \uparrow \vec{p}(p,\psi)) \rightarrow (var)\)

17. \((\vec{g}(y,\omega)\vec{\omega} \cdot \eta) \uparrow \vec{q}((\vec{g}(y,\omega)\vec{\omega} \cdot \eta) \uparrow \vec{x}(x,\delta)) \uparrow [q] \vec{h}(h,\psi)) \rightarrow (dR)\)

18. \((\vec{g}(y,\omega)\vec{\omega} \cdot \eta) \uparrow \vec{q}((\vec{g}(y,\omega)\vec{\omega} \cdot \eta) \uparrow \vec{x}(x,\delta)) \uparrow [q] \vec{h}(h,\psi)) \rightarrow (exp)\)

19. \((\vec{g}(y,\omega)\vec{\omega} \cdot \eta) \uparrow \vec{q}((\vec{g}(y,\omega)\vec{\omega} \cdot \delta) \uparrow [q] \vec{h}(h,\psi)) \rightarrow (insL)\)

20. \((\vec{g}(y,\omega)\vec{\omega} \cdot \delta) \uparrow \vec{g}(y,\omega) \uparrow \vec{h}(h,\psi)) \rightarrow (act - R)\)

21. \((\vec{g}(y,\omega)\vec{\omega} \cdot \delta) \uparrow \vec{g}(y,\omega) \uparrow \vec{h}(h,\psi)) \rightarrow (R5)\)

22. \(((\vec{g}(y,\omega)\vec{\omega} \cdot \delta) \uparrow \vec{g}(y,\omega) \uparrow \vec{h}(h,\psi)) \rightarrow (act - L)\)

23. \(((\vec{g}(y,\omega)\vec{\omega} \cdot \delta) \uparrow \vec{g}(y,\omega) \uparrow \vec{h}(h,\psi)) \rightarrow (R1)\)

24. \(((\vec{g}(y,\omega)\vec{\omega} \cdot \delta) \uparrow \vec{g}(y,\omega) \uparrow \vec{h}(h,\psi)) \rightarrow (dR)\)

25. \(((\vec{g}(y,\omega)\vec{\omega} \cdot \delta) \uparrow \vec{g}(y,\omega) \uparrow \vec{h}(h,\psi)) \rightarrow (exp)\)

26. \((\vec{g}(y,\omega)\vec{\omega} \cdot \omega) \uparrow \vec{h}(h,\psi) \uparrow (L2)\)

27. \((\vec{g}(y,\omega) \uparrow \vec{h}(h,\psi) \uparrow \vec{\omega} \cdot \alpha) \uparrow \vec{h}(h,\psi) \rightarrow (dL)\)

28. \((\vec{g}(y,\omega) \uparrow \vec{h}(h,\psi) \uparrow \vec{\omega} \cdot \alpha) \uparrow \vec{h}(h,\psi) \rightarrow (exp)\)

29. \((\vec{g}(y,\omega) \uparrow \vec{h}(h,\psi) \uparrow \vec{\omega} \cdot \psi) \rightarrow (var)\)

30. \((\vec{g}(y,\omega) \uparrow \vec{\omega} \cdot \psi)\)
Bibliography


