# Diplomarbeit

## Static analysis of functional programs via Linear Logic

Alexander Bach

Advisor:
Priv. Doz. Dr. David Basin

MPI–I–96– March 1996

**Author's Address**

Alexander Bach
Max-Planck Institut fuer Informatik
Im Stadtwald,
Building 46.1,
D-66123 Saarbrücken
E-mail: bach@mpi-sb.mpg.de

**Acknowledgements**

First and foremost I would like to thank Priv.Doz. Dr. David Basin for supervising this thesis and Dipl.Inf. A.Tönne who has commented extensively on this work. I owe them a lot of important insights. I also want to thank Dr.P.Barth and PhD.S.Matthews at MPI for their readiness to answer every question that I had. I would like to thank Dr.E.Weydert for nonmonotonic arguments and Uwe Brahm for his fine Internet expertise. My friend Abdelwaheb Ayari was there when I needed him. I must also thank my fellow students Erik, Georg, Torsten, Peter, Stefan and Jan for good conversations we had. My mother, Ingeborg Bach, has supported me from the beginning. I thank Dr.J.E.Doherty for his encouragement. My girlfriend Antje must be thanked for putting up with all this.

This work is dedicated to the memory of my father, Dr.med. Hans-Joachim Bach.

**Abstract**

This thesis investigates aspects of the general relationship between simply typed $\lambda$-calculus and a linear term calculus based on Intuitionistic Linear Logic. It introduces a notion of minimization on linear $\lambda$-terms that removes superflous nonlinear operations (storage). Two different embeddings of the simply typed $\lambda$-calculus into the linear term calculus are studied with respect to their properties under minimization. We define operational semantics for both term calculi. In support of Abramsky's thesis, that linear types are useful in doing abstract interpretation of functional programs, we demonstrate - using translation together with minimization - a syntactic method to do strictness analysis on $\lambda$-terms, via the linear typing calculus. This leads to useful optimizations of call-by-name reduction on $\lambda$-terms.

**Erklärung**

Ich erkläre, die vorliegende Arbeit selbstständig im Sinne der Diplomprüfungsordnung erstellt und ausschließlich die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Saarbrücken, den 14.3.1996

A.Bach

# Contents

# List of Figures

*Analysis does not reduce the complex to the simple. Rather, it's a process by which we substitute a familiar complexity for one that we have found novel.*

*D.C.North, Nobel Prize in Economics 1994*

# Chapter 1

# Introduction

## 1.1 General Outline

The main concern of this work are different notions of *type*.

### 1.1.1 Types

The word 'type' denotes a collection of objects that are 'uniform' with respect to certain relevant properties. Classifying objects by assigning types abstracts away from irrelevant properties. Different classification systems, or systems of type assignment, draw upon different notions of relevance. Objects may be identified in one system that can be distinguished by a subtler, more complex system. The right choice of type assignment system for a given collection of objects must be guided by pragmatic considerations: a type should reflect just enough - but no more - information to solve the problem at hand.

The principal objects to which the concept of typing will be applied in this work are *functional programs*. Types in this context encode abstract properties of functional programs, serving two different purposes:

1. They can be read as partial specifications of a program's behavior. Verifying that a certain program has indeed a (pre-)specified type may thus be seen as a partial correctness proof for the program.

2. The information they carry may be used for optimizing the program's execution.

### 1.1.2 Terms

Two different typed functional programming languages will be studied in the sequel. The first is based on the simply-typed lambda-calculus [Chu40], extended with programming constructs such as lists, let-definitions and recursion. As a prototype language, it contains the core constructs of existing functional programming languages such as Standard ML [MTH90].

The second language is based on Linear Logic [Gir87] and was first proposed by Abramsky [Abr93]. It is equally extended with lists, recursion and let-definitions in order to match the strength of the previous language. However, programs will look quite differently in the two systems. This is due to the fact that the language based on Linear Logic features additional operators that have no counterpart in traditional functional programs. These additional operators make the use of resources in a program explicit:

- Whenever a programmer using linear programs wants to discard a value, he must make his intention known through the explicit use of a **drop**-operator.
- Likewise, if a programmer wants to copy a value he must do so by writing an explicit **copy** operation.

Such operations are dealt with implicitly in traditional functional programming languages. In this respect, the linear functional language may be seen as a *refinement* of traditional-style languages, since more information is required from the programmer about the intended behaviour of his program.

This refinement has advantages and drawbacks. On the one hand, it provides additional means to fine-tune the use of resources in a computation, but on the other hand, tracking these uses consistently throughout a large program is an almost impossible task [Mac91].

In dealing with this drawback, this work proposes a pragmatic approach. Instead of compelling the programmer to care about the use of resources, tools are described that empower the compiler of the traditional language to *infer* information about the use of resources. This information may subsequently be used to automatically perform those optimizations that increase the overall efficiency of a program's execution, thereby sparing the programmer's time and attention.

This approach is much in line with the common property of typed functional languages, that *types* do not have to be attached to terms explicitly, but can be algorithmically inferred.

And indeed, it will become clear that the problem of analyzing the use of resources in a functional program amounts to *inferring a linear type* for the respective program. This is due to the fact that linear types are intended to capture exactly how resources are used. In this sense it may be said that linear types reflect operational properties of programs.

### 1.1.3 Translations

There are several distinct, incomparable ways to assign linear types to traditional functional programs however. This work will give examples on how they differ and why they are incomparable.

The problem of linear type assignment to traditional functional programs is equivalent to the problem of *relating terms* in the two different calculi. In order to study this problem, two different, incomparable translations between the calculi are defined in the sequel.

Each translation that we present however, induces a collection of linear types that are uniform with respect to the structure of the typing proofs that derive them. Recalling the definition of types from the beginning of this introduction, one may say that different translations embody different types (kinds) of linearity types, each of which draws upon a different notion of *proof structure*.

These translations serve as the basic tool in investigating the utility of the information encoded in linear types.

### 1.1.4 Logic-based program analysis

Following the proposal of Abramsky [Abr93], to use the information obtained from linearity analysis to optimize the execution of the the respective term, we demonstrate with our methods how one particular translation may be used to do

Strictness Analysis via Linear Logic Types

The basic feasibility of Logic-based program analysis is the main result of this thesis.

## 1.2 Overview

- Chapter 2:After explaining some general concepts, two different type assignment calculi for functional programs - $\lambda^{\rightarrow}$ based on Intuitionistic Logic and $\lambda^{!,\multimap}$ based on Intuitionistic Linear Logic - will be defined. They are the basis for comparing notions of type with different levels of granularity.

Moreover, a constrained calculus, $c - \lambda^{!,\multimap}$, will be derived from $\lambda^{!,\multimap}$ that relates terms not only to types but also to certain propositional constraints. These reflect additional properties of the respective terms that may not be captured on the level of linear types alone.

The information encoded by these constraints is subsequently used to define a procedure $\mu_!()$ ('minimization') that maps well-typed $\lambda^{!,\multimap}$-terms to well-typed $\lambda^{!,\multimap}$-terms which may be considered 'smaller' in a precisely definable sense. Soundness of $\mu_!(.)$ with respect to provability in $\lambda^{!,\multimap}$ is shown.

This yields the first example of an optimization based on typing information.

- Chapter 3 defines the operational semantics of $\lambda^{\rightarrow}$ and $\lambda^{!,\multimap}$ and surveys their most important properties.
- Chapter 4 defines translations from $\lambda^{\rightarrow}$ to $\lambda^{!,\multimap}$ and studies their properties. It will be argued that these translations reflect the basic choices that exist in making explicit the linearity structure of $\lambda^{\rightarrow}$-terms.
- Chapter 5 considers the applicability of $\lambda^{!,\multimap}$ to the implementation of call-by-name reduction. The main theorem states that it is safe to reduce linear terms in an eager fashion without corrupting call-by-name termination semantics only under one particular translation.
- Chapter 6 describes an extension to the constraint system that characterizes the structure of linear proofs in order to obtain call-by-need reduction.

# Chapter 2

# Preliminaries

## 2.1 Basic terminology

All the typing calculi considered in this work share common characteristics that shall be presented before the individual systems are defined. The following sections will introduce the basic ways of forming and associating terms and types:

- A *type assignment* is a pair $t : \tau$ associating a $\lambda$-term with a type.
- A *type environment* $\Delta = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ is a *finite multiset* of type assignments to term variables (called *assumptions*). The empty type environment will be denoted by $\emptyset$.
- A *typing judgment* is a relation $\vdash$ between type environments and type assignments written as $\Delta \vdash t : A$ and read as 'in the type environment $\Delta$, $t$ is a well-typed term of type $A$.
- *Validity* of a typing judgment is expressed as derivability in a typing calculus presented by a (finite) set of axioms and rules of the form

  $$\overline{\Delta \vdash t : A}$$

  and

  $$\frac{\Delta_1 \vdash t_1 : A_1 \quad \ldots \quad \Delta_n \vdash t_n : A_n}{\Delta \vdash t : A}$$

  The set of valid typing judgments with respect to such a system is the least relation that contains all axioms and is closed under application of the rules. *Derivability* of a judgment $s$ in a type-system $R$ will be denoted by $s \in R$.
- A *typing derivation* is a tree with nodes labeled by typing judgments.

$$\begin{array}{c} p \\ I \\ \Delta \vdash t : A \end{array}$$

denotes the typing derivation p ending with typing judgment $\Delta \vdash t : A$ derived by inference I.

- A term $t$ is said to be *typeable* in a typing calculus $K$ if and only if $\Delta \vdash t : A \in K$ for some type $A$ and environment $\Delta$. The notation $t \in K$ is used as a shorthand for expressing typeability of $t$ in $K$.

- Given a typing calculus K, $\lambda^K$ denotes the set $\{t \mid \Delta \vdash t : A \in K\}$. $\lambda_0^K = \{t \mid \emptyset \vdash t : A \in K\}$ denotes the set of closed terms typeable in $K$.

- the symbol $\equiv$ denotes syntactic equality between terms.

**Definition 2.1 Variables** We assume infinite countable sets

- $\mathcal{V} = \{x_1, x_2, x_3, \ldots\}$ of *program variables*. Program variables will also be written as $x, y, z$.

- $\mathcal{T} = \{A_1, A_2, A_3, \ldots\}$ of *type variables*. These will also be written as $A, B, C$.

$\triangleleft$

## 2.2 The Simply Typed $\lambda$-Calculus $\lambda^{\rightarrow}$

The system $\lambda^{\rightarrow}$, the prototype of all typed function calculi first introduced by Curry in [Cur34], together with extensions for programming purposes, will now be introduced as a basis for all later investigations.

**Definition 2.2** $\lambda^{\rightarrow}$ Assume an infinite collection $\mathcal{V} = \{v_1, v_2, \ldots\}$ of *type variables*. The set of simple types, denoted by $\tau$, is defined by the following abstract grammar:

$$\tau =_{df} \mathcal{V} \mid \mathbf{1} \mid \tau \rightarrow \tau \mid \tau \wedge \tau \mid \tau \vee \tau$$

A type is said to be *ground* if it contains no variables. The type $\mathbf{1}$ will be called the*base type*. $\triangleleft$

**Definition 2.3 Untyped pre-terms for** $\lambda^{\rightarrow}$ The set of untyped pre-terms in $\lambda^{\rightarrow}$ is defined by the following grammar:

$$
\begin{array}{rcl}
t & =_{df} & x \\
 & | & () \\
 & | & \mathbf{let}\ ()\ \mathbf{be}\ t_1\ \mathbf{in}\ t_2 \\
 & | & \lambda x.t \\
 & | & (t_1\, t_2) \\
 & | & \pi_2\, t \\
 & | & \pi_1\, t \\
 & | & (t_1, t_2) \\
 & | & \mathbf{inr}(t) \\
 & | & \mathbf{inl}(t) \\
 & | & \mathbf{let}\ (x; y)\ \mathbf{be}\ t_1\ \mathbf{in}\ (t_2; t_3)
\end{array}
$$

◁

## 2.2.1 Core system

**Definition 2.4 Typing rules for** $\lambda^{\rightarrow}$ The typing rules for $\lambda^{\rightarrow}$ defining the set of derivable judgments and consequently, the set of well-typed terms are:

**Axiom**
$$\frac{}{x : A \vdash x : A}\ axiom\ (A \in \tau)$$

**Unit**
$$\frac{}{\vdash () : \mathbf{1}}\ \mathbf{1}_I \qquad \frac{\Delta \vdash t_1 : \mathbf{1} \quad \Delta \vdash t_2 : A}{\Delta \vdash \mathbf{let}\ ()\ \mathbf{be}\ t_1\ \mathbf{in}\ t_2 : A}\ \mathbf{1}_E$$

**Structure**
$$\frac{\Delta \vdash t : B}{\Delta, x : A \vdash t : B}\ w \qquad \frac{\Delta, x : A, x : A \vdash t : B}{\Delta, x : A \vdash t : B}\ c$$

**Function**
$$\frac{\Delta, x : A \vdash t : B}{\Delta \vdash \lambda x.t : A \rightarrow B}\ \rightarrow_I \qquad \frac{\Delta \vdash t_1 : A \rightarrow B \quad \Delta \vdash t_2 : A}{\Delta \vdash (t_1\, t_2) : B}\ \rightarrow_E$$

**Product**
$$\frac{\Delta \vdash t : A \wedge B}{\Delta \vdash \pi_1\, t : A}\ \wedge_{E1} \qquad \frac{\Delta \vdash t : A \wedge B}{\Delta \vdash \pi_2\, t : B}\ \wedge_{E2}$$
$$\frac{\Delta \vdash t_1 : A \quad \Delta \vdash t_2 : B}{\Delta \vdash (t_1, t_2) : A \wedge B}\ \wedge_I$$

**Sum**
$$\frac{\Delta \vdash t : A}{\Delta \vdash \mathbf{inl}(t) : A \vee B}\ \vee_{I1} \qquad \frac{\Delta \vdash t : B}{\Delta \vdash \mathbf{inr}(t) : A \vee B}\ \vee_{I2}$$
$$\frac{\Delta \vdash t_1 : A \vee B \quad \Delta, x : A \vdash t_2 : C \quad \Delta, y : B \vdash t_3 : C}{\Delta \vdash \mathbf{let}\ (x; y)\ \mathbf{be}\ t_1\ \mathbf{in}\ (t_2; t_3) : C}\ \vee_{E1}$$

◁

Note that the structural rules merely manipulate assumptions in the context without changing the term on the right-hand side of the turnstile. The [w]-rule is also called weakening and the [c]-rule contraction. Weakening removes an assumption from a context while contraction duplicates an assumption.

### 2.2.2 Substitutions

We shall now define the notions of free and bound variables, and subterms in $\lambda^{\rightarrow}$. Let $t, t', t_1, t_2 \in \lambda^{\rightarrow}$.

$Fv(t)$, the set of *free* variables in a $\lambda^{\rightarrow}$-term is defined simultaneously with the set of $Bv(t)$, the set of *bound* variables and $St(t)$, the set of sub-terms of $t$, by the following table:

**Definition 2.5 Free and bound variables, Sub-terms**

| $t$ | $Fv(t)$ | $Bv(t)$ | $St(t)$ |
|---|---|---|---|
| $x$ | $\{x\}$ | $\emptyset$ | $\{x\}$ |
| $()$ | $\emptyset$ | $\emptyset$ | $\{t\}$ |
| **let** $()$ **be** $t_1$ **in** $t_2$ | $Fv(t_1) \cup Fv(t_2)$ | $Bv(t_1) \cup Bv(t_2)$ | $\{t\} \cup St(t_1) \cup St(t_2)$ |
| $\lambda x.t'$ | $Fv(t') \setminus \{x\}$ | $Bv(t') \cup (\{x\} \cap Fv(t'))$ | $\{t\} \cup St(t')$ |
| $(t_1\, t_2)$ | $Fv(t_1) \cup Fv(t_2)$ | $Bv(t_1) \cup Bv(t_2)$ | $\{t\} \cup St(t_1) \cup St(t_2)$ |
| $(t_1, t_2)$ | $Fv(t_1) \cup Fv(t_2)$ | $Bv(t_1) \cup Bv(t_2)$ | $\{t\} \cup St(t_1) \cup St(t_2)$ |
| $\pi_1\, t'$ | $Fv(t')$ | $Bv(t')$ | $\{t\} \cup St(t')$ |
| $\pi_2\, t'$ | $Fv(t')$ | $Bv(t')$ | $\{t\} \cup St(t')$ |
| **inl**$(t')$ | $Fv(t')$ | $Bv(t')$ | $\{t\} \cup St(t')$ |
| **inr**$(t')$ | $Fv(t')$ | $Bv(t')$ | $\{t\} \cup St(t')$ |
| **let** $(x; y)$ **be** $t_1$ **in** $(t_2; t_3)$ | $Fv(t_1)$ $\cup(Fv(t_2) \setminus \{x\}$ $\cup Fv(t_3) \setminus \{y\})$ | $Bv(t_1)$ $\cup Bv(t_2) \cup (\{x\} \cap Fv(t_2))$ $\cup Bv(t_3) \cup (\{y\} \cap Fv(t_3))$ | $\{t\} \cup St(t_1)$ $\cup St(t_2) \cup St(t_3)$ |

$\triangleleft$

Obviously, $\forall t \in \lambda_0^{\rightarrow}.Fv(t) = \emptyset$. If $u$ and $t$ are terms, the notation $t[u]$ will be used as a shorthand for $u \in St(t)$.

In an abstraction $\lambda x.t$, the first occurrence of x is called its *binding occurrence*, the variable $x$ itself is called the *formal parameter* and the term $t$ is called the *body* of the abstraction. The same terminology applies to case-splits **let** $(x; y)$ **be** $t$ **in** $(t_1; t_2)$ where $x$ and $y$ are the formal parameters and $(t_1; t_2)$ constitutes the body of the alternative.

**Definition 2.6 Substitutions** Let $u \in \lambda^{\rightarrow}$. Substitutions are written as $[x/u]$ and read as 'replace x by u. The composition of several substitutions is written as $[x_1/t_1] \dots [x_n/t_n]$. Simultaneous substitution of several terms for distinct variables is written as $[x_1/t_1, \dots x_n/t_n]$. The result of applying a substitution $[x/u]$ to term $t$, written as $t[x/u]$, is defined by induction on the structure of t. Most cases are obvious, the only interesting ones being when substitutions are applied to terms with binding operators. Consider abstraction as the standard case. We must distinguish three distinct possibilities:

$$\lambda y.t[x/u] \equiv \begin{cases} \lambda y.t & \text{if } x = y \\ \lambda z.t[y/z][x/u] & \text{if } x \neq y \wedge y \in Fv(u) \text{ where } z \notin Fv(u) \wedge z \notin Bv(t) \\ \lambda y.t[x/u] & \text{otherwise, } x \neq y \end{cases}$$

The first case is concerned with avoiding substitution into bound variables. The second case is concerned with avoiding the problem of *name capture*, where a free variable in the substituted term may become bound because it is substituted into the scope of an equally named binding variable. This pathology is avoided by renaming bound variables ($\alpha$-renaming) prior to substitution.

The treatment of the other binding operators is similar. ◁

**Remark 2.7 (Variable convention,cf. [Bar84a])** In order not to worry about the alternative cases that arise when substituting into terms with binding operators, we will tacitly assume that problematical cases are automatically avoided. In particular, terms that are syntactically equal under $\alpha$-renaming will be implicitly identified:

$$\lambda x.t \equiv_{\alpha} \lambda y.t[x/y]$$

This convention must suffice for the present theoretical treatment. However, keep in mind that an implementation of the $\lambda^{\to}$-calculus will have to deal explicitly with such problems. One possible solution is provided by the name-free notation ('deBruijn indices') of deBruijn (cf. [Bar84b], Appendix C.). ◁

### 2.2.3 Extensions

With the aim of turning $\lambda^{\to}$ into a veritable functional programming language, we extend it with lists, recursion and let definitions.

The abstract grammar for $\tau$, the set of simple types is extended by the list type $(\tau)\,list$.

The grammar for the set of untyped pre-terms in $\lambda^{\to}$ is extended with the following abstract termforms:

$$\begin{array}{ll} & [\,] \\ | & \lambda^{\to}{::}\lambda^{\to} \\ | & \mathbf{case}\ \lambda^{\to} = [\,] \Rightarrow \lambda^{\to} \mid (x{::}y) \Rightarrow \lambda^{\to} \\ | & \mathbf{let}\ x\ \mathbf{be}\ \lambda^{\to}\ \mathbf{in}\ \lambda^{\to} \\ | & \mathbf{fix}(x, \lambda^{\to}) \end{array}$$

Accordingly, the system of type inference rules is extended as follows:

**Definition 2.8 Programming extensions**

**Lists**

$$\frac{}{\vdash [\,] : (A)\,list}\,Nil \qquad \frac{\Delta \vdash t_1 : A \quad \Delta \vdash t_2 : (A)\,list}{\Delta \vdash t_1{::}t_2 : (A)\,list}\,Cons$$

$$\frac{\Delta \vdash t_1 : (A)\,list \quad \Delta \vdash t_2 : C \quad \Delta, h : A, tl : (A)\,list \vdash t_3 : C}{\Delta \vdash \mathbf{case}\; t_1 = [\,] \Rightarrow t_2 \mid (h{::}tl) \Rightarrow t_3 : C}\,lcase$$

**Let**

$$\frac{\Delta \vdash t_1 : A \quad \Delta, x : A \vdash t_2 : B}{\Delta \vdash \mathbf{let}\; x\; \mathbf{be}\; t_1\; \mathbf{in}\; t_2 : B}\,let_I$$

**Recursion**

$$\frac{\Delta, x : A \vdash t : A}{\Delta \vdash \mathbf{fix}(x, t) : A}\,fix$$

◁

## 2.3 Linear types

Now the second type-assignment system is introduced. It is based on the intuitionistic fragment of Linear Logic (cf. [Gir87]), providing a more refined notion of type than $\lambda^{\rightarrow}$:

**Definition 2.9 Linear types** The set of linear types, denoted by $\tau_!$ is defined by the following abstract grammar:

$\tau_! ::= \mathcal{V} \mid \mathbf{1} \mid !\tau_! \mid \tau_! \multimap \tau_! \mid \tau_! \otimes \tau_! \mid \tau_! \,\&\, \tau_! \mid \tau_! \oplus \tau_!$ ◁

The type-forming constructors that are used instead of the traditional implication, conjunction and disjunction of $\lambda^{\rightarrow}$ in this definition need some explanation:

- $A \multimap B$, the *linear implication*, is the type of functions that use their argument exactly once.
- $A \,\&\, B$ is the type of *additive* pairs whose use is restricted to projecting either one component while the other component is released.
- $A \otimes B$ is the type of *multiplicative* pairs whose use requires projecting both components simultaneously, obviating the choice that is necessitated by additive pairs.
- $A \oplus B$ is the *sum*-constructor
- $!A$, pronounced as 'of-course A', denotes the set of terms that are subject to nonlinear use. This operator has no counterpart in $\lambda^{\rightarrow}$. It is used in $\lambda^{!,\multimap}$ to distinguish terms that are contracted or weakened from linear terms, a distinction not made in $\lambda^{\rightarrow}$.

Before the set of untyped pre-terms can be defined, note that the notions of free and bound variable are also applicable in $\lambda^{!,\multimap}$, just as substitutions are, under the usual provisos.

**Definition 2.10 Untyped pre-terms of $\lambda^{!,\multimap}$** Terms in $\lambda^{!,\multimap}$ may not be defined in the same liberal style as $\lambda^{\rightarrow}$-terms. The free variables they contain are subject to certain constraints. The rules of the grammar are given together with these constraints.

$$
\begin{array}{rlll}
t & =_{df} & x & \\
 & | & () & \\
 & | & \mathbf{let}\ ()\ \mathbf{be}\ t_1\ \mathbf{in}\ t_2 & Fv(t_1) \cap Fv(t_2) = \emptyset \\
 & | & \lambda x.t & \\
 & | & (t_1\, t_2) & Fv(t_1) \cap Fv(t_2) = \emptyset \\
 & | & (t_1, t_2) & Fv(t_1) \cap Fv(t_2) = \emptyset \\
 & | & \mathbf{let}\ (x, y)\ \mathbf{be}\ t_1\ \mathbf{in}\ t_2 & Fv(t_1) \cap (Fv(t_2) \setminus \{x, y\}) = \emptyset \\
 & | & \langle t_1,\ t_2 \rangle & Fv(t_1) = Fv(t_2) \\
 & | & \pi_1\, t & \\
 & | & \pi_2\, t & \\
 & | & \mathbf{inl}(t) & \\
 & | & \mathbf{inr}(t) & \\
 & | & \mathbf{let}\ (x; y)\ \mathbf{be}\ t_1\ \mathbf{in}\ (t_2; t_3) & Fv(t_1) \cap (Fv(t_2) \setminus \{x\} \cup Fv(t_3) \setminus \{y\}) = \emptyset \\
 & & & Fv(t_2) \setminus \{x\} = Fv(t_3) \setminus \{y\} \\
 & | & !(t) & \\
 & | & \star t & \\
 & | & \mathbf{drop}\ t_1\ \mathbf{in}\ t_2 & Fv(t_1) \cap Fv(t_2) = \emptyset \\
 & | & \mathbf{copy}\ t_1\ \mathbf{as}\ x/y\ \mathbf{in}\ t_2 & Fv(t_1) \cap Fv(t_2) = \emptyset
\end{array}
$$

$\triangleleft$

### 2.3.1 Core system

Remember that type environments $\Delta$ in typing judgements $\Delta \vdash t : A$ have been defined as multisets. This included the possibility that assumptions $x : A$ have multiset status greater than 1, which is admissible in $\lambda^{\rightarrow}$. In $\lambda^{!,\multimap}$ however, it is required that every assumption occuring in the environment of a derivable typing judgement must have multiset status 1 in order to ensure that the free variable constraints on untyped pre-terms can not be violated. Under this restriction, all variables in the type environment are *distinct*.

The typing rules for $\lambda^{!,\multimap}$ defining the set of derivable judgments and consequently, the set of well-typed terms are:

**Axiom** $$\frac{}{x : A \vdash x : A}\ axiom\ (A \in \tau_!)$$

**Unit** $$\frac{}{\vdash () : \mathbf{1}}\ \mathbf{1}_I \qquad \frac{\Delta \vdash t_1 : \mathbf{1} \quad \Gamma \vdash t_2 : A}{\Delta, \Gamma \vdash \mathbf{let}\ ()\ \mathbf{be}\ t_1\ \mathbf{in}\ t_2 : A}\ \mathbf{1}_E$$

**Function** $$\frac{\Delta, x : A \vdash t : B}{\Delta \vdash \lambda x.t : A \multimap B}\ \multimap_I \qquad \frac{\Delta \vdash u : A \quad \Gamma \vdash t : A \multimap B}{\Delta, \Gamma \vdash (t\ u) : B}\ \multimap_E$$

**Tensor** $$\frac{\Delta \vdash t_1 : A \quad \Gamma \vdash t_2 : B}{\Delta, \Gamma \vdash (t_1, t_2) : A \otimes B}\ \otimes_I \qquad \frac{\Delta \vdash t_1 : A \otimes B \quad \Gamma, x : A, y : B \vdash t_2 : C}{\Delta, \Gamma \vdash \mathbf{let}\ (x, y)\ \mathbf{be}\ t_1\ \mathbf{in}\ t_2 : C}\ \otimes_E$$

**With** $$\frac{\Delta \vdash t_1 : A \quad \Delta \vdash t_2 : B}{\Delta \vdash \langle t_1,\ t_2 \rangle : A \,\&\, B}\ \&_I$$

$$\frac{\Delta \vdash t : A \,\&\, B}{\Delta \vdash \pi_1\, t : A}\ \&_{E1} \qquad \frac{\Delta \vdash t : A \,\&\, B}{\Delta \vdash \pi_2\, t : B}\ \&_{E2}$$

**Sum** $$\frac{\Delta \vdash t : A}{\Delta \vdash \mathbf{inl}(t) : A \oplus B}\ \oplus_{I1} \qquad \frac{\Delta \vdash t : B}{\Delta \vdash \mathbf{inr}(t) : A \oplus B}\ \oplus_{I2}$$

$$\frac{\Delta \vdash t_1 : A \oplus B \quad \Gamma, x : A \vdash t_2 : C \quad \Gamma, y : B \vdash t_3 : C}{\Delta, \Gamma \vdash \mathbf{let}\ (x; y)\ \mathbf{be}\ t_1\ \mathbf{in}\ (t_2; t_3) : C}\ \oplus_E$$

[!] – **Group** $$\frac{!\Delta \vdash t : A}{!\Delta \vdash \mathbf{!}(t) :!A}\ ! \qquad \frac{\Delta \vdash t :!A}{\Delta \vdash \star t : A}\ d$$

$$\frac{\Delta \vdash t_1 :!A \quad \Gamma \vdash t_2 : B}{\Delta \vdash \mathbf{drop}\ t_1\ \mathbf{in}\ t_2 : B}\ w! \qquad \frac{\Delta \vdash t_1 :!A \quad \Gamma, x :!A, y :!A \vdash t_2 : B}{\Delta \vdash \mathbf{copy}\ t_1\ \mathbf{as}\ x/y\ \mathbf{in}\ t_2 : B}\ c!$$

Note that the the counterparts of the structural rules ([w] and [c]) in $\lambda^{\rightarrow}$ are restricted to terms of [!]-type. [!]-typed terms may be introduced by the [!]-rule ('promotion') under the condition, that all assumptions are already [!]-typed. The notation $!\Delta$ stands for contexts containing only [!]-typed assumptions. [!]-operators may be eliminated by the [d]-rule ('dereliction').

In comparison to $\lambda^{\rightarrow}$, weakening and contraction are represented as explicit operations in the derived term.

The rules of this system are similar to the ones introduced by Mackie in [Mac91], the most notable exception being the [d]-rule, which is formulated in his framework as

$$\frac{\Delta \vdash t_1 :!A \quad \Gamma, x : A \vdash t_2 : B}{\Delta, \Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ (\star \mathbf{t_1})\ \mathbf{in}\ \mathbf{t_2} : B}\ d_M$$

The present rule is a simplified notation for

$$\frac{\Delta \vdash t :!A \quad y : A \vdash y : A}{\Delta \vdash \mathbf{let}\ y\ \mathbf{be}\ (\star \mathbf{t})\ \mathbf{in}\ \mathbf{y} : A}\ d_M$$

which is an instance of Mackie's dereliction rule. They both serve exactly the same purpose of eliminating a [!]-operator on the right-hand side of the turnstile.

The present rule is preferred because it has a simpler notation. The [d]-rule used here is identical to the one used by Benton, Bierman et al. [BBHdP92] for their linear natural deduction calculus. Another minor deviation concerns the elimination rules for &-pairs. They have been chosen in analogy to the elimination rules for $\wedge$-pairs in $\lambda^{\rightarrow}$. Again, they serve the same purpose of projecting one component of an additive pair as the corresponding rules in Mackie's system, but are notationally simpler since they do without an explicit substitution into an auxiliary term.

The following aggregate rules will be used in proofs in order to increase readability:

$$\frac{\Delta \vdash t : B}{\Delta, x :!A \vdash \mathbf{drop}\ x\ \mathbf{in}\ t : B}\ w!* \quad \equiv_{df} \quad \frac{x :!A \vdash x :!A \quad \Delta \vdash t : B}{\Delta, x :!A \vdash \mathbf{drop}\ x\ \mathbf{in}\ t : B}\ w!$$

$$\frac{\Delta, x :!A, y :!A \vdash t : B}{\Delta, z :!A \vdash \mathbf{copy}\ z\ \mathbf{as}\ x/y\ \mathbf{in}\ t : B}\ c!* \quad \equiv_{df} \quad \frac{z :!A \vdash z :!A \quad \Delta, x :!A, y :!A \vdash t : B}{\Delta \vdash \mathbf{copy}\ z\ \mathbf{as}\ x/y\ \mathbf{in}\ t : B}\ c!$$

In order to simplify matters further, the convention is adopted that $\lambda^{!,\multimap}$-proofs are identified under the following neutralization rule:

$$\dfrac{\dfrac{\begin{array}{c} p \\ \Delta \vdash t : A \end{array}}{\Delta \vdash !(t) :!A}\ !}{\Delta \vdash \star!(t) : A}\ d \quad \equiv \quad \begin{array}{c} p \\ \Delta \vdash t : A \end{array}$$

The system $\lambda^{!,\multimap}$, unlike $\lambda^{\rightarrow}$, has the following property:

**Lemma 2.11 Benton, Bierman et al.,[BBHdP92]** If a sequent $\Delta \vdash t : A$ may be derived in $\lambda^{!,\multimap}$, then its derivation is unique. $\triangleleft$

This is due to the fact that also the structural operations are now reflected in the term by explicit operators.

### 2.3.2 Extensions

The following paragraphs will introduce the linear counterparts to the programming extensions in $\lambda^{\rightarrow}$. Extending $\lambda^{!,\multimap}$ with lists, let-definitions and recursion is not straightforward but requires a careful account of the alternatives and restrictions that are characteristic to $\lambda^{!,\multimap}$.

In the following we will refrain from giving grammars for types and untyped pre-terms because they should be obvious, and concentrate on the inference rules instead.

#### 2.3.2.1 Lists

Lists in $\lambda^{\rightarrow}$ may be characterized be the recursive type equation

$$(A)\,list \quad \simeq \quad (A)\,list \vee (A \wedge (A)\,list)$$

meaning that they may be built using disjunction and conjunction which are primitive in $\lambda^{\rightarrow}$. In order to derive a similar building principle for lists in $\lambda^{!,\multimap}$, one must look to the primitive operations in $\lambda^{!,\multimap}$ that correspond to disjunction and conjunction. Obviously, there is no choice in $\lambda^{!,\multimap}$ regarding disjunction, but $\lambda^{!,\multimap}$ provides two different kinds of conjunction: the multiplicative ( $\otimes$ ) and the additive conjunction ( $\&$ ). This opens up the possibility for multiplicative and additive lists, which can be formalized as follows:

**Definition 2.12 Lists in $\lambda^{!,\multimap}$**

- Multiplicative lists

$$\frac{}{\vdash [] : (A)^{*}list}\,Nil \qquad \frac{\Delta \vdash t_1 : A \quad \Gamma \vdash t_2 : (A)^{*}list}{\Delta, \Gamma \vdash t_1{*}{*}t_2 : (A)^{*}list}\,Cons$$

$$\frac{\Delta \vdash l : (A)^{*}list \quad \Gamma \vdash t_1 : C \quad \Gamma, h : A, tl : (A)^{*}list \vdash t_2 : C}{\Delta, \Gamma \vdash \textbf{case}\; l = [] \Rightarrow t_1 \mid (h{*}{*}tl) \Rightarrow t_2 : C}\,\otimes - lcase$$

- Additive lists

$$\frac{}{\vdash [] : (A)^{+}list}\,Nil \qquad \frac{\Delta \vdash t_1 : A \quad \Delta \vdash t_2 : (A)^{+}list}{\Delta \vdash t_1{+}{+}t_2 : (A)^{+}list}\,Cons$$

$$\frac{\Delta \vdash l : (A)^{+}list \quad \Gamma \vdash t_1 : C \quad \Gamma, h : A \vdash t_2 : C}{\Delta \vdash \textbf{case}\; l = [] \Rightarrow t_1 \mid (h{+}{+}\_) \Rightarrow t_2 : C}\,\& - lcase_1$$

$$\frac{\Delta \vdash l : (A)^{+}list \quad \Gamma \vdash t_1 : C \quad \Gamma, tl : (A)^{+}list \vdash t_2 : C}{\Delta \vdash \textbf{case}\; l = [] \Rightarrow t_1 \mid (\_{+}{+}tl) \Rightarrow t_2 : C}\,\& - lcase_2$$

$\triangleleft$

Both list-elimination rules may be understood as aggregate inferences composed of a $[\oplus_E]$-inference together with a $[\otimes_E]$- (resp. $[\&_E]-$) inference. Consequently, additive lists may be taken apart by either projecting their head or their tail, while multiplicative lists require both head and tail to be used.

Which type of list is more appropriate depends on the term that is to be derived. For example, the well-known *map* function is best coded in $\lambda^{!,\multimap}$ using multiplicative lists because it uses both the head and the tail of the list argument.

#### 2.3.2.2 Recursion

With regard to recursion this work follows the considerations put forward in [Mac91]. Look at the corresponding rule in $\lambda^{\rightarrow}$:

$$\frac{\Delta, x : A \vdash t : A}{\Delta \vdash \mathbf{fix}(x,t) : A}\ fix$$

It cannot be carried over to $\lambda^{!,\multimap}$ without anticipating the intended (operational) meaning of this term and its consequences with respect to linearity.

Recursion will be interpreted by the rule:

$$\mathbf{fix}(x,t) \rightsquigarrow t[x/\mathbf{fix}(x,t)]$$

where $t[x/\mathbf{fix}(x,t)]$ means substituting $\mathbf{fix}(x,t)$ for $x$. The fact that $t$ is duplicated in this reduction requires $\mathbf{fix}(x,t), t$ and $x$ all to be of [!]-type. This condition is captured by the following rule:

$$\frac{\Delta, x :!A \vdash t :!A}{\Delta \vdash \mathbf{fix}(x,t) :!A}\ fix$$

#### 2.3.2.3 Let-definitions

Let-definitions can be introduced to $\lambda^{!,\multimap}$ straightforwardly by the rule

$$\frac{\Delta \vdash t_1 : A \quad \Gamma, x : A \vdash t_2 : B}{\Delta, \Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ t_1\ \mathbf{in}\ t_2 : B}$$

## 2.4 Constrained Linear Types

The following definitions are concerned with the problem of formalizing a notion of *logical necessity* for [!]- and [d]-inferences in $\lambda^{!,\multimap}$-proofs. They derive from the work of Wadler [Wad91a] and Toenne [T9̈3] on *use-types*, aiming at a formal tool to identify those [!]- and [d]-inferences that are 'implied' by a [w!]- or [c!]-inference. Only those [!]- (resp. [d]-) inferences are to be considered indispensable in a linear proof, while the others may be removed while preserving the proof's correctness. A removal procedure that employs these use-types to 'optimize' $\lambda^{!,\multimap}$-proofs will be presented as an operational alternative to the 'removal theory' proposed in [DJS95].

### 2.4.1 Preliminaries

**Definition 2.13 Propositional use-constraints** Let $\mathcal{U} = \{u_0, u_1, \ldots\}$ be an infinite supply of (use-) variables. The language of boolean use-constraints over these variables is defined by the following grammar:

$$\mathcal{C} ::= u \mid \bot \mid \top \mid (\mathcal{C} \to \mathcal{C}) \mid (\mathcal{C} \wedge \mathcal{C})$$

where $u \in \mathcal{U}$.

Variables range over the set $\mathbf{bool} = \{0, 1\}$ of *truth values*. A *valuation* $\overline{\mathcal{A}}(.) : \mathcal{U} \to \mathbf{bool}$ is a map from variables into the set **bool**. The (canonical) extension of $\overline{\mathcal{A}}(.)$ over constraints $\mathcal{C}$ to a valuation $\mathcal{A}(.) : \mathcal{C} \to \mathbf{bool}$, giving semantics to the different connectives, is defined as

$$\begin{array}{rcl}
\mathcal{A}(u) & = & \overline{\mathcal{A}}(u) \\
\mathcal{A}(\bot) & = & 0 \\
\mathcal{A}(\top) & = & 1 \\
\mathcal{A}(\mathcal{C} \to \mathcal{D}) & = & \begin{cases} 1 & \text{if } \mathcal{A}(\mathcal{C}) = 0 \text{ or } \mathcal{A}(\mathcal{D}) = 1 \\ 0 & \text{otherwise} \end{cases} \\
\mathcal{A}(\mathcal{C} \wedge \mathcal{D}) & = & \begin{cases} 1 & \text{if } \mathcal{A}(\mathcal{C}) = 1 \text{ and } \mathcal{A}(\mathcal{D}) = 1 \\ 0 & \text{otherwise} \end{cases}
\end{array}$$

A constraint $\mathcal{C}$ is said to be *satisfiable* if and only if there exists a valuation $\mathcal{A}(.)$, such that $\mathcal{A}(\mathcal{C}) = 1$. A constraint $\mathcal{D}$ is a *consequence* of constraint $\mathcal{C}$, written as $\mathcal{C} \models \mathcal{D}$ if and only if for all valuations $\mathcal{A}(.)$,

$$\mathcal{A}(\mathcal{C}) = 1 \Rightarrow \mathcal{A}(\mathcal{D}) = 1$$

$\triangleleft$

In order to reduce the number of parentheses in an expression, the convention is adopted that $\wedge$ binds stronger than $\to$.

The following shorthand notation will be used:

$$\bigwedge(\mathcal{C}_1, \ldots, \mathcal{C}_n) =_{df} \mathcal{C}_1 \wedge \ldots \wedge \mathcal{C}_n$$

### 2.4.2 Constraints in $c - \lambda^{!,\multimap}$

Following the idea of Wadler [Wad91a], use-constraints are supposed to reflect properties of types, terms and sequents. The set $\tau_!^c$ of linear *use-types* is defined by the following grammar:

$$\tau_!^c ::= \mathbf{1} \mid !^u \tau_!^c \mid \tau_!^c \multimap \tau_!^c \mid \tau_!^c \otimes \tau_!^c \mid \tau_!^c \,\&\, \tau_!^c \mid \tau_!^c \oplus \tau_!^c$$

where $u \in \mathcal{U}$ a boolean variable.

The derivable judgments in $c - \lambda^{!,\multimap}$ associate a typing environment, a term, a type $\tau_!^c$ and a so called *use-constraint* $\mathcal{C}$. They are written as $\Delta \vdash t : \tau_!^c[C]$ and read 'term $t$ has type $\tau_!^c$ and associated use-constraint $C$ in type environment $\Delta$'.

The set of derivable use-constraints is strictly a subset of the general use-constraints constraints defined above. They are derived together with typing judgements with the intention of relating a promotion inference to a nonlinear usage (by [w!] or [c!]) of the resp. subproof or tell that it is superfluous.

Consider the following nonlinear inference rules for constrained sequents:

$$\frac{\Delta \vdash t_1 :!^u A[\mathcal{D}] \quad \Gamma \vdash t_2 : B[\mathcal{C}]}{\Delta \vdash \mathbf{drop}\, t_1 \,\mathbf{in}\, t_2 : B[u \wedge \mathcal{C} \wedge \mathcal{D}]}\ w!$$
$$\frac{\Delta \vdash t_1 :!^{u_3} A[\mathcal{C}] \quad \Gamma, x :!^{u_1} A, y :!^{u_2} A \vdash t : B[\mathcal{D}]}{\Delta \vdash \mathbf{copy}\, t_1 \,\mathbf{as}\, x/y \,\mathbf{in}\, t_2 : B[u_1 \wedge u_2 \wedge u_3 \wedge \mathcal{C} \wedge \mathcal{D}]}\ c!$$

Obviously, $u \wedge \mathcal{C} \models u$ where u is the use-variable introduced into the constraint by the weakening inference. Also,

$u_1 \wedge u_2 \wedge u_3 \wedge \mathcal{C} \models u_1 \wedge u_2 \wedge u_3$

where the $u_i, i \in \{1, 2, 3\}$ are the use-variables introduced by the $c!$-inference. This means, that the constraints produced by the two inferences entail those use-variables that are attached to *necessary* [!]-operators.

**Remark 2.14 (Uniqueness of use-variables)** Upon their introduction, use-variables are assumed to be unique, distinct from all other use-variables already occuring further below in the proof. ◁

Next consider the constrained [!]- and [d]-rules:

$$\frac{!\Delta \vdash t : A[\mathcal{C}]}{!^\delta \Delta \vdash \mathbf{!}(t)^u :!^u A[(u \rightarrow \delta) \wedge \mathcal{C}]}\ ! \quad \frac{\Delta \vdash t :!^u B[\mathcal{C}]}{\Delta \vdash \star t^u : B[\mathcal{C}]}\ d$$

where $\delta = \bigwedge(u_1, \ldots, u_n) \iff \Delta =!^{u_1}, \ldots, !^{u_n}$.

**Remark 2.15 (Constrained terms)** The astute reader should have noticed that we have attached use-variables to promoted and derelicted *terms* as well. This measure is necessary because constraints are supposed to talk about the structure not only of proofs, but also of terms (sequents). ◁

The constrained [!]-rule transfers the side-condition on contexts of the regular [!]-rule to the level of constraints without predetermining the *factual necessity* of the [!]-operators on both sides of the turnstile:

$u \to (u_1 \wedge \ldots \wedge u_n) \not\models u \wedge u_1 \wedge \ldots \wedge u_n$ but $u \wedge u \to (u_1 \wedge \ldots \wedge u_n) \models u \wedge u_1 \wedge \ldots \wedge u_n$

This means that only if the precondition of the promotion-constraint is added to the constraint (exclusively due to [w!] or [c!]) does the [!]-inference and its nonlinear-context requirement become necessary and effective.

The remaining inferences do not introduce additional, but merely combine existing constraints:

**Definition 2.16** ($c - \lambda^{!,\multimap}$)

The set of derivable constrained judgments is defined as the least relation closed under the union of the above mentioned constrained nonlinear inferences ([d],[!],[w!],[c!]) and the following axioms and rules:

- Core system

$$\frac{}{x : A \vdash x : A[\top]}\ axiom(A \in \tau_!)$$

$$\frac{}{\vdash \bullet[\top] : \mathbf{1}}\ \mathbf{1}_I \qquad \frac{\Delta \vdash t_1 : \mathbf{1}[\mathcal{C}] \quad \Gamma \vdash t_2 : A[\mathcal{D}]}{\Delta, \Gamma \vdash \mathbf{let}\ ()\ \mathbf{be}\ t_1\ \mathbf{in}\ t_2 : A[\mathcal{C} \wedge \mathcal{D}]}\ \mathbf{1}_E$$

$$\frac{\Delta, x : A \vdash t : B[\mathcal{C}]}{\Delta \vdash \lambda x.t : A \multimap B[\mathcal{C}]}\ \multimap_I \qquad \frac{\Delta \vdash u : A[\mathcal{C}] \quad \Gamma \vdash t : A \multimap B[\mathcal{D}]}{\Delta, \Gamma \vdash (t\ u) : B[\mathcal{C} \wedge \mathcal{D}]}\ \multimap_E$$

$$\frac{\Delta \vdash t_1 : A[\mathcal{C}] \quad \Gamma \vdash t_2 : B[\mathcal{D}]}{\Delta, \Gamma \vdash (t_1, t_2) : A \otimes B[\mathcal{C} \wedge \mathcal{D}]}\ \otimes_I \qquad \frac{\Delta \vdash t_1 : A \otimes B[\mathcal{C}] \quad \Gamma, x : A, y : B \vdash t_2 : C[\mathcal{D}]}{\Delta, \Gamma \vdash \mathbf{let}\ (x, y)\ \mathbf{be}\ t_1\ \mathbf{in}\ t_2 : C[\mathcal{C} \wedge \mathcal{D}]}\ \otimes_E$$

$$\frac{\Delta \vdash t_1 : A[\mathcal{C}] \quad \Delta \vdash t_2 : B[\mathcal{D}]}{\Delta \vdash \langle t_1,\ t_2 \rangle : A \,\&\, B[\mathcal{C} \wedge \mathcal{D}]}\ \&_I$$

$$\frac{\Delta \vdash t : A \,\&\, B[\mathcal{C}]}{\Delta \vdash \pi_1\, t : A[\mathcal{C}]}\ \&_{E1} \qquad \frac{\Delta \vdash t : A \,\&\, B[\mathcal{C}]}{\Delta \vdash \pi_2\, t : B[\mathcal{C}]}\ \&_{E2}$$

$$\frac{\Delta \vdash t : A[\mathcal{C}]}{\Delta \vdash \mathbf{inl}(t) : A[\mathcal{C}] \oplus B}\ \oplus_{I1} \qquad \frac{\Delta \vdash t : B[\mathcal{C}]}{\Delta \vdash \mathbf{inr}(t) : A[\mathcal{C}] \oplus B}\ \oplus_{I2}$$

$$\frac{\Delta \vdash t_1 : A \oplus B[\mathcal{C}] \quad \Gamma, x : A \vdash t_2 : C[\mathcal{D}] \quad \Delta, y : B \vdash t_3 : C[\mathcal{E}]}{\Delta \vdash \mathbf{let}\ (x; y)\ \mathbf{be}\ t_1\ \mathbf{in}\ (t_2; t_3) : C[\mathcal{C} \wedge \mathcal{D} \wedge \mathcal{E}]}\ \oplus_E$$

- Extensions

  Constraint assignment for lists and let-definitions is straightforward and will therefore be omitted. The case of recursion is more interesting. Since the nonlinearity requirements of the $[fix]$-rule are due to extra-logical rather than logical, i.e. due to dynamic rather than static properties of the resp. term, they may not generally be related to a structural source (w! or c!). The resp. use-variables decorating the [!]-operators occurring in the rule's premise and conclusion are therefore not generally implied by the derived constraint, although they definitely should be due to the above considerations. This recommends adopting

$$\frac{\Delta, x :!^u A \vdash t :!^u A[\mathcal{C}]}{\Delta \vdash \mathbf{fix}(x,t) :!^u A[u \wedge \mathcal{C}]}$$

as a rule for the introduction of recursion, which simply adds the resp. use-variable to the constraint.

$\triangleleft$

Note that modulo constraints, $c-\lambda^{!,\multimap}$ derives the same sequents as $\lambda^{!,\multimap}$. The abbreviations for [w!*] and [c!*] in $\lambda^{!,\multimap}$ are carried over to $c-\lambda^{!,\multimap}$ accordingly.

As an example, consider the following constrained typing proof of the term

$$(\lambda x.\mathbf{copy}\ x\ \mathbf{as}\ x_1/x_2\ \mathbf{in}\ \mathbf{drop}\ x_1\ \mathbf{in}\ \mathbf{drop}\ x_2\ \mathbf{in}\ ()\ \mathbf{!}(()))$$

**Example 2.17 (Constrained derivations)**

$$\dfrac{\dfrac{\dfrac{}{\vdash () : \mathbf{1}}\,\mathbf{1}_I}{\vdash \mathbf{!}(()) :!^u\mathbf{1}}\,arg \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash () : \mathbf{1}}\,\mathbf{1}_I}{x_2 :!^{u_2}\mathbf{1} \vdash \mathbf{drop}\ x_2\ \mathbf{in}\ () : \mathbf{1}[u_2]}\,w!*}{x_1 :!^{u_1}\mathbf{1}, x_2 :!^{u_2}\mathbf{1} \vdash \mathbf{drop}\ x_1\ \mathbf{in}\ \mathbf{drop}\ x_2\ \mathbf{in}\ () : \mathbf{1}[u_1 \wedge u_2]}\,w!*}{x :!^u\mathbf{1} \vdash \mathbf{copy}\ x\ \mathbf{as}\ x_1/x_2\ \mathbf{in}\ \mathbf{drop}\ x_1\ \mathbf{in}\ \mathbf{drop}\ x_2\ \mathbf{in}\ () : \mathbf{1}[u_1 \wedge u_2 \wedge u]}\,c!*}{\vdash \lambda x.\mathbf{copy}\ x\ \mathbf{as}\ x_1/x_2\ \mathbf{in}\ \mathbf{drop}\ x_1\ \mathbf{in}\ \mathbf{drop}\ x_2\ \mathbf{in}\ () :!^u\mathbf{1}\multimap\mathbf{1}[u_1 \wedge u_2 \wedge u]}\,fn}{\vdash (\lambda x.\mathbf{copy}\ x\ \mathbf{as}\ x_1/x_2\ \mathbf{in}\ \mathbf{drop}\ x_1\ \mathbf{in}\ \mathbf{drop}\ x_2\ \mathbf{in}\ ()\ \mathbf{!}(())) : \mathbf{1}[u_1 \wedge u_2 \wedge u]}\,app$$

Then

$$u_1 \wedge u_2 \wedge u \models u$$

indicates that the promotion inference is necessary. $\triangleleft$

## 2.5 Minimization

The information that distinguishes necessary from superfluous [!]-inferences may be used to 'minimize' a given proof, removing those [!]-inferences whose use-variable is not implied by the associated constraint. Such a proof-minimization procedure based on constraint entailment will be defined incrementally in the following paragraphs, by passing from type- and term-minimization, minimization of environments and sequents, to proofs. It may be seen as a practical alternative to the kind of removal theory put forward in [DJS95].

**Definition 2.18 Minimization procedure** $\mu_!(.)$ Assume that $\mathcal{C}$ is a (derivable) constraint.

1. $\mu_!(.) : \tau_!^c \to \tau_!$, minimization of constrained types is defined as:

$$
\begin{array}{rcl}
\mu_!(\mathbf{1}) & = & \mathbf{1} \\
\mu_!(A) & = & A \text{ if } A \text{ atomic} \\
\mu_!(!^u A) & = & \begin{cases} !\mu_!(A) & \text{if } \mathcal{C} \models u \\ \mu_!(A) & \text{otherwise} \end{cases} \\
\mu_!(A \otimes B) & = & \mu_!(A) \otimes \mu_!(B) \\
\mu_!(A \,\&\, B) & = & \mu_!(A) \,\&\, \mu_!(B) \\
\mu_!(A \oplus B) & = & \mu_!(A) \oplus \mu_!(B) \\
\mu_!(A \multimap B) & = & \mu_!(A) \multimap \mu_!(B)
\end{array}
$$

2. $\mu_!(.) : (c - \lambda^{!,\multimap}) \to \lambda^{!,\multimap}$ is defined as:

$$
\begin{array}{rcl}
\mu_!(x) & = & x \\
\mu_!(()) & = & () \\
\mu_!(\mathbf{let}\ ()\ \mathbf{be}\ t_1\ \mathbf{in}\ t_2) & = & \mathbf{let}\ ()\ \mathbf{be}\ \mu_!(t_1)\ \mathbf{in}\ \mu_!(t_2) \\
\mu_!(\lambda x.t) & = & \lambda x.\mu_!(t) \\
\mu_!(\mathbf{let}\ x\ \mathbf{be}\ (t_1\, u)\ \mathbf{in}\ t_2) & = & \mathbf{let}\ x\ \mathbf{be}\ (\mu_!(t_1)\, \mu_!(u))\ \mathbf{in}\ \mu_!(t_2) \\
\mu_!((t, u)) & = & (\mu_!(t), \mu_!(u)) \\
\mu_!(\mathbf{let}\ (x, y)\ \mathbf{be}\ t_1\ \mathbf{in}\ t_2) & = & \mathbf{let}\ (x, y)\ \mathbf{be}\ \mu_!(t_1)\ \mathbf{in}\ \mu_!(t_2) \\
\mu_!(\langle t,\ u\rangle) & = & \langle \mu_!(t),\ \mu_!(u)\rangle \\
\mu_!(\pi_1\, t) & = & \pi_1\, \mu_!(t) \\
\mu_!(\pi_2\, t) & = & \pi_2\, \mu_!(t) \\
\mu_!(\mathbf{inl}(t)) & = & \mathbf{inl}(\mu_!(t)) \\
\mu_!(\mathbf{inr}(t)) & = & \mathbf{inr}(\mu_!(t)) \\
\mu_!(\mathbf{let}\ (x; y)\ \mathbf{be}\ t\ \mathbf{in}\ (t_1; t_2)) & = & \mathbf{let}\ (x; y)\ \mathbf{be}\ \mu_!(t)\ \mathbf{in}\ (\mu_!(t_1); \mu_!(t_2)) \\
\mu_!(\mathbf{!}(t)^u) & = & \begin{cases} \mathbf{!}(\mu_!(t)) & \text{if } \mathcal{C} \models u \\ \mu_!(t) & \text{otherwise} \end{cases} \\
\mu_!(\star t^u) & = & \begin{cases} \star\mu_!(t) & \text{if } \mathcal{C} \models u \\ \mu_!(t) & \text{otherwise} \end{cases} \\
\mu_!(\mathbf{copy}\ t_1\ \mathbf{as}\ x/y\ \mathbf{in}\ t_2) & = & \mathbf{copy}\ \mu_!(t_1)\ \mathbf{as}\ x/y\ \mathbf{in}\ \mu_!(t_2) \\
\mu_!(\mathbf{drop}\ t_1\ \mathbf{in}\ t_2) & = & \mathbf{drop}\ \mu_!(t_1)\ \mathbf{in}\ \mu_!(t_2)
\end{array}
$$

The crucial cases are the ones for promotion and dereliction. We can see now why it was necessary to attach use-variables to terms as well. Only if the respective constraint entails the use-variables associated with a particular nonlinear operator is it left in its place. Otherwise, minimization removes it.

3. Type assignments are minimized as $\mu_!(t : A) = (\mu_!(t) : \mu_!(A))$. This is extended in an obvious manner over environments $\Delta = \{t_1 : A_1, \ldots, t_n : A_n\}$ and

4. (constrained) sequents: $\mu_!(\Delta \vdash t : A[\mathcal{C}]) = \mu_!(\Delta) \vdash \mu_!(t) : \mu_!(A)$

5. Wherever the constraint $\mathcal{C}$ with respect to which minimization takes place is of particular interest, it will be indicated through a subscript, as in $\mu_{!\mathcal{C}}(.)$.

$\triangleleft$

**Remark 2.19 (Minimization with $\perp$)** $\mu_{!\perp}(.)$ with respect to constraint $\perp$ leaves all [!]- resp. [d]-operators in place, removing none since $\forall u.\perp \models u$. This implies that $\forall p \in c - \lambda^{!,\multimap}.\mu_{!\perp}(p) \in \lambda^{!,\multimap}$ since $\mu_{!\perp}(p)$ is the identity on terms.

Note that the constraint $\perp$ is never derivable in $c-\lambda^{!,\multimap}$. We will use it merely as a handy tool in subsequent arguments. $\triangleleft$

**Remark 2.20 (Overloading)** Note that $\mu_!(.)$ is defined to encompass minimization of objects from different syntactic categories. Furthermore, every term in $\lambda^{!,\multimap}$ may be assigned a unique constrained term in $c - \lambda^{!,\multimap}$, as well as every type in $\mathcal{T}_!$ may be assigned a unique constrained type in $\mathcal{T}_!^c$ in an obvious manner. Therefore, $\mu_!(.)$ may equally well be extended to range over $\lambda^{!,\multimap}$ and $\mathcal{T}_!$. $\triangleleft$

Most importantly, derivable sequents $\Delta \vdash t : A[\mathcal{C}] \in c-\lambda^{!,\multimap}$, when minimized with respect to their associated constraint $\mathcal{C}$ yield derivable sequents in $\lambda^{!,\multimap}$, that is $\mu_{\mathcal{C}}(.)$ preserves provability:

**Theorem 2.21 Soundness** $\forall \Delta \vdash t : A[\mathcal{C}] \in c - \lambda^{!,\multimap}.\mu_{!\mathcal{C}}(\Delta \vdash t : A) \in \lambda^{!,\multimap}$ $\triangleleft$

Besides proving the above theorem, the following demonstrates how $\mu_!(.)$ can be overloaded further to additionally range over proofs in $c - \lambda^{!,\multimap}$ (or $\lambda^{!,\multimap}$ respectively):

**Proof Thm.2.21** By induction on a derivation in $c - \lambda^{!,\multimap}$. Suppose the last inference was

- $\dfrac{}{x : A \vdash x : A[]}\ axiom$

  Obviously the proposition holds.

- $\dfrac{\Delta, x : A \vdash t : B[\mathcal{C}]}{\Delta \vdash \lambda x.t : (A \multimap B)[\mathcal{C}]}\ \multimap_I$

  Then

$$\mu_{!\mathcal{C}}(\Delta, x : A \vdash t : B[\mathcal{C}]) \in \lambda^{!,\multimap}$$

by the induction hypothesis and

$$\mu_{!\mathcal{C}}(\Delta \vdash \lambda x.t : A \multimap B) \in \lambda^{!,\multimap}$$

follows from the $[\multimap_E]$-rule.

- $$\frac{\Delta \vdash u : A[\mathcal{C}] \quad \Gamma \vdash t : A \multimap B[\mathcal{D}]}{\Delta, \Gamma \vdash (t\, u) : B[\mathcal{C} \wedge \mathcal{D}]} \multimap_E$$

  By the induction hypothesis:

  $$\mu_{!\mathcal{C}}(\Delta \vdash u : A[\mathcal{C}])$$

  and

  $$\mu_{!\mathcal{D}}(\Gamma \vdash t : A \multimap B[\mathcal{D}])$$

  are both in $\lambda^{!,\multimap}$. Now

  $$\mu_{!\mathcal{C}\wedge\mathcal{D}}(\Delta \vdash u : A[\mathcal{C}]) \text{ and } \mu_{!\mathcal{C}\wedge\mathcal{D}}(\Gamma \vdash t : A \multimap B[\mathcal{D}])$$

  necessitate at least as many or more [d]- and [!]-inferences, due to the fact that joining (derivable) constraints in a $c - \lambda^{!,\multimap}$-derivation *monotonically* increases the number of implied use-variables. Adding the respective [!]- and [d]-inferences does not corrupt soundness with respect to provability because

  $$\Delta \vdash u : A \text{ and } \Gamma \vdash t : A \multimap B$$

  are certainly in $\lambda^{!,\multimap}$ as well. By the $[\multimap_E]$-rule it follows that

  $$\mu_{!\mathcal{C}\wedge\mathcal{D}}(\Delta, \Gamma \vdash (t\, u) : [\mathcal{C} \wedge \mathcal{D}]) \in \lambda^{!,\multimap}$$

  The other cases are similar and will therefore be omitted. Note that the cases for promotion and dereliction follow from simple applications of the induction hypothesis and the fact that the derived constraint can never imply them *at the point of their introduction.*

$\square$

Minimization $\mu_!(.)$ may be mapped over substitutions as $\mu_!(t[x/u]) = \mu_!(t[x/\mu_!(u)])$.

### 2.5.1 Properties

Consider the following inference rule, which is not contained in the rule sets for $\lambda^{\rightarrow}$ or $\lambda^{!,\multimap}$:

$$\frac{\Delta \vdash t : A \quad \Gamma, x : A \vdash u : B}{\Delta, \Gamma \vdash u[x/t] : B}$$

It is only valid if the set of sequents that are derivable using only the original rules is closed under substitution. In $\lambda^{\rightarrow}$, the following is a standard result (cf. [Mit90]).

**Lemma 2.22 Substitution property for $\lambda^{\rightarrow}$** Let $\Delta \vdash t : A$ and $\Gamma, x : A \vdash u : B$ be derivable in $\lambda^{\rightarrow}$. Then $\Delta, \Gamma \vdash u[x/t] : B$ is derivable

in $\lambda^{\rightarrow}$. ◁

It is provable by induction on the generation of $\Gamma, x : A \vdash u : B$

The observation that the same does not hold for $\lambda^{!,\multimap}$ is due to Benton, Bierman et al.[BBHdP92]:

**Example 2.23 (Failure of the substitution property)** Consider two derivable sequents

$!\Delta, z : (A \otimes B) \vdash \mathbf{let}\ (x, y)\ \mathbf{be}\ z\ \mathbf{in}\ \mathbf{!}(u) :!A$ and

$!\Gamma, x :!A \vdash \mathbf{!}(t) :!B$.

Observe that the type of $z$ is *linear*. Admissibility of the substitution rule would require that the sequent

$\Delta, \Gamma, z : (A \otimes B) \vdash \mathbf{!}(u[x/\mathbf{let}\ (x, y)\ \mathbf{be}\ z\ \mathbf{in}\ \mathbf{!}(t)])$

is derivable in $\lambda^{!,\multimap}$. This however is not possible, since obviously, the last inference in the derivation would have to be a [!]-inference. This in turn requires that the entire context is *nonlinear*, including the assumption on $z$. A similar example is provided by P.Wadler in [Wad91b]. ◁

The above counterexample hinges on the fact that substitution may act counter to the context constraint of the [!]-inference, smuggling a linear variable into a context that must be entirely nonlinear. An obvious restriction yields the following:

**Lemma 2.24 Substitution property for closed terms** Let $\vdash u : A$ and $\Delta, x : A \vdash t : B$ be derivable in $\lambda^{!,\multimap}$. Then

$\Delta \vdash u[x/t] : B$

is derivable in $\lambda^{!,\multimap}$. ◁

The proof is by induction on the generation of $\Delta, x : A \vdash t : B$. This restricted property will suffice entirely for the purposes of the present work.

**Remark 2.25 (Alternative)** There may be situations however, where the general substitution property is indispensable, e.g. in proving equivalence with a sequent calculus formulation of Linear Logic type assignment (cf. [BBHdP92], sec.5 and [LM92]) or for proving the correctness of proof-normalization in a Natural Deduction based calculus.

Benton, Bierman et al. (cf. [BBHdP92]) have devised an alternative [!]-inference rule in order to remedy the failure of the substitution property:

$$\frac{!\Gamma_1 \vdash u_1 :!A_1 \quad \dots \quad \Gamma_n \vdash u_n :!A_n \quad !\Delta, x_1 :!A_1, \dots, x_n :!A_n \vdash t : A}{!\Delta, !\Gamma_1, \dots, !\Gamma_n \vdash \mathbf{let}\ \vec{x}\ \mathbf{be}\ \vec{u}\ \mathbf{in}\ \mathbf{!}(t) :!A}\ !_B$$

It prevents eventual substitutions from propagating into the scope of the newly introduced [!]-operator by making them explicit. With this rule the substitution property is valid in its general form (cf. [BBHdP92],sec.4).

Since the following chapters will consider only a very limited notion of term (proof) normalization where critical cases do not arise, the simpler termform $\mathbf{!}(t)$ will be adopted in this work. ◁

## 2.6 Related work

Linear type assignment systems, based both on Natural Deduction and Sequent Calculus formalizations have been studied in [Abr93, Mac91, BBHdP92, LM92, Wad91a, Wad93, T9̈3, Tro95].

Mackie's Thesis [Mac91] contains a lucid account of different programming extensions that may be added to a linear language. This work adopts his typing rule for recursion. Additive and multiplicative lists derive from our own considerations.

The idea of use-types has been put forward by Wadler [Wad91a] in the pursuit of a notion of most general linear type for $\lambda^{\rightarrow}$-terms. We have modified his approach to suit our own purposes. The main intention was to devise - in the form of $\mu_!(.)$ - an 'implementation' of the removal theory proposed in [DJS95].

# Chapter 3

# Reduction

The presentation in the previous chapter has been entirely syntactical, specifying how terms are *built* but not how they should be *interpreted.* This chapter will not provide a domain-, or category-theoretic model to $\lambda^{\rightarrow}$- and $\lambda^{!,\multimap}$-terms since all relevant problems considered in this work may be posed and solved by less abstract means. Instead, the meaning of a term - if it exists - is taken to be the normal form that may be obtained by rewriting (reducing) the term according to a given set of rules. Therefore, terms will just denote other terms.

A binary relation over terms, written as $t \Downarrow t'$, will be used to associate (closed) terms $t$ and denotations $t'$. It will be specified by a finite set of rules having the form

$$\frac{t_1 \Downarrow t'_1 \quad \ldots \quad t_n \Downarrow t'_n}{t \Downarrow t'}$$

which may be read as

'The meaning of $t$ is $t'$ if the meaning of $t_1$ is $t'_1$, ...and the meaning of $t_n$ is $t'_n$.'

An alternative, equally possible reading is

'$t$ reduces to $t'$' if $t_1$ reduces to $t'_1$, ...and $t_n$ reduces to $t'_n$

Rules with empty premises are called the axioms of the inference system. These rules define the 'canonical forms of terms under reduction.

Superscripts are used to distinguish different notions of reduction.

This approach to specifying the meanings of terms is known under the name of 'Natural (Operational) Semantics'. The reader is referred to [Kah87] for an overview.

The material presented in the following sections is taken from the seminal article of Abramsky [Abr93].

## 3.1 Reduction semantics of $\lambda^{\rightarrow}$

The two most prominent notions of reduction for functional programs, call-by-value ($\Downarrow^V$) and call-by-name ($\Downarrow^N$), are both relevant to this work and will thus be defined here. The call-by-x terminology refers to the alternative treatments of the argument in a function call, which constitutes the major difference between $\Downarrow^V$ and $\Downarrow^N$:

- Call-by-value reduces an argument to a value *before* it is substituted into the function body - hence the name - while
- Call-by-name substitutes the argument without prior evaluation.

This will be reflected in the following definitions.

### 3.1.1 Call-by-value

**Definition 3.1 Call-by-value reduction $\Downarrow^V$**

$\Downarrow^V \subset \lambda_0^{\rightarrow} \times \lambda_0^{\rightarrow}$ is the smallest relation closed under the following axioms and rules:

$$\frac{}{\lambda x.t \Downarrow^V \lambda x.t} \qquad \frac{t \Downarrow^V \lambda x.t' \quad u \Downarrow^V u' \quad t'[x/u'] \Downarrow^V c}{(t\,u) \Downarrow^V c}$$

$$\frac{t \Downarrow^V t' \quad u \Downarrow^V u'}{(t, u) \Downarrow^V (t', u')} \qquad \frac{t \Downarrow^V (t_1, t_2) \quad t_1 \Downarrow^V c}{\pi_1\, t \Downarrow^V c} \qquad \frac{t \Downarrow^V (t_1, t_2) \quad t_2 \Downarrow^V c}{\pi_2\, t \Downarrow^V c}$$

$$\frac{t \Downarrow^V t'}{\mathbf{inl}(t) \Downarrow^V \mathbf{inl}(t')} \qquad \frac{t \Downarrow^V \mathbf{inl}(t') \quad t_1[x/t'] \Downarrow^V c}{\mathbf{let}\ (x; y)\ \mathbf{be}\ t\ \mathbf{in}\ (t_1; t_2) \Downarrow^V c}$$

$$\frac{t \Downarrow^V t'}{\mathbf{inr}(t) \Downarrow^V \mathbf{inr}(t')} \qquad \frac{t \Downarrow^V \mathbf{inr}(t') \quad t_2[y/t'] \Downarrow^V c}{\mathbf{let}\ (x; y)\ \mathbf{be}\ t\ \mathbf{in}\ (t_1; t_2) \Downarrow^V c}$$

$\triangleleft$

### 3.1.2 Call-by-name

**Definition 3.2 Call-by-name reduction $\Downarrow^N$**

$\Downarrow^N \subset \lambda_0^{\rightarrow} \times \lambda_0^{\rightarrow}$ is the smallest relation closed under the following axioms and rules:

$$\frac{}{\lambda x.t \Downarrow^V \lambda x.t} \qquad \frac{t \Downarrow^V \lambda x.t' \quad t'[x/u] \Downarrow^V c}{(t\,u) \Downarrow^V c}$$

$$\frac{}{(t,u) \Downarrow^V (t,u)} \qquad \frac{t \Downarrow^V (t_1,t_2) \quad t_1 \Downarrow^V c}{\pi_1\, t \Downarrow^V c} \qquad \frac{t \Downarrow^V (t_1,t_2) \quad t_2 \Downarrow^V c}{\pi_2\, t \Downarrow^V c}$$

$$\frac{}{\mathbf{inl}(t) \Downarrow^V \mathbf{inl}(t)} \qquad \frac{t \Downarrow^V \mathbf{inl}(t') \quad t_1[x/t'] \Downarrow^V c}{\mathbf{let}\ (x;y)\ \mathbf{be}\ t\ \mathbf{in}\ (t_1;t_2) \Downarrow^V c}$$

$$\frac{}{\mathbf{inr}(t) \Downarrow^V \mathbf{inr}(t)} \qquad \frac{t \Downarrow^V \mathbf{inr}(t') \quad t_2[y/t'] \Downarrow^V c}{\mathbf{let}\ (x;y)\ \mathbf{be}\ t\ \mathbf{in}\ (t_1;t_2) \Downarrow^V c}$$

◁

Note that both notions of reduction take *closed* terms to *closed* terms.

### 3.1.3 Operational semantics of programming extensions in $\lambda^{\rightarrow}$

In addition to the basic rules above, the following rules specify how the programming extensions are to be interpreted:

**Definition 3.3 Call-by-value reduction of extensions**

$$\frac{t_1 \Downarrow^V t' \quad t_2[x/t'] \Downarrow^V c}{\mathbf{let}\ x\ \mathbf{be}\ t_1\ \mathbf{in}\ t_2 \Downarrow^V c} \qquad \frac{t[x/\mathbf{fix}(x,t)] \Downarrow^V c}{\mathbf{fix}(x,t) \Downarrow^V c}$$

$$\frac{}{[] \Downarrow^V []} \qquad \frac{h \Downarrow^V h' \quad tl \Downarrow^V tl'}{h{::}tl \Downarrow^V h'{::}tl'}$$

$$\frac{l \Downarrow^V [] \quad t_1 \Downarrow^V c}{\mathbf{case}\ l = [] \Rightarrow t_1 \mid (x{::}y) \Rightarrow t_2 \Downarrow^V c} \qquad \frac{l \Downarrow^V h{::}tl \quad t_2[x/h, y/tl] \Downarrow^V c}{\mathbf{case}\ l = [] \Rightarrow t_1 \mid (x{::}y) \Rightarrow t_2 \Downarrow^V c}$$

◁

**Definition 3.4 Call-by-name reduction of extensions**

$$\frac{t_2[x/t_1] \Downarrow^V c}{\mathbf{let}\ x\ \mathbf{be}\ t_1\ \mathbf{in}\ t_2 \Downarrow^V c} \qquad \frac{t[x/\mathbf{fix}(x,t)] \Downarrow^V c}{\mathbf{fix}(x,t) \Downarrow^V c}$$

$$\frac{}{[] \Downarrow^V []} \qquad \frac{}{h{::}tl \Downarrow^V h{::}tl}$$

$$\frac{l \Downarrow^V [] \quad t_1 \Downarrow^V c}{\mathbf{case}\ l = [] \Rightarrow t_1 \mid (x{::}y) \Rightarrow t_2 \Downarrow^V c} \qquad \frac{l \Downarrow^V h{::}tl \quad t_2[x/h, y/tl] \Downarrow^V c}{\mathbf{case}\ l = [] \Rightarrow t_1 \mid (x{::}y) \Rightarrow t_2 \Downarrow^V c}$$

◁

## 3.2 $\lambda^{!,\multimap}$

The operators of linear functional programs ($\lambda_0^{!,\multimap}$) whose reduction semantics shall now be defined, are related to, but differ significantly from the operators found in functional programs ($\lambda_0^{\rightarrow}$).

### 3.2.1 Basic operators

Abstractions will be delayed as under $\Downarrow^N$ or $\Downarrow^V$. Eager and lazy reduction are two equally reasonable options for the execution of applications. Whenever the argument in the application is discarded in the body of the applied function, lazy reduction performs more efficiently. Conversely, eager reduction is more efficient if the argument is duplicated.

In $\lambda^{!,\multimap}$, discarding and duplication of terms require that the respective term is wrapped by a [!]-operator (promotion). As was said before, the first case would recommend halting (delaying) reduction at this point while the second case would recommend the opposite in order not to duplicate the computation. However, w-ether a term is duplicated or discarded may not be recognized in the present system. In order to do so one would need two different kinds of storage operators.

This lack of information hinders the adoption of a more flexible approach to the reduction of promoted terms. Consequently, they will be delayed uniformly. This uniform approach inherits the inefficiencies of either $\Downarrow^V$ or $\Downarrow^N$ as discussed above, but at the present stage we cannot do better.

Multiplicative and additive pairs differ mainly in the way that their components are projected: the destruction of a multiplicative pair uses both components, while destruction of an additive pair uses only one while discarding the other. Regarding efficiency, this indicates that evaluation of a multiplicative pair can go ahead reducing the components eagerly, while evaluation of the additive pair should leave the sub-terms untouched (cf. the discussion in [Abr93]).

These considerations are reflected in the following

**Definition 3.5 (Abramsky,[Abr93])** $\Downarrow^M \subset (\lambda_0^{!,\multimap} \times \lambda_0^{!,\multimap})$ is the smallest relation closed under the following axioms and rules:

$$\frac{}{\lambda x.t \Downarrow^M \lambda x.t} \qquad \frac{t_1 \Downarrow^M \lambda x.t \quad t_2 \Downarrow^M u \quad t[x/u] \Downarrow^M c}{(t_1\, t_2) \Downarrow^M c}$$

$$\frac{t \Downarrow^M t' \quad u \Downarrow^M u'}{(t, u) \Downarrow^M (t', u')} \qquad \frac{t \Downarrow^M (t_1, t_2) \quad t'[x/t_1, y/t_2] \Downarrow^M c}{\mathbf{let}\ (x, y)\ \mathbf{be}\ t\ \mathbf{in}\ t' \Downarrow^M c}$$

$$\frac{}{\langle t,\, u\rangle \Downarrow^M \langle t,\, u\rangle} \qquad \frac{t \Downarrow^M \langle t_1,\, t_2\rangle \quad t'[x/t_1] \Downarrow^M c}{\pi_1\, t \Downarrow^M c} \qquad \frac{t \Downarrow^M \langle t_1,\, t_2\rangle \quad t'[y/t_2] \Downarrow^M c}{\pi_2\, t \Downarrow^M c}$$

$$\frac{t \Downarrow^M t'}{\mathbf{inl}(t) \Downarrow^M \mathbf{inl}(t')} \qquad \frac{t \Downarrow^M \mathbf{inl}(t') \quad t_1[x/t'] \Downarrow^M c}{\mathbf{let}\ (x,y)\ \mathbf{be}\ t\ \mathbf{in}\ (t_1,t_2) \Downarrow^M c}$$

$$\frac{t \Downarrow^M t'}{\mathbf{inr}(t) \Downarrow^M \mathbf{inr}(t')} \qquad \frac{t \Downarrow^M \mathbf{inr}(t') \quad t_2[y/t'] \Downarrow^M c}{\mathbf{let}\ (x,y)\ \mathbf{be}\ t\ \mathbf{in}\ (t_1,t_2) \Downarrow^M c}$$

$$\frac{}{\mathbf{!}(t) \Downarrow^M \mathbf{!}(t)} \qquad \frac{t \Downarrow^M \mathbf{!}(t') \quad t' \Downarrow^M c}{\star t \Downarrow^M c}$$

$$\frac{t_1 \Downarrow^M \mathbf{!}(t_1') \quad t_2 \Downarrow^M c}{\mathbf{drop}\ t_1\ \mathbf{in}\ t_2 \Downarrow^M c} \qquad \frac{t_1 \Downarrow^M \mathbf{!}(t_1') \quad t_2[x/\mathbf{!}(t_1'), y/\mathbf{!}(t_1')] \Downarrow^M c}{\mathbf{copy}\ t_1\ \mathbf{as}\ x/y\ \mathbf{in}\ t_2 \Downarrow^M c}$$

$\triangleleft$

### 3.2.2 Extensions

The operational semantics of let-definitions, lists and recursion are given as

**Definition 3.6** **$\Downarrow^M$-semantics of programming extensions in $\lambda^{!,\multimap}$**

- **Lists**

$$\frac{}{[] \Downarrow^M []}$$

$$\frac{h \Downarrow^M h' \quad tl \Downarrow^M tl'}{h{*}{*}tl \Downarrow^M h'{*}{*}tl'} \qquad \frac{}{h{+}{+}tl \Downarrow^M h{+}{+}tl}$$

$$\frac{l \Downarrow^M [] \quad t_1 \Downarrow^M c}{\mathbf{case}\ l = [] \Rightarrow t_1 \mid (h{*}{*}tl) \Rightarrow t_2 \Downarrow^M c} \qquad \frac{l \Downarrow^M t_3{*}{*}t_4 \quad t_2[h/t_3, tl/t_4] \Downarrow^M c}{\mathbf{case}\ l = [] \Rightarrow t_1 \mid (h{*}{*}tl) \Rightarrow t_2 \Downarrow^M c}$$

$$\frac{l \Downarrow^M t_1{+}{+}t_2 \quad t[h/t_1, tl/t_2] \Downarrow^M c}{\mathbf{case}\ l = [] \Rightarrow h \mid (t{+}{+}\_) \Rightarrow \Downarrow^M c} \qquad \frac{l \Downarrow^M t_1{+}{+}t_2 \quad t[h/t_1, tl/t_2] \Downarrow^M c}{\mathbf{case}\ l = [] \Rightarrow tl \mid (\_{+}{+}t) \Rightarrow \Downarrow^M c}$$

- **Let**

$$\frac{t_1 \Downarrow^M t_1' \quad t_2[x/t_1'] \Downarrow^M c}{\mathbf{let}\ x\ \mathbf{be}\ t_1\ \mathbf{in}\ t_2 \Downarrow^M c}$$

Note that this interpretation of let corresponds to the call-by-value interpretation of the corresponding operator in $\lambda_0^{\rightarrow}$, so it is not just substitution. It validates the well-known condition

$$(\lambda x.t\,u) \Downarrow^M t' \iff \mathbf{let}\ x\ \mathbf{be}\ u\ \mathbf{in}\ t \Downarrow^M t'$$

- **Recursion**

$$\frac{t[x/\mathbf{fix}(x,t)] \Downarrow^M c}{\mathbf{fix}(x,t) \Downarrow^M c}$$

$\triangleleft$

Note that $\Downarrow^M$ takes closed terms to closed terms.

Call-by-mix has been chosen as the name for this particular notion of reduction, because evaluation now switches between a call-by-value and a call-by-value style. This can be seen when considering the reduction rules for additive and multiplicative cons ($**$ and $++$): the elements of a multiplicative list are reduced in a call-by-value fashion while they are delayed in additive lists. Consequently, we have call-by-value and call-by-name lists in our linear calculus.

## 3.3 Properties of reduction

The following lemmata compile some results about the reduction relations $\Downarrow^V$, $\Downarrow^N$ and $\Downarrow^M$ that will be useful in subsequent chapters. The first one states all notions of reduction commute with substitution:

**Lemma 3.7**

- $u \Downarrow^V u' \Rightarrow \left[t[x/u] \Downarrow^V t'[x/u] \iff t[x/u'] \Downarrow^V t'[x/u']\right]$
- $u \Downarrow^N u' \Rightarrow \left[t[x/u] \Downarrow^N t'[x/u] \iff t[x/u'] \Downarrow^N t'[x/u']\right]$
- $u \Downarrow^M u' \Rightarrow \left[t[x/u] \Downarrow^M t'[x/u] \iff t[x/u'] \Downarrow^M t'[x/u']\right]$

$\triangleleft$

**Proof Lemma 3.7** Each statement is proved by induction on the derivation of a term. $\square$

The most important relationship between the different type systems and reduction semantics is stated in the next lemma:

**Lemma 3.8 Subject reduction**

1. $\forall t. \vdash t : A \in \lambda_0^{\rightarrow} \wedge t \Downarrow^V t' \Rightarrow \vdash t' : A \in \lambda_0^{\rightarrow}$
2. $\forall t. \vdash t : A \in \lambda_0^{\rightarrow} \wedge t \Downarrow^N t' \Rightarrow \vdash t' : A \in \lambda_0^{\rightarrow}$
3. $\forall t. \vdash t : A \in \lambda_0^{!,\multimap} \wedge t \Downarrow^M t' \Rightarrow \vdash t' : A \in \lambda_0^{!,\multimap}$

$\triangleleft$

The proof of these claims is made possible by the admissibility of substitution as discussed in the previous chapter.

**Proof Lemma 3.8**

1. By induction on a derivation in $\Downarrow^V$ using lemma 2.22.
2. Likewise, by induction on a derivation in $\Downarrow^N$.
3. By induction on a derivation in $\Downarrow^M$, using lemma 2.24. The applicability of this lemma is due to the fact that $\Downarrow^M$ is only defined on closed terms.

$\square$

Note that lemma 3.8 makes sense only for closed terms since these are the only ones for which reduction is defined. Only well-typed closed terms will be considered as *programs* that can be submitted to a reduction procedure. Subject reduction guarantees that the normalization of a program does not lead to run-time type errors. The property of being a program in the previous sense may be checked before execution.This means for example, that if a program incorrectly tries to project the head and the tail of an abstraction, it may be rejected on the grounds of typing information alone, without actually having to execute it and wait until it crashes. Typability analysis in this sense may be seen as a form of static analysis, a more economic means to guarantee the absence of run-time type errors than costly dynamic checks.

Moreover, the reduction relations $\Downarrow^V$ *and* $\Downarrow^N$ are deterministic:

**Lemma 3.9**

- $\forall t \in \lambda_0^{\rightarrow}. t \Downarrow^V t'$ and $t \Downarrow^V t''$ implies $t' = t''$.
- $\forall t \in \lambda_0^{\rightarrow}. t \Downarrow^N t'$ and $t \Downarrow^N t''$ implies $t' = t''$.

$\triangleleft$

The same may be shown for $\Downarrow^M$:

**Lemma 3.10 (Abramsky,[Abr93])**

$\forall t \in \lambda_0^{!,\multimap}.t \Downarrow^M t' \wedge t \Downarrow^M t'' \Rightarrow t' = t''$ ◁

The proofs of propositions (1) and (2) proceed by induction over $\Downarrow$-derivations ([Win93]). The proof of proposition (3) is due to Abramsky and may be found in [Abr93].

## 3.4 Summary and outlook

This chapter has introduced the basic semantic notions that we need in the sequel. Their basic properties were stated. We have seen that the failure of the substitution rule for $\lambda^{!,\multimap}$ does not destroy the subject-reduction property for $\Downarrow^M$, our limited notion of reduction on $\lambda_0^{!,\multimap}$-terms. This is due to the fact that reduction is only defined on closed terms.

How are these semantic notions related to our investigation on the utility of linear types?

It has been noted by Lincoln and Mitchell in their work on operational interpretations of linear lambda terms ([LM92]), that a single $\lambda^{\rightarrow}$-term may be assigned *two different* $\lambda^{!,\multimap}$-terms with *distinct* linear types. The example they give is as follows:

**Example 3.11 (Lincoln/Mitchell,[LM92])**

Consider the $\lambda^{\rightarrow}$-term

$$\lambda q.\lambda r.\lambda s.(\lambda x.q\,(r\,s)) : A \rightarrow (C \rightarrow B) \rightarrow C \rightarrow A$$

Making the inherent nonlinear operations explicit, one may obtain two different $\lambda^{!,\multimap}$-terms, namely

$$\lambda q.\lambda r.\lambda s.(\lambda x.\mathbf{drop}\;x\;\mathbf{in}\;q\,!((\star r \star s))) : A \multimap !(!C \multimap !B) \multimap !C \multimap A$$

and

$$\lambda q.\lambda r.\lambda s.(\lambda x.\mathbf{drop}\;x\;\mathbf{in}\;q\,(r\,s)) : A \multimap (C \multimap !B) \multimap C \multimap A$$

In the first term, the argument to $\lambda x.\mathbf{drop}\;x\;\mathbf{in}\;q$ is not reduced before it is discarded under $\Downarrow^M$, while in the second term it is. In the first case, this induces a call-by-name reduction of the original $\lambda^{\rightarrow}$-term and a call-by-value reduction in the second case (if these terms are actually applied). ◁

From this example they make two very important observations:

1. A $\lambda^{\rightarrow}$-term may be assigned different, incomparable linear types. Otherwise said, there a generally several ways to make the nonlinear operations inherent in $\lambda^{\rightarrow}$-terms explicit.

2. Different linear types correlate with different evaluation orders.

These observations motivate the investigation in the following chapter. There, two uniform ways to assign $\lambda^{!,\multimap}$-terms to $\lambda^{\rightarrow}$-terms are studied that reflect the basic alternatives in making nonlinear operations explicit.

# Chapter 4

# Translations

After having defined the syntax and semantics of $\lambda^{\rightarrow}$- and $\lambda^{!,\multimap}$-terms, the question is now considered how these different systems may be related.

This is done with the ultimate intention of extending the analytical means provided by use-constraints to the universe of $\lambda^{\rightarrow}$-terms. The goal is to apply the notions of essential linearity and necessary nonlinearity to $\lambda^{\rightarrow}$-terms as well.

In order to do so, a method must be found that assigns $\lambda^{!,\multimap}$-terms to $\lambda^{\rightarrow}$-terms.

J.Y.Girard, in his original article on Linear Logic ([Gir87]), had introduced two different embeddings of $\lambda^{\rightarrow}$ into (a sequent calculus formulation of) $\lambda^{!,\multimap}$. These will be adopted here to illustrate and discuss the basic choices that must be confronted when passing from $\lambda^{\rightarrow}$ to $\lambda^{!,\multimap}$.

The distinctive characteristic of $\lambda^{!,\multimap}$ is the decomposition of the $\tau$-type constructors into more basic operators (cf. [Abr93]):

- $\rightarrow$ into ! (repetition) and $\multimap$ (linear implication) and
- $\wedge$ into & and $\otimes$ (additive and multiplicative conjunction)

There is some liberty in deciding *when* to introduce *which* operator in translating from $\tau$-types to $\tau_!$-types:

**Definition 4.1 (Girard, [Gir87])** Let

1. the $(.)^L$-translation be defined as

$$\begin{aligned} A^L &= A \\ \mathbf{1}^L &= \mathbf{1} \\ (A \rightarrow B)^L &= {!A^L} \multimap B^L \end{aligned}$$

$$\begin{array}{rcl}(A \wedge B)^L & = & (A^L \,\&\, B^L) \\ (A \vee B)^L & = & (!A^L \oplus !B^L)\end{array}$$

2. the $(.)^E$-translation be defined as

$$\begin{array}{rcl}A^E & = & !A \\ \mathbf{1}^E & = & !\mathbf{1} \\ (A \to B)^E & = & !(A^E \multimap B^E) \\ (A \wedge B)^E & = & !(A^E \otimes B^E) \\ (A \vee B)^E & = & !(A^E \oplus B^E)\end{array}$$

(It is understood that the first case in each of these translations applies to *atomic* types, that is variables only.) $\triangleleft$

The alternative translations reflect the above mentioned liberty. They differ in two important aspects:

- the [!]-operator is introduced at different occasions and
- $\wedge$ is translated as $\&$ under $(.)^L$ and as $\otimes$ under $(.)^E$.

This liberty may be understood as resulting from an asymmetry in the information content of $\tau$- and $\tau_!$-types: the latter are more explicit, they encode more information about the underlying term than the former types and may be used to draw finer distinctions. The above dichotomy shows that the less informative $\tau$-types may be interpreted in several ways if finer distinctions are made.

## 4.1 Proof level translations

It can be shown that both translations are sound with respect to provability in $\lambda^{!,\multimap}$:

**Theorem 4.2 (Girard,[Gir87])** For every sequent $\Delta \vdash t : A \in \lambda^{\to}$ there exists

1. a sequent $!\Delta^L \vdash t^L : A^L \in \lambda^{!,\multimap}$
2. a sequent $\Delta^E \vdash t^E : A^E \in \lambda^{!,\multimap}$

$\triangleleft$

The proof of these two propositions proceeds by an induction on a derivation in $\lambda^{\to}$. This induction demonstrates how the type-level translations from above may be lifted to the level of proofs. It is illustrative to see how the resulting proofs in $\lambda^{!,\multimap}$ differ in their structure. It shows that the type-level translations may be read as specifying different classes of proofs *structured in a certain uniform way.*

In order to spare the reader from tracking too many cases, just the ones for axiom, application, pair and fixedpoint will be considered.

**Proof Theorem 4.2**

1. let $d = \dfrac{}{x : A \vdash x : A}\,axiom$ where $A \in \tau$.

   Then

   (a) $$d^L = \dfrac{\dfrac{}{x :!A \vdash x :!A}\,axiom}{x :!A \vdash x : A}\,d$$

   (b) $$d^E = \dfrac{\dfrac{\dfrac{}{x :!A \vdash x :!A}\,axiom}{x :!A \vdash \star x : A}\,d}{x :!A \vdash !(\star x) :!A}\,!$$

2. let $d = \dfrac{\overset{d'}{\Sigma \vdash u : A} \quad \overset{d''}{\Sigma \vdash t : A \to B}}{\Sigma \vdash (t\, u) : B}\,\to_E$

   (a) $$d^L = \dfrac{\dfrac{\overset{d'^L}{!\Delta \vdash u : A^L}}{!\Delta^L \vdash !(u) :!A^L}\,! \quad \overset{d''^L}{!\Gamma^L \vdash t^L :!A^L \multimap B^L}}{!\Delta^L, !\Gamma^L \vdash (t\,!(u))^L : B^L}\,\multimap_E$$

   (b) $$d^E = \dfrac{\overset{d'^E}{\Delta \vdash u^E : A^E} \quad \dfrac{\overset{d''E}{\Gamma^E \vdash t^E :!(A^E \multimap B^E)}}{\Gamma^E \vdash \star t^E : A^E \multimap B^E}\,d}{\Delta^E, \Gamma^E \vdash (\star t^E\, u^E) : B^E}\,\multimap_E$$

   It is remarkable that in the case of $d^L$, the argument in the application is wrapped with a [!]-operator while the case of $d^E$ does not insert a storage operation. This difference in structure causes the operational semantics of these applications to differ as well. Since storage is interpreted as delay in our present semantics, the reduction of the $d^L$-term will not try to normalize the argument while the reduction of the $d^E$-term will.

3. let $d = \dfrac{\Sigma \vdash t_1 : A \quad \Sigma \vdash t_2 : B}{\Sigma \vdash (t_1, t_2) : A \wedge B}\,\wedge_I$

   (a) $$d^L = \dfrac{\begin{matrix} d'^L \\ !\Delta^L \vdash t_1^L A^L \\ weakening(s) \\ !\Delta^L, !\Gamma^L \vdash t_1^L : A^L \end{matrix} \quad \begin{matrix} d''^L \\ !\Gamma^L \vdash t_2^L : B^L \\ weakening(s) \\ !\Delta^L, !\Gamma^L \vdash t_2^L : B^L \end{matrix}}{!\Delta^L, !\Gamma^L \vdash \langle t_1^L,\, t_2^L \rangle : A^L \,\&\, B^L}\,\&_I$$

(b) $$d^E = \cfrac{\cfrac{\overset{d'^E}{\Delta^E \vdash t_1^E : A^E} \quad \overset{d''^E}{\Gamma^E \vdash t_2^E : B^E}}{\Delta^E, \Gamma^E \vdash (t_1^E, t_2^E) : A^E \otimes B^E}\ \otimes_I}{\Delta^E, \Gamma^E \vdash !(t_1^E, t_2^E) !(A^E \otimes B^E)}\ !$$

Note that the weakenings have to be introduced in the additive translation in order to make contexts equal as required by the $[\&_I]$-rule. Depending on the structure of the term, this requirement may trigger more nonlinearity than the multiplicative translation. However, neither one may be expected to outperform the other in all cases.

4. let $$d = \cfrac{\overset{d'}{\Delta, x : A \vdash t : A}}{\Delta \vdash \mathbf{fix}(x, t) : A}\ fix$$

(a) $$d^L = \cfrac{\cfrac{\cfrac{\overset{d'^L}{!\Delta^L, x :!A^L \vdash t^L : A^L}}{!\Delta^L, x :!A^L \vdash !(t^L) :!A^L}\ !}{!\Delta^L \vdash \mathbf{fix}(x, !(t^L)) :!A^L}\ fix}{!\Delta^L \vdash \star\mathbf{fix}(x, !(t^L)) : A^L}\ d$$

(b) $$d^E = \cfrac{\overset{d'^E}{\Delta, x : A^E \vdash t^E : A^E}}{\Delta^E \vdash \mathbf{fix}(x, t^E) : A^E}\ fix$$

(Note that the $A^E$-types comply with the requirement of the $[fix]$-rule since they always have ! as topmost operator.)

□

Given the alternative translations exposed in the proof of theorem 4.2 we define the following uniform proof classes:

**Definition 4.3 Uniform proofs**

- $(\lambda^{\rightarrow})^L = \{!\Delta^L \vdash t^L : A^L \mid \Delta \vdash t : A \in \lambda^{\rightarrow}\}$
- $(\lambda^{\rightarrow})^E = \{\Delta^E \vdash t^E : A^E \mid \Delta \vdash t : A \in \lambda^{\rightarrow}\}$

◁

The first class corresponds closely to the class of proofs that are derivable in P.Wadler's system of 'standard types' [Wad91a], with the minor deviation that he uses $\otimes$-pairs.

## 4.2 Comparing proof classes under minimization

This section departs from the basic observation that neither translation is sensitive to the *actual* nonlinearity requirements dictated by the structure of the term. They both introduce [!]-operators irrespecive of whether they are logically necessary or not.

After translation, the resulting proofs (terms) may be minimized using $\mu_!(.)$. It turns out that different translations *of the same term* reveal totally different information about the linearity of e.g. formal function parameters.

### 4.2.1 Nonuniformity across proof classes

It was said in the proof that, depending on the structure of the pair, one translation may trigger more necessary [!]- or [d]-inferences than the other. No strategy may be expected to perform optimally in all situations. (We use minimization in order to strip off superflous [!]-operators.

**Example 4.4 (Alternative translations of pairs)** The sequent

$$\vdash \lambda x.(x, x) : A \to (A \wedge A)$$

may be mapped and subsequently minimized to

- $\vdash \lambda x.\langle x,\ x\rangle : A \multimap (A \,\&\, A)$ or
- $\vdash \lambda x.\mathbf{copy}\ x\ \mathbf{as}\ x_1/x_2\ \mathbf{in}\ (\star x_1, \star x_2) :!A \multimap (A \otimes A)$

In turn, the sequent

$$\vdash \lambda x.\lambda y.(x, y) : A \to B \to (A \wedge B)$$

may be mapped and minimized to

- $\vdash \lambda x.\lambda y.\langle \mathbf{drop}\ y\ \mathbf{in}\ x,\ \mathbf{drop}\ x\ \mathbf{in}\ y\rangle :!A \multimap !B \multimap (A \,\&\, B)$ or
- $\vdash \lambda x.\lambda y.(x, y) : A \multimap B \multimap (A \otimes B)$

◁

These examples demonstrate that when trying to do linearity analysis of $\lambda^{\to}$-terms, the information that may be obtained by using one uniform translation differs strongly from the information obtained under the other uniform translation.

The next example will discuss the fact that the basic distinction between these proof classes also applies when the alternative translations of a pair are disregarded. If these had been the only obstacle to devising a unique, simple method for analysing the linearity of $\lambda^{\rightarrow}$-terms, we could have made the decision to drop this distinction. The next example shows that the ambiguity would not have vanished.

**Example 4.5 (Incongruence of translations with respect to $\mu_!(.)$)** Consider the term

$$\lambda x.(\lambda y.()\,(x\,())) : (\mathbf{1} \rightarrow A) \rightarrow \mathbf{1}$$

It may be translated as:

- $\lambda x.(\lambda y.\mathbf{drop}\, y\, \mathbf{in}\, ()\, !(\star x\, !())) :!(!\mathbf{1} \multimap A) \multimap \mathbf{1}$ by $(.)^L$, and as
- $!(\lambda x.(\star!(\lambda y.\mathbf{drop}\, y\, \mathbf{in}\, !())\,(\star!\star x\, !()))) :!(!(!\mathbf{1} \rightarrow !A) \rightarrow !\mathbf{1})$ by $(.)^E$.

Minimization of both terms yields:

- $\lambda x.(\lambda y.\mathbf{drop}\, y\, \mathbf{in}\, ()\, !(\star x\, ())) :!(\mathbf{1} \multimap A) \multimap \mathbf{1}$
- $\lambda x.(\lambda y.\mathbf{drop}\, y\, \mathbf{in}\, ()\,(x\, ())) : (\mathbf{1} \multimap !A) \multimap \mathbf{1}$

The striking observation is that the minimized terms have different types. In the first term, the formal parameter x is required to be nonlinear while in the second term it becomes linear under $\mu_!(.)$.

This example is similar to the one of Lincoln and Mitchell given at the end of the previous chapter: the alternative linear types may be understood as resulting from the above uniform translations with subsequent minimization. $\triangleleft$

From example 4.5 it follows that the dichotomy between the different translations **does not collapse** when applying minimization to extract linearity information from the resulting terms, even when the alternative translations of pairs are not effective.

### 4.2.2 Nonuniformity of 'optimal types'

Different translations may be compared according to the amount of nonlinarity they require with respect to minimization. A translation may be considered better than the other if it requires less storage operations. This property is reflected in the linear type different that translations assign to a $\lambda^{\rightarrow}$-term.

Another reasonable restriction that could be employed to eliminate the ambiguity is trying to achieve some kind of optimal necessitation of nonlinear operations by always choosing the best translation.

This criterion ensures in example 4.4 that there always exists a unique way to reveal maximal linearity information. However, which translation is to be employed differs from term to term.

There exist terms where only a non-uniform translation that locally alternates between different alternatives may reveal maximal linearity information:

**Example 4.6 (Mixed translations may be better)** Combining the example terms from 4.4 the sequent

$$\vdash \lambda x.\lambda y.(x,(y,y)) : A \to B \to (A \wedge B \wedge B)$$

should according to the present optimality criterion be translated by a mixed strategy as

$$\vdash \lambda x.\lambda y.(x,\langle y,\ y\rangle) : A \multimap B \multimap (A \otimes (B \,\&\, B))$$

$\triangleleft$

The nonuniform translation of the example term represents the single best method to obtain this information, while both uniform approaches may be seen to perform suboptimally.

The question remains wether the very strong requirement of maximal linearity information may help to eliminate the ambiguity in general.

Example 4.5 however shows, that even this stronger requirement does not help in defining an unambigous notion of linearity analysis for $\lambda^{\to}$-terms. The types of both terms both contain only one [!]-operator, and it may easily been seen that no other term-assignment may do better.

## 4.3 Linear type inference

How does the above, pragmatic approach compare with other approaches to relate $\lambda^{\to}$- and $\lambda^{!,\multimap}$-terms? In order to make comparisons, we must understand that assigning $\tau_!$-types to $\lambda^{\to}$-terms and assigning $\lambda^{!,\multimap}$-terms to $\lambda^{\to}$-terms are equivalent problems.

The examples 4.4 and 4.6 concerning pairs have made it clear that a single one translation does not exhaust the entire range of possibilities in assigning $\tau_!$-types to $\lambda_0^{\to}$-terms. *Different translations may derive different types.* This means that when considering the *general* problem of assigning $\tau_!$-types to $\lambda_0^{\rightarrow}$-terms, one possibly has to confront the whole variety of alternatives inherent in linear proof structures.

In trying to cope with the multitude of possibilities, a different approach, drawing upon a notion of 'most general' linear type to assign $\tau_!$-types to $\lambda_0^{\rightarrow}$-terms may be taken. Most general linear types are supposed to 'subsume' the multitude of different possible type-assignments exhibited in the previous chapter, analogous to the most general types derived in e.g. Hindley-Milner type-assignment for $\lambda^{\rightarrow}$-terms(cf. [Wad91a].

Without further defining the notions of 'most general' and 'subsumption', let us mention that this approach hits upon the same ambiguity that was discussed with regard to uniform translations. Recalling example 4.5, the term $\lambda x.(\lambda y.()\,(x\,()))$ may be assigned two different $\tau_!$-types, $!(\mathbf{1} \multimap A) \multimap \mathbf{1}$ and $(\mathbf{1} \multimap !A) \multimap \mathbf{1}$. It is easy to observe that they cannot both be substitution instances of a common, more general type (cf. [LM92] for similar example). This means that the ambiguity destroys the principal typing property that was to serve as a basis for picking unique translations (cf. [T$\ddot{9}$3]).

Concering type inference, proof classes have an advantage: while general linear type inference is nondeterministic because it must enumerate a set of incomparable types in general, type inference is deterministic *within* each of the above mentioned classes. It starts by inferring a $\tau$-type for the respective $\lambda_0^{\rightarrow}$-term and then translates the corresponding proof by one of the uniform (deterministic) translations from above. If desired, the result may subsequently be (deterministically) minimized.

Let us mention that the translations defined here, as well as minimization, interact well with Hindley-Milner-style polymorphism.

Since $\lambda_0^{\rightarrow}$-terms as defined here may always be assigned a *most general* $\tau$-type, ML-style polymorphic type inference may be done in $(\lambda_0^{\rightarrow})^L$ and $(\lambda_0^{\rightarrow})^E$ as well, usign deterministic translations. Instead of giving and explicit treatment of this here, consider the following minimized *translation* of the central example from [DM82]. We use the rules

$$\frac{\Delta \vdash t : B[A]}{\Delta \vdash t : \forall \xi . B[\xi]}\ \forall_I, A \text{ not free in } \Delta \quad \frac{\Delta \vdash t : \forall \xi . B[\xi]}{\Delta \vdash t : B[\xi / A]}\ \forall_E$$

for type abstraction and instantiation.

**Example 4.7 (Applying the identity function to itself)**

$$
\dfrac{\Phi \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{g :! \bigwedge \xi.\xi \multimap \xi \vdash g :! \bigwedge \xi.\xi \multimap \xi}\,ax}{g :! \bigwedge \xi.\xi \multimap \xi \vdash \star g : \bigwedge \xi.\xi \multimap \xi}\,d}{g :! \bigwedge \xi.\xi \multimap \xi \vdash \star g : a \multimap a}\,\forall_E \qquad \dfrac{\dfrac{\dfrac{}{f :! \bigwedge \xi.\xi \multimap \xi \vdash f :! \bigwedge \xi.\xi \multimap \xi}\,ax}{f :! \bigwedge \xi.\xi \multimap \xi \vdash \star f : \bigwedge \xi.\xi \multimap \xi}\,d}{f :! \bigwedge \xi.\xi \multimap \xi \vdash \star f : (a \multimap a) \multimap (a \multimap a)}\,\forall_E}{f :! \bigwedge \xi.\xi \multimap \xi, g :! \bigwedge \xi.\xi \multimap \xi, \vdash (\star f \star g) : a \multimap a}\,\multimap_E}{I :! \bigwedge \xi.\xi \multimap \xi \vdash \textbf{copy } I \textbf{ as } f/g \textbf{ in } (\star f \star g)}\,c!*}{\vdash \textbf{let } I \textbf{ be } !(\lambda x.x) \textbf{ in copy } I \textbf{ as } f/g \textbf{ in } (\star f \star g) : a \multimap a}\,let
$$

$$
\text{where } \Phi = \dfrac{\dfrac{\dfrac{\dfrac{}{x : a \vdash x : a}\,ax}{\vdash \lambda x.x : a \multimap a}\,\multimap_I}{\vdash \lambda x.x : \bigwedge \xi.\xi \multimap \xi}\,\forall_I}{\vdash !(\lambda x.x) :! \bigwedge \xi.\xi \multimap \xi}\,! \qquad \triangleleft
$$

For a discussion of Hindley-Milner-style polymorphic type inference in the general class of $\lambda_0^{!,\multimap}$-terms, the reader is referred to the exposition in [Mac91]. Note that the approach via uniform translations avoids the inconvenience concerning the generalization of nonlinear types that are discussed in [Mac91]. This is due to the fact that polymorphic type inference takes place in $\lambda^{\rightarrow}$. Generalization is handled on this level. The translations only add the nonlinearity information afterwards.

## 4.4 Discussion

How must the uniform proof classes that where considered here be understood with regard to the general problem of analyzing linearity properties of $\lambda^{\rightarrow}$-terms?

The first thing that can be noted is that they deliberately narrow the perspective to one particular kind of linearity information. By making choices with respect to the positions were [!]-operators are introduced in the target term, they *fix a particular notion of linearity*, which has been seen to be unique to each class considered here.

Uniform proof classes make different $\lambda^{\rightarrow}$-terms *commensurable* with regard to their structure (linearity) by fixing the notion of linearity that is to be applied.

Different uniform proof classes employ different notions of linearity. The linearity information they convey about one and the same term must generally be considered incommensurable (cf. example 4.5).

Thus, the proof classes represent *distinct analytical tools* that must not be confounded.

Whichever analytical tool is to be employed depends on the problem that was posed.

For example, given two alternative $\lambda^{\rightarrow}$-formulations of *the same algorithm*, a categorical decision for one particular analytical tool is required to compare these formulations with respect to their *linearity properties*.

Such comparisons arise in the study of *program transformations* that do not change the operational semantics of the respective term but may support its efficient execution. Thus, being able to make comparisons regarding linearity is certainly of practical interest.

Concerning different proof classes, we know that the same term may have different semantics in different proof classes. This means that comparisons, such as between program transformations, are *futile* if the analytical tools are allowed to vary from instance to instance.

## 4.5 Summary and outlook

Uniform proof classes will serve as our basis for investigating the applicability of $\tau_!$-types to the analysis of $\lambda^{\rightarrow}$-terms.

It was shown that for every $\lambda_0^{\rightarrow}$-term there may exist several valid assignments of $\tau_!$-types. The criteria needed to guide a possible choice among these alternatives are not inherent in linear types.

The fact that there exist several, seemingly equally valid possibilities to relate $\lambda^{\rightarrow}$- and $\lambda^{!,\multimap}$-terms (several proof classes) is clearly not very satisfying because of the inherent ambiguity.

What is needed is further knowledge about the potential applications of different translation methods. So far, it is only known that they provide different kinds of linearity information about a term.

This knowledge must be found by looking to the practical problems that pose themselves in the implementation and use of functional programming languages.

In relation to these extra-logical concerns, ambiguity with regard to $\lambda^{\rightarrow}$ must not be seen as a weakness of $\lambda^{!,\multimap}$, but as an asset: it broadens the range of potential applications because one is free to select different problem-specific translations from logically equivalent alternatives without being restricted to one particular translation a-priori.

Only if we know which information can be applied to which problem can an informed choice be made.

Consequently, the next chapter will examine these alternatives from a pragmatic point of view, focussing on efficency problems in the compilation and execution of lazy functional languages.

# Chapter 5

# Implementing call-by-name

It was demonstrated that there exists a discouraging variety of ways to make the structural properties of $\lambda^{\rightarrow}$-proofs explicit. In order to apply the additional information that is encoded in linear types to the analysis of $\lambda_0^{\rightarrow}$-programs, a principled selection must be made from this variety. Reasonable selections have been portrayed under the heading of uniform proof classes. These characterize different possible *notions of structure* that may be applied to $\lambda_0^{\rightarrow}$-terms.

In the following, one particular notion of structure (the $(\lambda_0^{\rightarrow})^L$-class) is applied to the analysis of run-time properties of $\lambda_0^{\rightarrow}$-terms under a $\Downarrow^N$-semantics.

This particular focus has been chosen with regard to a proposal of S.Abramsky. In [Abr93] he put forward the idea that linearity information may be used to optimize the execution of call-by-name reduction. By translating functional programs to linear functional programs (he too proposes to use a uniform translation) and subsequently optimizing the resultant linear terms, he saw the opportunity to supplement traditional methods of static analysis based on abstract interpretation with *logic-based program analysis.*

With the minimization procedure at hand, we are able to perform the kind of 'linearization' on $\lambda_0^{!,\multimap}$-terms that he mentions as the main tool in pursuing such a proposal.

In the following, it is shown how the minimization procedure $\mu_!(.)$ can be used to detect strictness information in $\lambda_0^{!,\multimap}$-terms. This particular application of linearity analysis to optimizing the execution of call-by-name reduction is the main contribution of this thesis. Its implications are listed below:

- It makes an explicit connection between the type system, a class of proof structures and the semantics of $\lambda^{!,\multimap}$-terms.

- It illustrates the basic utility of the information made explicit by $\tau_!$-types, and

- it gives a very strong indication that the restriction to a uniform proof-class is not so severe since just the right information is obtained to solve interesting problems.
- It presents a very efficient *method* for doing strictness analysis that differs radically from approaches based on abstract interpretation and denotational semantics. This is due to the different angle taken in this work: *semantic* properties of a term are traced back to its *syntactic* properties which are fairly easy to analyse.

## 5.1 Motivation

An implementation of $\Downarrow^N$ may be understood as a function

$$I : (\lambda^{\rightarrow}) \rightarrow (\lambda^{\rightarrow})$$

satisfying the correctness condition

$$t \Downarrow^N t' \iff I(t) = t' \tag{5.1}$$

Implementing $\Downarrow^N$ incurs a high cost (cf. [PJ87]):

- computations must be suspended (delayed) when passed as an argument to a function. Building suspensions is expensive, and has the further effect that
- whenever an argument is duplicated, the computation is carried out several times instead of being shared.

In general $\Downarrow^N$ is not implemented directly, but $I$ is decomposed into several distinct subfunctions, such as static analysis, compilation and subsequent execution on an abstract (or concrete) machine.

Static analysis tries to determine run-time properties before actual execution. This information can be used to make certain checks and decisions in the compilation phase already, thereby saving the run-time overhead. Static type-checking, for example, may eliminate the need for costly run-time type checking.

It may also aid compilation in ordering the execution of a program's parts in a way that avoids the above mentioned problems. For example it may determine that the execution of an argument need not be delayed in order to satisfy the correctness condition (5.1), to the effect that less computations are suspended and eventually duplicated.

This particular application of static analysis is known under the name of *strictness analysis*. A function f is said to be *strict* in its argument if

$$(f\ \perp) = \perp$$

where $\perp$ means nontermination. Strictness implies that an argument to a function may safely be evaluated *before* substituting it into the body of the function, without corrupting the call-by-name termination semantics of the expression.

The goal of the following sections is to investigate wether the information contained in $\tau_!$-types may help to implement call-by-name reduction semantics $\Downarrow^N$ for functional programs more efficiently.

The claim that $\lambda^{!,\multimap}$ may serve as a tool to solve such problems is partly justified by the following

**Lemma 5.1 Simulation** $\forall t : \mathbf{1} \in \lambda_0^{\rightarrow}.t \Downarrow^N () \iff t^L \Downarrow^M ()$ ◁

A proof of this will not be given here for reasons of brevity. The validity of this claim may be understood when considering the $(.)^L$-translation for applications:

$$(t_1\, t_2)^L \equiv (t_1^L\, !t_2^L)$$

The argument in the application is wrapped with a [!]-operator with the effect that in the reduction

$$\frac{\begin{array}{c}\vdots\\ t_1^L \Downarrow^M \lambda x.t_1'\end{array} \quad \dfrac{}{!t_2^L \Downarrow^M !t_2^L} \quad \begin{array}{c}\vdots\\ t_1'[x/!t_2^L] \Downarrow^M c\end{array}}{(t_1^L\, !t_2^L) \Downarrow^M c}$$

it is effectively delayed, just like in the call-by-name reduction of the original term.

An analogous result is mentioned (without proof) by Mackie in [Mac].

However, this result does not imply a gain in execution efficiency, since *all* function arguments are delayed uniformly.

It has been proposed (by e.g. S.Abramsky in [Abr93]) that 'optimizing' the resulting linear terms by stripping superflous [!]-operators from it may serve to optimize lazy execution of functional programs.

This intuition draws upon examples like the following.

**Definition 5.2** Let $\mathbf{\Omega} \equiv \mathbf{fix}(x, x)$ ◁

**Remark 5.3 (Standard divergent term)** $\mathbf{\Omega}$ will serve as the standard divergent term in the sequel. Note that its $(.)^L$ translation is divergent as well, and has *linear type* (trivially so by the definition of $(.)^L$). ◁

**Example 5.4 (Relating strictness and syntactic structure)** The term $(\lambda x.()\,\mathbf{\Omega})$ has a call-by-name normal form:

$$\frac{\overline{\lambda x.() \Downarrow^N \lambda x.()} \qquad \overline{()[x/\mathbf{\Omega}] \Downarrow^N ()}}{(\lambda x.()\,\mathbf{\Omega}) \Downarrow^N ()}$$

The same term *diverges* under $\Downarrow^V$, which means that there does not exist a $d$ for which $(\lambda x.()\,\mathbf{\Omega}) \Downarrow^V d$ is derivable. Consider the following proof attempt:

$$\frac{\overline{\lambda x.() \Downarrow^V \lambda x.()} \qquad \dfrac{\dfrac{\begin{array}{c}\vdots\ \infty\\ \mathbf{\Omega} \Downarrow^V c\end{array}}{\mathbf{\Omega} \Downarrow^V c}}{\mathbf{\Omega} \Downarrow^V c} \qquad \overline{()[x/c] \Downarrow^V d}}{(\lambda x.()\,\mathbf{\Omega}) \Downarrow^V d}$$

The crucial observation here is, that the discrepancy between the two reductions arises because $\Downarrow^N$ *discards* a nonterminating term without reducing it, while $\Downarrow^V$ is compelled to reduce it, thereby entering into a loop.

Now consider the analogous situation in $\lambda^{!,\multimap}$. The term

$$(\lambda x.()\,\mathbf{\Omega})^L = (\lambda x.\mathbf{drop}\ x\ \mathbf{in}\ ()\ \mathbf{!}(\mathbf{\Omega}^L))$$

has a normal form under $\Downarrow^M$:

$$\frac{\overline{\lambda x.\mathbf{drop}\ x\ \mathbf{in}\ () \Downarrow^M \lambda x.\mathbf{drop}\ x\ \mathbf{in}\ ()} \qquad \overline{\mathbf{!}(\mathbf{\Omega}^L) \Downarrow^M \mathbf{!}(\mathbf{\Omega}^L)} \qquad \dfrac{\overline{() \Downarrow^M ()}}{\mathbf{drop}\ \mathbf{!}(\mathbf{\Omega}^L)\ \mathbf{in}\ () \Downarrow^M ()}}{(\lambda x.\mathbf{drop}\ x\ \mathbf{in}\ ()\ \mathbf{!}(\mathbf{\Omega}^L)) \Downarrow^M ()}$$

◁

The argument is *necessarily* wrapped with a [!]-operator because it will subsequently be discarded. The corresponding reduction rule ensures that the derivation is finite, although the argument in the application is reduced eagerly.

Linearity in turn can be taken as a guarantee that a term will be reduced eventually and may therefore just as well be reduced immediately. The intuitive conclusion from this is that linear arguments may be considered *strict*, and that linearity analysis carried out in $\lambda^{!,\multimap}$ may therefore complement the kind of strictness analysis carried out in the framework of domain-theoretic semantics (cf. [Abr93, BHA86]).

A result in support of this claim is the following

**Lemma 5.5** Let $x : A \vdash t : B \in \lambda^{!,\multimap}$ with $x$ linear, $u : A$. Then

$$t[x/u] \Downarrow^M N \wedge u \notin St(N) \Rightarrow u \Downarrow^M u' \tag{5.2}$$

$\triangleleft$

**Proof**

The proof is by induction on a $\Downarrow^M$-derivation. Assume

$$t[x/u] \Downarrow^M N \wedge u \notin St(N)$$

If $t[x/u] \Downarrow^M N$ is

- $\lambda y.t[x/u] \Downarrow^M N$. Then the reduction rule for abstractions requires that

  $$N \equiv \lambda y.t[x/u]$$

  Since $x$ is linear, it follows that $u \in St(\lambda y.t[x/u])$, contradicting the assumption that $u \notin St(N)$. This implies $u \Downarrow^M u'$.

- $!(t)[x/u] \Downarrow^M N$. Then type-correctness requires $x$ to be nonlinear, contradicting the assumption that it is linear. It follows that $u \Downarrow^M u'$.

- $() \Downarrow^M N$. The fact that $x \notin ()$ contradicts the linearity assumption. Thus $u \Downarrow^M u'$

- $(t_1\, t_2)[x/u] \Downarrow^M N$. Two cases have to be distinguished. Either $x \in Fv(t_1)$ or $x \in Fv(t_2)$. In the first case, $(t_1\, t_2)[x/u] \Downarrow^M N$ was obtained from premises $t_1[x/u] \Downarrow^M \lambda y.t_1', t_2 \Downarrow^M t_2'$ and $t_1'[y/t_2'] \Downarrow^M N$. If $u \notin St(t_1')$ then $u \Downarrow^M u'$ holds by the induction hypothesis. Otherwise $u \in St(t_1'[y/t_2'])$. Since $u \notin St(N)$ and $t_1'[y/t_2'] \Downarrow^M N$, the induction hypothesis yields $u \Downarrow^M u'$. The second case follows by the same line of reasoning applied to $t_2$.

- **drop** $t_1$ **in** $t_2[x/u] \Downarrow^M N$. Two cases have to be distinguished. Either $x \in Fv(t_1)$ or $x \in Fv(t_2)$. In the first case, $t_1 \Downarrow^M t_2$ was obtained from premises $t_1[x/u] \Downarrow^M !(t_1')$ and $t_2 \Downarrow^M N$. If it were the case that $u \in St(t_1')$ then linearity of $x$ would be violated and the conclusion would follow. Otherwise, the conclusion follows from the induction hypothesis. In the second case, the induction hypothesis can be applied because $t_2[x/u] \Downarrow^M N$ and $u \notin St(N)$ by assumption, implying $u \Downarrow^M u'$ as required.

The other cases are similar and will be omitted. $\square$

Lemma 5.5 presents evidence that strictness indeed has something to do with linearity. It tells that if a computation 'consumes' a linear substitution, it does so by reducing, not discarding it. Consequently, this substitution may as well be reduced beforehand, without altering the semantics of the term.

**Remark 5.6 ()** Note that this result holds for the entire class of $\lambda_0^{!,\multimap}$-terms under $\Downarrow^M$-reduction and is not restricted to particular proof classes. $\triangleleft$

With use-constraints at hand, linearity information can be extracted from typing proofs. With the minimization procedure at hand, these can be subsequently optimized, thereby increasing eagerness in their execution without violating the correctness condition (5.1) for implementations of $\Downarrow^N$.

Evidence in this direction is provided by the following lemma. It states that $\Downarrow^M$-reduction in $\mu_!(\lambda^{\to L})$ validates a restricted analogue to the $\beta$-axiom from traditional $\lambda$-calculus (cf.[Bar84b]).

**Lemma 5.7** Let $\vdash (\lambda x.t\, uT_1 \dots T_n) : \mathbf{1}[\mathcal{C}] \in c - \lambda^{!,\multimap}$. Then

$$\mu_{!\mathcal{C}}((\lambda x.t\, uT_1 \dots T_n))^L \Downarrow^M () \iff \mu_{!\mathcal{C}}((t[x/u]\, T_1 \dots T_n))^L \Downarrow^M () \tag{5.3}$$

$\triangleleft$

**Proof Lemma 5.7**

$\Rightarrow$

The claim follows from the stronger proposition

$$\begin{aligned} & \Big[(\lambda x.\mu_!(t^L)\, \mu_!(!u^L)\mu_!(!T_1^L) \dots \mu_!(!T_n^L)) \Downarrow^M N\Big] \\ \Rightarrow\ & \Big[(\mu_!(t^L)[x/\mu_!(!u^L)]\, \mu_!(!T_1^L) \dots \mu_!(!T_n^L)) \Downarrow^M N\Big] \end{aligned}$$

for $N \in \lambda_0^{!,\multimap}$

This is proved by induction on $n$:

- $n = 0$
  Assume $(\lambda x.\mu_!(t^L)\, !(\mu_!(u^L))) \Downarrow^M N$. Then there exists a finite derivation

$$\dfrac{\dfrac{}{\lambda x.\mu_!(t^L) \Downarrow^M \lambda x.\mu_!(t^L)} \qquad \dfrac{}{!(\mu_!(u^L)) \Downarrow^M !(\mu_!(u^L))} \qquad \begin{array}{c}\vdots\,(*)\\ t[x/!(\mu_!(u^L))] \Downarrow^M N\end{array}}{(\lambda x.\mu_!(t^L)\, !(\mu_!(u^L))) \Downarrow^M N}$$

  with derivable subproof $(*)$

- $n \Rightarrow n+1$
  Assume

$$(\lambda x.\mu_!(t^L)\,\mu_!(!u^L)\mu_!(!T_1^L)\ldots\mu_!(!T_{n+1}^L)) \Downarrow^M N \tag{5.4}$$

  and

$$\mu_!(!T_i^L) = !(\mu_!(T_i^L)), i = 1,\ldots,n+1$$

  without loss of generality
  Then (5.4) must have been derived from premises

  1. $(\lambda x.\mu_!(t^L)\,\mu_!(!u^L)\mu_!(!T_1^L)\ldots\mu_!(!T_n^L)) \Downarrow^M \lambda y.s$
  2. $!(\mu_!(T_{n+1}^L)) \Downarrow^M !(\mu_!(T_{n+1}^L))$
  3. $s[y/!(\mu_!(T_{n+1}^L))] \Downarrow^M N$

  By induction

$$(\mu_!(t^L)[x/\mu_!(!u^L)]\,\mu_!(!T_1^L)\ldots\mu_!(!T_n^L)) \Downarrow^M \lambda y.s$$

  By the application reduction rule:

$$(\mu_!(t^L)[x/\mu_!(!u^L)]\,\mu_!(!T_1^L)\ldots\mu_!(!T_{n+1}^L)) \Downarrow^M N$$

  as required.

$\Leftarrow$

The claim follows from the stronger proposition

$$\left[\mu_!((t[x/u]\,T_1\ldots T_n)^L) \Downarrow^M N \Rightarrow \mu_!((\lambda x.t\,uT_1\ldots T_n)^L) \Downarrow^M N\right] \tag{5.5}$$

for $\mu_!(!(u^L)) \notin St(N)$ by taking $N \equiv ()$.
This is proved by induction on $n$:

- $n = 0$
  Assume

$$\mu_!(t^L)[x/\mu_!(!(u^L))] \Downarrow^M N \tag{5.6}$$

  with $\mu_!(!(u^L)) \notin St(N)$. The following requirements are fulfilled:

  1. $\lambda x.\mu_!(t^L) \Downarrow^M \lambda x.\mu_!(t^L)$ is true by the reduction rule for abstractions.
  2. $\mu_!(!(u^L)) \Downarrow^M u'$ follows from the assumption (5.6) and lemma 5.5.

3. $\mu_!(t^L)[x/u'] \Downarrow^M N$ may be concluded by applying lemma 3.7, which states that reduction commutes with substitution, to the assumption (5.6) and condition (2).

By the $\Downarrow^M$-reduction rule for applications

$$(\lambda x.\mu_!(t^L)\,\mu_!(!(u^L))) \Downarrow^M N$$

as required.

- $n \Rightarrow n+1$

Assume

$$(\mu_!(t^L)[x/\mu_!(!u^L)]\,!(\mu_!(T_1^L))\ldots!(\mu_!(T_{n+1}^L))) \Downarrow^M N \tag{5.7}$$

with $\mu_!(!(u^L)) \notin St(N)$. Furthermore assume $\mu_!(!(T_i^L)) = !(\mu_!(T_i^L)), i = 1\ldots n$ without loss of generality. Then (5.7) must have been derived from premises

1. $(\mu_!(t^L)[x/\mu_!(!u^L)]\,!(\mu_!(T_1^L))\ldots!(\mu_!(T_n^L))) \Downarrow^M \lambda y.s$,
2. $!(\mu_!(T_{n+1}^L)) \Downarrow^M !(\mu_!(T_{n+1}^L))$ and
3. $s[y/!(\mu_!(T_{n+1}^L))] \Downarrow^M N$

Two cases must be considered. If $\mu_!(!u^L) \notin St(\lambda y.s)$ then

$$(\lambda x.\mu_!(t^L)\,\mu_!(!u^L)!(\mu_!(T_1^L))\ldots!(\mu_!(T_n^L))) \Downarrow^M \lambda y.s$$

may be concluded from the induction hypothesis.
Otherwise, $\lambda y.s = \lambda y.s'[x/\mu_!(!(u^L))]$. By (the derivable) premise (3):

$$s'[x/\mu_!(!u^L)][y/!(\mu_!(T_{n+1}^L))] \Downarrow^M N$$

with $\mu_!(!(u^L)) \notin N$.
If $\mu_!(!u^L) = \mu_!(u^L)$, then

$$\mu_!(!u^L) \Downarrow^M u'$$

by lemma 5.5. Otherwise $\mu_!(!u^L) \Downarrow^M !\mu_!(u^L)$ by the reduction rule for of-course.
From this and lemma 3.7 conclude

$$(\mu_!(t^L)[x/u']\,!(\mu_!(T_1^L))\ldots!(\mu_!(T_n^L))) \Downarrow^M \lambda y.s'[x/u']$$

Since now $\mu_!(!u^L) \notin St(\lambda y.s'[x/u'])$, the induction hypothesis yields

$$(\lambda x.\mu_!(t^L)\, u' !(\mu_!(T_1^L)) \ldots !(\mu_!(T_n^L))) \Downarrow^M \lambda y.s$$

from which

$$(\lambda x.\mu_!(t^L)\, \mu_!(!u^L) !(\mu_!(T_1^L)) \ldots !(\mu_!(T_n^L))) \Downarrow^M \lambda y.s$$

may safely be concluded.

Applying the reduction rule for applications yields the required result in both cases.

The truth of the claim that (5.5) does imply the $\Rightarrow$-direction of (5.3) may be understood by the fact that only case in which (5.5) is not applicable is when $u \equiv ()$ is the normal form that the term reduces to, because then certainly $u \in St(N)$. But since $u \Downarrow^M ()$, the $\Rightarrow$-direction of (5.3) obviously holds in this exceptional case.

$\square$

**Remark 5.8 (Safety)** The $\Leftarrow$-direction of lemma 5.7 is **central** in justifying the postulated relationship of minimization to strictness analysis. It states that - under $\mu_!(.)$ - passing from lazy reduction (by substitution) to eager reduction is **safe** for terms of **1**-type. In particular, termination is preserved. $\triangleleft$

**Corollary 5.9** $(\lambda x.t\, uT_1 \ldots T_n)^L \Downarrow^M () \iff (t[x/u]\, T_1 \ldots T_n)^L \Downarrow^M ()$ $\triangleleft$

**Proof Corollary 5.9** The claim follows from

1. the observation that taking $\mathcal{C} = \perp$ does not invalidate but simplify the above proof since it abolishes a few case distinctions and
2. the fact that $\mu_{!\perp}(t^L) \equiv t^L$

$\square$

It remains to show that $\Downarrow^N$ may indeed correctly be implemented in $\mu_!((\lambda_0^{\rightarrow})^L)$.

## 5.2 Strictness analysis via $\mu_!(.)$

The next theorem states that *logically* superflous [!]-operators may be safely removed from a $(\lambda^{\rightarrow})^L$-proof without altering the termination semantics of the corresp. term. (Due to space and time constraints, the following investigation will be restricted to the implicational fragments of $\lambda^{\rightarrow}$ and $\lambda^{!,\multimap}$. Note however, that it may very well be extended to cover pairs and disjunction as well.)

**Theorem 5.10 Implementability for $\lambda_0^{\rightarrow}$-terms of ground type**

$\forall \vdash t : \mathbf{1} \in \lambda_0^{\rightarrow}. t^L \Downarrow^M () \iff \mu_!(t^L) \Downarrow^M ()$ ◁

In order to prove this, a family of predicates $\mathcal{I}_\tau(.) \subseteq \lambda_0^{\rightarrow}$ on $\lambda_0^{\rightarrow}$-terms is defined by induction over the structure of $\tau$-types:

**Definition 5.11** $\mathcal{I}_\tau(t)$ Let $\vdash t : \tau \in \lambda_0^{\rightarrow}$

- $\mathcal{I}_{\mathbf{1}}(t) \iff \left[t^L \Downarrow^M () \iff \mu_!(t^L) \Downarrow^M ()\right]$
- $\mathcal{I}_{A\rightarrow B}(t) \iff \forall u \in \mathcal{I}_A . \mathcal{I}_B((t\,u))$

◁

Implementability defined in this way is an example of a *logical predicate* (cf. [Mit90], section 3) due to the way it is extended to cover terms of functional type. The key property is expressed by the second clause which says that a $\lambda_0^{\rightarrow}$-term of functional type is implementable if and only if it takes implementable arguments to implementable results.

**Lemma 5.12 Properties of $\mathcal{I}_\tau$** Let

$$\vdash \lambda x.t : \sigma \rightarrow \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \mathbf{1} \in \lambda^{\rightarrow}$$

and $T_1 \in \mathcal{I}_{\tau_1}, \ldots, T_n \in \mathcal{I}_{\tau_n}$. Then

1. $\mathcal{I}_{\mathbf{1}}(())$
2. $[\forall u \in \mathcal{I}_\sigma . \mathcal{I}_{\mathbf{1}}((t[x/u]\,T_1 \ldots T_n)) \Rightarrow \forall u \in \mathcal{I}_\sigma . \mathcal{I}_{\mathbf{1}}(((\lambda x.t)\,u T_1 \ldots T_n))]$
3. $[\forall u \in \mathcal{I}_\sigma . \mathcal{I}_\tau(t[x/u]) \Rightarrow \forall u \in \mathcal{I}_\sigma . \mathcal{I}_\tau(((\lambda x.t)\,u))]$ for arbitrary types $\tau$.

◁

The meaning (and truth) of the first proposition is obvious. The second proposition essentially states for **1**-typed terms that if lazy reduction (by substitution) is implementable then eager reduction is implementable. In proposition three this statement is generalized to terms of arbitrary $\tau$-type.

**Proof Lemma 5.12**

1. Immediately from $()^L = \mu_!(()^L) = ()$.

2. Fix an arbitrary n. Assuming

$$\left[(t[x/u]\,T_1 \ldots T_n)^L \Downarrow^M () \iff \mu_!((t[x/u]\,T_1 \ldots T_n)^L) \Downarrow^M ()\right] \qquad (5.8)$$

calculate

$$\begin{aligned}
(\lambda x.t\,uT_1 \ldots T_n)^L \Downarrow^M () &\iff (\lambda x.t^L\,!(u^L)!(T_1^L) \ldots !(T_2^L)) \Downarrow^M () && (\mathrm{Def.}(.)^L)\\
&\iff (t[x/!(u^L)]\,!(T_1^L) \ldots !(T_2^L)) \Downarrow^M () && (\text{Coroll. 5.9})\\
&\iff (t[x/u]\,T_1 \ldots T_n)^L \Downarrow^M () && (\mathrm{Def.}(.)^L)\\
&\iff \mu_!((t[x/u]\,T_1 \ldots T_n)^L) \Downarrow^M () && (\text{by Ass. (5.8)})\\
&\iff \mu_!((\lambda x.t\,uT_1 \ldots T_n)^L) \Downarrow^M () && (\text{Lemma 5.7})
\end{aligned}$$

which yields

$$\left[(\lambda x.t\,uT_1 \ldots T_n)^L \Downarrow^M () \iff \mu_!((\lambda x.t\,uT_1 \ldots T_n)^L) \Downarrow^M ()\right]$$

as required.

3. Prop. 3 follows from

$$[\forall u \in \mathcal{I}_\sigma.\mathcal{I}_\tau(t[x/u]N_1 \ldots N_k) \Rightarrow \forall u \in \mathcal{I}_\sigma.\mathcal{I}_\tau(((\lambda x.t)\,uN_1 \ldots N_k))] \qquad (5.9)$$

which can be shown by structural induction on types:

- $\tau = \mathbf{1}$. Then the implication 5.9 follows immediately from prop. 2.
- $\tau = \tau_1 \to \tau_2$. Assume that 5.9 holds for types $\tau_1$ and $\tau_2$ and assume

$$\forall u \in \mathcal{I}_\sigma.\mathcal{I}_{\tau_1 \to \tau_2}(t[x/u]N_1 \ldots N_k)$$

Let $N_{k+1} \in \mathcal{I}_{\tau_1}$. Then by the definition of $\mathcal{I}_\tau$

$$\forall u \in \mathcal{I}_\sigma.\mathcal{I}_{\tau_2}(t[x/u]N_1 \dots N_k N_{k+1})$$

Thus

$$\forall u \in \mathcal{I}_\sigma.\mathcal{I}_{\tau_2}(((\lambda x.t)\, u N_1 \dots N_k N_{k+1}))$$

by the induction hypothesis.
By the fact that $N_{k+1} \in \mathcal{I}_{\tau_1}$ was arbitrary

$$\forall u \in \mathcal{I}_\sigma.\mathcal{I}_{\tau_1 \to \tau_2}(((\lambda x.t)\, u N_1 \dots N_k))$$

as required.

□

The key technical lemma from which theorem 5.10 may be concluded may now be stated.

**Lemma 5.13** Let $x_1 : A_1, \dots, x_n : A_n \vdash t : A \in \lambda^{\to}$ with $Fv(t) = \{x_1, \dots, x_n\}$. If $\vdash t_i : A_i \in \lambda^{\to}$ and $\mathcal{I}_{A_i}(t_i)$ for $i = 1, \dots, n$ then

$$\mathcal{I}_A(t[x_1/t_1, \dots, x_n/t_n])$$

◁

**Proof Lemma 5.13** The proof is by induction over a derivation in $\lambda^{\to}$. Let $\Delta \vdash t : A \in \lambda^{\to}$. Case $\Delta \vdash t : A \equiv$

- $x : A \vdash x : A$. Suppose $\mathcal{I}_A(t)$ for some $t : A$. Immediately $\mathcal{I}_A(x[x/t])$.
- $\vdash () : \mathbf{1}$. Then $\Delta = \emptyset$ and $\mathcal{I}_{\mathbf{1}}(())$ by proposition (1) in lemma 5.12.
- $x_1 : A_1, \dots, x_n : A_n \vdash \lambda x.t : A \to B$ obtained from a sequent $x_1 : A_1, \dots, x_n : A_n, x : A \vdash t : B$ by the $[\to_I]$-rule. Suppose $\mathcal{I}_{A_1}(t_1), \dots, \mathcal{I}_{A_n}(t_n)$ for $t_1 : A_1, \dots, t_n : A_n$. Moreover, suppose $\mathcal{I}_A(t)$ for some $t : A$. Then $\mathcal{I}_B(t[x/t, x_1/t_1, \dots, x_n/t_n])$ by the induction hypothesis. It follows by prop. (3) in lemma 5.12 that $\mathcal{I}_B((\lambda x.t\, u)[x_1/t_1, \dots, x_n/t_n])$ as required.
- $x_1 : A_1, \dots, x_n : A_n \vdash (f\, t) : B$ obtained from sequents $x_1 : A_1, \dots, x_n : A_n \vdash f : A \to B$ and $x_1 : A_1, \dots, x_n : A_n \vdash t : B$ by applying the $[\to_E]$-inference rule. Suppose $\mathcal{I}_{A_1}(t_1), \dots, \mathcal{I}_{A_n}(t_n)$ for $t_1 : A_1, \dots, t_n : A_n$.

  By the induction hypothesis:

$$\mathcal{I}_{A\to B}(f[x_1/t_1,\dots,x_n/t_n]) \text{ and } \mathcal{I}_A(t[x_1/t_1,\dots,x_n/t_n])$$

By definition 5.11:

$$\mathcal{I}_{A\to B}(t) \iff \forall u \in A.\mathcal{I}_A(u) \Rightarrow \mathcal{I}_B((t\,u)).$$

Taking $t = f[x_1/t_1,\dots,x_n/t_n]$ and $u = t[x_1/t_1,\dots,x_n/t_n]$ yields

$$\mathcal{I}_B((f[x_1/t_1,\dots,x_n/t_n]\,t[x_1/t_1,\dots,x_n/t_n])).$$

By the definition of term substitution $\mathcal{I}_B((f\,t)[x_1/t_1,\dots,x_n/t_n])$ as required.

- $x_1 : A_1,\dots,x_n : A_n \vdash \mathbf{fix}(x,t) : A$ obtained from a sequent $x_1 : A_1,\dots,x_n : A_n, x : A \vdash t : A$ by the $[fix]$-rule. The reasoning is analogous to the case of an abstraction.

□

**Proof Thm. 5.10** Taking $t$ closed and of type **1** the claim of theorem 5.10 follows. □

Immediately following from simulation (lemma 5.1) one obtains:

**Corollary 5.14 Implementability for cbn**

$$\forall t \in \lambda_0^{\to}.t \Downarrow^N () \iff \mu_!(t^L) \Downarrow^M ()$$

◁

Finally, the main result of this thesis is obtained. It states that removing *logically* superflous [!]-operators in $(\lambda^{\to})^L$ does not invalidate the simulation property of $(\lambda^{\to})^L$ with respect to call-by-name reduction. Otherwise said, $\mu_!(.)$ could have been shown to be a safe method for optimizing the execution of call-by-name reduction.

## 5.3 Discussion

In the light of the above result the following observations can be made.

### 5.3.1 Consequences regarding strictness analysis

#### 5.3.1.1 Relationship to traditional strictness analysis

The astute reader will have noticed, that we never try to determine wether

$$(f\ \perp) = \perp$$

in order to find out whether an argument is strict, as do traditional approaches to strictness analysis.

This traditional approach can become very costly (cf. [PJ87]), in particular when analyzing recursive functions. Instead we have developed a sufficient, entirely syntactic condition, namely linearity under $\mu_!(.)^L$, that implies this equation for linear arguments.

#### 5.3.1.2 Linearity does not *always* imply strictness

The claim, that *any* linear argument may be reduced eagerly without corrupting call-by-name termination semantics, will now be examined. This claim is based on the intuition, that linearity implies the eventual evaluation of the argument since no linear value may be discarded. It was shown above that this is indeed true. However, faithfulness to call-by-name semantics is only given in $(\lambda^{\rightarrow})^L$-proof structures.

Consider the term

$$(\lambda x.(\lambda y.()\,(x\,()))\,\mathbf{\Omega})$$

whose call-by-name normal form exists by the derivation

$$\dfrac{\dfrac{}{\lambda x.(\lambda y.()\,(x\,()))\Downarrow^N \lambda x.(\lambda y.()\,(x\,()))} \qquad \dfrac{\dfrac{}{\lambda y.()\Downarrow^N \lambda y.()} \qquad \dfrac{}{()[y/(\mathbf{\Omega}\,())]\Downarrow^N ()}}{(\lambda y.()\,(\mathbf{\Omega}\,()))\Downarrow^N ()}}{(\lambda x.(\lambda y.()\,(x\,()))\,\mathbf{\Omega})\Downarrow^N ()}$$

It may be assigned a non-$(\lambda^{\rightarrow})^L$ term in $\lambda^{!,\multimap}$ with *linear* formal parameter $x$, namely

$$(\lambda x.(\lambda y.\mathbf{drop}\; y\;\mathbf{in}\;()\,(x\,()))\,\mathbf{\Omega}^L)$$

whose $\Downarrow^M$-reduction does not terminate since the linear argument $\mathbf{\Omega}^L$ has no normal form.

#### 5.3.1.3 There is no strictness type

The decisive restriction that has been introduced to capture faithful implementations was introduced as a restriction on the structure of typing proofs. One might wonder, wether this restriction is too narrow, wether the minimal linear type, assigned to a $\lambda_0^{\rightarrow}$-term under $\mu_!(.^L)$ does in itself characterize all correct implementations. In other words, can the question of faithful implementation already be settled on the level of *types*, such that *all* proofs that are members of that type yield faithful implementations of the original term?

**Example 5.15 (No strictness type)**

Consider the term

$$P \equiv (\lambda x.()\,\mathbf{\Omega})$$

having type **1**. It may be translated as

$$\begin{array}{rcl} \mu_!(P^L) & \equiv & (\lambda x.\mathbf{drop}\ x\ \mathbf{in}\ ()\ !(\mathbf{\Omega}^L)) \\ \mu_!(P^E) & \equiv & (\lambda x.\mathbf{drop}\ x\ \mathbf{in}\ ()\ \mathbf{\Omega}^E) \end{array}$$

where both $\mu_!(P^L)$ and $\mu_!(P^L)$ have the same type **1** and the formal parameter $x$ has type **!1** in both instances.

The $\Downarrow^M$-reduction of $\mu_!(P^L)$ terminates as shown in example 5.4. The $\Downarrow^M$-reduction of $\mu_!(P^E)$ however doesn't:

$$\dfrac{\dfrac{}{\lambda x.\mathbf{drop}\ x\ \mathbf{in}\ () \Downarrow^M \lambda x.\mathbf{drop}\ x\ \mathbf{in}\ ()} \quad \dfrac{\dfrac{\begin{array}{c}\vdots\ \infty \\ \mathbf{\Omega}^E \Downarrow^M c\end{array}}{\mathbf{\Omega}^E \Downarrow^M c}}{\mathbf{\Omega}^E \Downarrow^M c} \quad \dfrac{}{\mathbf{drop}\ x\ \mathbf{in}\ ()[x/c] \Downarrow^M d}}{(\lambda x.\mathbf{drop}\ x\ \mathbf{in}\ ()\ \mathbf{\Omega}^E) \Downarrow^M d}$$

(Note the analogy to the $\Downarrow^V$-reduction of $P$ discussed in example 5.4.) ◁

### 5.3.2 Higher order functions

Clearly, the methods developed above are strong enough to analyze the strictness of higher order functions, as long as they are fully applied and the resp. term is of type **1**. Can higher-order functions also be studied in isolation?

Let

$$apply : \tau_! = \lambda f.\lambda x.(f\,x) : (A \multimap B) \multimap A \multimap B \tag{5.10}$$

From the information that $x$ is linear, is it safe to conclude that *apply* is always strict in its second argument?

Certainly not, since that depends on how it is *used* by the input function $f$. This means that the status of $x$ can not be determined until the first argument to *apply* is actually known.

### 5.3.3 Polymorphism

The applicability of type-based strictness analysis - as defined above - in languages as ML (cf. [MT91, MTH90]), whose type system is extended by a weak form of polymorphism remains to be shown.

An intuitive argument in favor of such a claim might simply appeal to the soundness of the resp. rules for type abstraction and instantiation in $\lambda^{!,\multimap}$: (non-)linearity information may not be altered in an unsound way, all possible types of a term must respect its structure.

On the grounds of this it is conjectured that the integration of polymorphism poses no additional problems.

### 5.3.4 Efficiency

Regarding efficiency, some gains are made when $\mu_!(.^L)$ identifies arguments that can be reduced prior to substitution. However, a large class of possible optimizations is not realized due to the fact that not only discarded arguments are delayed, but also contracted ones. These could - with minor restrictions - be equally well reduced eagerly without corrupting call-by-name termination semantics. The drawbacks of the present method are illustrated in figure 5.3.4. Let $I \equiv \lambda x.x$.

From left to right, figure 5.3.4 depicts three alternative reductions to normal form issuing from the term:

$$(\lambda c.\lambda w.c\,(c\,I)\,(I\,I)^1)\,(I\,I)^2$$

The by-name strategy, unlike the by-value strategy, treats redex (2) optimally by discarding it without prior evaluation. However, in the treatment of redex (1) call-by-value performs better. Both need more steps than the optimal strategy.

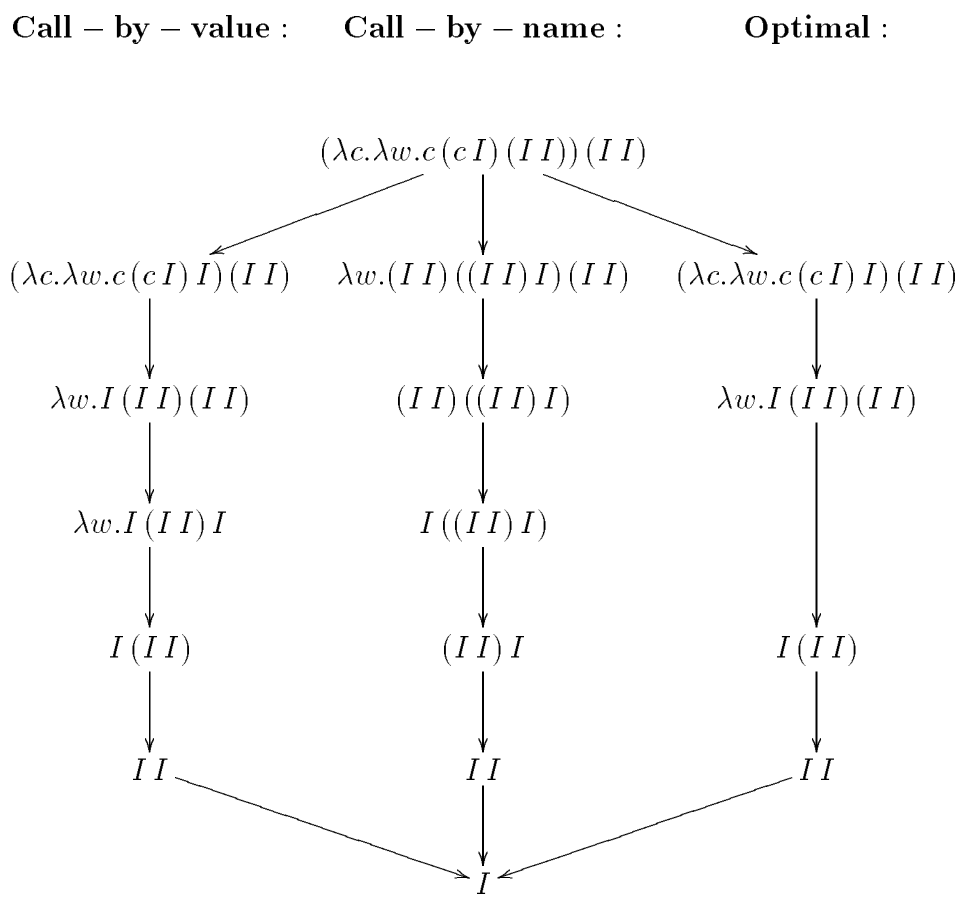


Figure 5.1: Comparing cbn, cbv and optimal reduction

- In the case of call-by-name, this may be attributed to the fact that arguments to functions are generally not reduced which means that computations are not shared when the argument is duplicated.

- Call-by-value performs worse than the optimal strategy because even unneeded arguments are reduced.

This shows, that neither $\Downarrow^N$, which delays too much, nor $\Downarrow^V$, which doesn't delay enough redex contractions, yield optimal reduction sequences.

The $\Downarrow^M$-reduction that is induced by the $(.)^L$-translation coincides with the call-by-name reduction. Prior minimization does not yield any improvements since both redexes are necessarily nonlinear.

The exact replication of this inefficient reduction can be attributed to the fact that $\lambda^{!,\multimap}$ does not distinguish discarded from copied terms. Both categories of terms are delayed, which is efficient in the case of weakening but inefficient in the case of contraction.

What is needed therefore is a more versatile interpretation of the [!]-operator.

How to modify the present framework in order to make such distinctions will be considered in the next chapter.

## 5.4 Related work

The line of investigation pursued until this chapter has been first put forward by S. Abramsky in [Abr93]. The basic method applied to obtain the final result, logical predicates, is exposed e.g. in [Mit90]. It is basically needed to obtain a strong enough induction hypothesis such that the key technical lemma goes through. Strictness analysis of higher order functions in a framework based on denotational semantics was considered in [BHA86]. The present treatment remains on the syntactic level, while using essentially the same method (logical relations) in order to accomodate higher order terms. Strictness analysis in the presence of polymorphism was studied in [Abr85]. The utility of such notions as 'polymorphic invariance' in our framework must remain unexplored.

# Chapter 6

# Extensions and future work

The alternative reductions exhibited in figure 5.3.4 have demonstrated that additional optimizations become possible by means of a principled distinction between discarded and duplicated sub-terms.

Only discarded sub-terms should be delayed while contracted (more generally: *needed*) sub-terms should be reduced right-away.

This operational distinction cannot sufficiently be supported in the present framework. As a remedy, it is proposed

- to annotate every nonlinear sub-term $!(t)$ with information indicating whether it is actually needed, as in $!^{\mathbf{n}}(t)$, and
- to *extend* the present $\Downarrow^M$-semantics with the following rule

$$\frac{t \Downarrow^M t'}{!^{\mathbf{n}}(t) \Downarrow^M !(t')} \qquad \text{(by-need)}$$

  The resulting reduction semantics will henceforth be referred to by the symbol $\Downarrow^M_N$.

It remains to be shown how $c - \lambda^{!,\multimap}$ may be adapted to reflect neededness.

## 6.1 Needed reductions

The constraints derived in $c - \lambda^{!,\multimap}$ were intended to distinguish necessarily nonlinear from essentially linear sub-terms. A different constraint system is needed to separate needed from discarded sub-terms.

Since $[w!]$- and $[c!]$-inferences may *interact* in subtle ways, *it does not suffice just to omit the introduction of the contraction constraint* to characterize all discarded subproofs. As example 2.17 in chapter 2 from above has shown, a duplicated subproof may be completely discarded by subsequent weakenings of its duplicates. Such subproofs must be also be marked as discarded, which requirement can be met by introducing a *weakening-oriented* constraint through the $[c!]$-rule.

Consider - ceteris paribus - the following replacement for the $[c!]$-rule in $c - \lambda^{!,\multimap}$:

$$\frac{\Delta \vdash t_1 :!^{u_3} A[\mathcal{C}] \quad \Gamma, x :!^{u_1} A, y :!^{u_2} A \vdash t_2 : B[\mathcal{D}]}{\Delta \vdash \textbf{copy } t_1 \textbf{ as } x/y \textbf{ in } t_2 : B[\mathcal{C} \wedge \mathcal{D} \wedge (u_1 \wedge u_2 \rightarrow u_3)]} \; c! \tag{6.2}$$

The resulting system is named $c_w - \lambda^{!,\multimap}$. The constraints derived in $c_w - \lambda^{!,\multimap}$ will be called *pure weakening constraints.*

The new contraction inference only introduces a transitivity condition $(u_1 \wedge u_2 \rightarrow u_3)$ which says that whenever both duplicates of a contracted subproof are nonlinear (due to weakening), the subproof itself is too.

The transitivity condition is coded in such a way that a contracted subproof of which only one duplicate is weakened, is not marked:

$$u_1 \wedge (u_1 \wedge u_2 \rightarrow u_3) \not\models u_3$$

This is plausible since the other duplicate will obviously survive.

Minimization using the constraints derivable in $c_w - \lambda^{!,\multimap}$ is clearly unsound with respect to provability in $\lambda^{!,\multimap}$, since contracted subproofs are not captured anymore. However, all discarded subproofs may be *recognized* as such, which was the sole intention in introducing rule 6.2.

Using pure weakening constraints $\mathcal{C}_w$ derived in $c_w - \lambda^{!,\multimap}$, a rule for annotating needed sub-terms is given by:

$$\text{mark}(\mathbf{!}(t)^u) = \begin{cases} \mathbf{!}(t) & \text{if } \mathcal{C}_w \models u \\ !^{\mathbf{n}}(t) & \text{otherwise} \end{cases}$$

It may be applied after translating a $\lambda_0^{\rightarrow}$-term with the effect that nonlinear sub-terms are marked as needed if and only if they are used in the ensuing reduction.

This property is the key to optimizing $\Downarrow^N$-reduction beyond the capabilities of $\mu_!(.)$. Under $\Downarrow^M_N$, not only linear subterms will be evaluated eagerly, but also duplicated ones which have been delayed under $\Downarrow^M$.

From the observation that lemma 5.5 in the previous chapter really holds because no less than the termination of *needed substitutions* is implied by the termination of the ambient term, the following may be claimed:

**Theorem 6.1 Faithfulness of $\Downarrow_N^M$**

$$\forall \vdash t : \mathbf{1} \in \lambda_0^{\rightarrow}.t \Downarrow^N () \iff \mathrm{mark}(t^L) \Downarrow_N^M ()$$

◁

Note that minimization must not be employed here, since essentially linear subterms will also be marked as needed by the annotation rule.

The reader is reminded that this is only a plausible claim. A possible proof of this theorem could proceed as the one given in the previous chapter for theorem 5.10, where it is proposed to use the following variant of lemma 5.5:

**Lemma 6.2**

Let $x_1 :!A_1, \ldots, x_n :!A_n \vdash t : B \in \lambda^{!,\multimap}, \mathbf{!}(u_1) :!A_1, \ldots, \mathbf{!}(u_n) :!A_n \in \lambda^{!,\multimap}$ and let $\mathbf{!}(u_i), i = 1 \ldots n$ all be *needed* copies of another. Then

$$\mathrm{mark}(t[x_1/\mathbf{!}(u_1) \ldots x_n/\mathbf{!}(u_n)]) \Downarrow_N^M N \wedge \mathbf{!}(u_i) \notin St(N) \Rightarrow \mathrm{mark}(\mathbf{!}(u_i)) \Downarrow_N^M u'$$

for $i \in \{1, \ldots, n\}$. It is understood that the annotations of the $\mathbf{!}(u_i)$ in the conclusion comply with the annotations of the $\mathbf{!}(u_i)$ in the premise of the implication. ◁

**Proof** By induction over a $\Downarrow_N^M$-derivation.

The only case in the proof of this lemma that is really more complicated than the corresponding case in the proof of lemma 5.5 is the reduction of a **drop** operator.

Let

$$\mathrm{mark}((\mathbf{drop}\, t_1 \,\mathbf{in}\, t_2)[x_1/\mathbf{!}(u_1) \ldots x_n/\mathbf{!}(u_n)]) \Downarrow_N^M N$$

Assume

$$\mathrm{mark}(\mathbf{!}(u_i)) = \mathbf{!^n}(u_i) \tag{6.3}$$

and

$$Fv(t_1) = \{x_1, \ldots, x_k\}_{k \leq n}$$

without loss of generality. The $\Downarrow_N^M$-reduction semantics give

$$\mathrm{mark}(t_1[x_1/\mathbf{!}(u_1) \ldots x_k/\mathbf{!}(u_k)]) \Downarrow_N^M \mathbf{!}(t_1')$$

and

$$\mathrm{mark}(t_2[x_{k+1}/\mathbf{!}(u_{k+1}) \ldots x_n/\mathbf{!}(u_n)]) \Downarrow_N^M N \tag{6.4}$$

If $\mathbf{!}(u_i)_{1 \leq i \leq k} \notin St(\mathbf{!}(t_1'))$ then the induction hypothesis applies. Otherwise assume

$$\{\mathbf{!^n}(u_1), \ldots, \mathbf{!^n}(u_j)\}_{j \leq k} \subset St(\mathbf{!}(t_1'))$$

without loss of generality.

If $j < k$ then the induction hypothesis applies. Otherwise it must that $k \neq n$ since the $!^{\mathbf{n}}(u_i)$ are needed by assumption 6.3 and this would clash with the transitivity condition of rule 6.2. This means that

$$!^{\mathbf{n}}(u_i)_{k+1 \leq i \leq n} \Downarrow_N^M u'$$

can be inferred by applying the induction hypothesis to 6.4.

(Note that the convergence of only one $!^{\mathbf{n}}(u_i)$ trivially implies the convergence of all of its copies.) □

## 6.2 Implementation issues

The optimizations proposed in this and the previous chapter must clearly be tested in an actual implementation against other optimization techniques. Since this work is based on the linear typing calculus of Abramsky [Abr93], which has already been implemented by his student Mackie [Mac91], an implementation need only be concerned with the translation, constraint assignment and minimization procedures which are new.

It is conjectured that the entailment relation between constraints and use-variables may be decided very efficiently, since these constraints have a particularly simple structure.

## 6.3 Summary

These considerations are only preliminary and need further investigation. However, if they succeed, the present method could indeed evolve into a highly efficient tool for optimizing call-by-name reduction.

# Chapter 7

# Conclusions

This work has studied aspects of the general relationship between the systems $\lambda^{\rightarrow}$ and $\lambda^{!,\multimap}$.

After noticing that this relationship is not functional, we decided to investigate the different translations by themselves. These investigations were supposed to judge alternative translations by their applicability to practical problems.

One translation was studied in particular, the $(.)^L$-translation. It was shown that using this translation the call-by-name reduction semantics of $\lambda_0^{\rightarrow}$-terms could be simulated using $\lambda_0^{!,\multimap}$ and its associated notion of reduction.

Moreeover, we obtain the interesting result that the notions of syntactic structure, logical necessity of storage operators and semantic properties of terms are closely related. This is reflected in a theorem stating that logically superflous [!]-operators may be removed from $(\lambda^{\rightarrow})^L$-terms without corrupting the fact that these terms 'implement' call- by-name reduction.

The last chapter proposes a more powerful formalism to obtain even more efficient implementations. However, the final proofs must be left to future work.

# Bibliography


[Abr85] S. Abramsky. Strictness analysis and polymorphic invariance. In *Programs as Data Objects*, pages 1–23. Springer-Verlag LNCS 217, 1985.

[Abr93] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[Bar84a] Hendrik Pieter Barendregt. *The lambda calculus : its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. Elsevier/North-Holland, Amsterdam, rev. ed., 2nd printing 1985 edition, 1984.

[Bar84b] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1984.

[Bar90] H.P. Barendregt. Functional programming and lambda calculus. In J.v.Leeuwen, editor, *Handbook of Theoretical Computer Science Vol.B*, part 7, pages 321–363. Elsevier Science Publishers, 1990.

[Bar92] H.P. Barendregt. Lambda calculi with types. In S.Abramsky, D.M.Gabbay, and T.S.E.Maibaum, editors, *Handbook of Logic in Computer Science Vol.2*, chapter 2, pages 117–309. Clarendon Press, 1992.

[BBHdP92] N. Benton, G. Bierman, J. M. E. Hyland, and V. de Paiva. Term assignment for intuitionistic linear logic. Technical Report 262, Computer Laboratory, University of Cambridge, August 1992.

[BHA86] Geoffrey L. Burn, C. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, (7):249–278, 1986.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, (5):56–68, 1940.

[Cur34] Curry,
H. B. Functionality in combinatory logic. *Proc.Nat.Acad.Science USA*, (20):584–590, 1934.

[DJS95] V. Danos, J.-B. Joinet, and H. Schellinx. On the linear decoration of intuitionistic derivations. In *Archive for Mathematical Logic*, volume 33, pages 387–412, 1995.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. *POPL*, pages 207–212, 1982.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[How80] W.A. Howard. The formulae-as-types notion of construction. In J.R.Hindley and J.P.Seldin, editors, *To H.B.Curry:Essays on combinatory logic, lambda calculus and formalism*, chapter 3. Academic Press, 1980.

[HS93] J. Roger Hindley and Jonathan P. Seldin. *Introduction to combinators and lambda-calculus*, volume 1 of *London Mathematical Society student texts*. Cambridge Univ. Press, New York, NY, reprinted edition, 1993.

[Kah87] G. Kahn. Natural semantics. In *Symposium on Theoretical Aspects of Computer Science*, number 247 in Lecture Notes in Computer Science, pages 22–39. Springer, 1987.

[LM92] P. Lincoln and J. Mitchell. Operational aspects of linear lambda calculus. In *Seventh Annual Symposium on Logic in Computer Science*, pages 235–246, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[Mac] I. Mackie. *The Geometry of Implementation*. PhD thesis, Imperial College of Science, Technology and Medicine.

[Mac91] I. Mackie. Lilac — a functional programming language based on linear logic. Master's thesis, Imperial College, London, 1991.

[Mit90] J.C. Mitchell. Type systems for programming languages. In J.v.Leeuwen, editor, *Handbook of Theoretical Computer Science Vol.B*, part 8, pages 365–458. Elsevier Science Publishers, 1990.

[MT91] Robin Milner and Mads Tofte. *Commentary on standard ML*. MIT Press, Cambridge, Mass., 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The definition of standard ML*. MIT Press, Cambridge, Mass., 1990.

[PJ87] Simon L. Peyton Jones. *The implementation of functional programming languages*. Prentice Hall, New York, 1987.

[Rea89] C. Reade. *Elements of functional programming*. Addison Wesley, 1989.

[Sce93] A. Scedrov. A brief guide to linear logic. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 377–394. World Scientific Publishing Company, 1993. Also in Bulletin of the European Association for Theoretical Computer Science, volume 41, pages 154–165.

[T9̈3] Andreas Tönne. Linear logic meets the Lambda calculus, part I. Technical Report MPI-I-93-258, Max-Planck-Institut fuer Informatik, Saarbruecken, 1993. To appear.

[Tro92] A. S. Troelstra. *Lectures on Linear Logic*. CSLI Lecture Notes 29, Center for the Study of Language and Information, Stanford, California, 1992.

[Tro95] A.S. Troelstra. Natural deduction for intuitionistic linear logic. *Annals of pure and applied logic*, 73(1):79–108, 1995.

[Wad91a] P. Wadler. Is there a use for linear logic? In *Proceedings of the Symposium on Partial Evaluations and Semantics-Based Program Manipulation*, pages 255–273, New Haven, Connecticut, June 1991. Also in SIGPLAN Notices; vol.26, no.9; September 1991.

[Wad91b] P. Wadler. There's no substitute for linear logic. Manuscript, December 1991.

[Wad93] P. Wadler. A syntax for linear logic. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings of the Ninth Conference on Mathematical Foundations of Programming Semantics*, pages 513–529, New Orleans, Louisiana, April 1993. Springer-Verlag LNCS 647.

[Win93] Glynn Winskel. *The formal semantics of programming languages : an introduction*. Foundations of computing. MIT Press, Cambridge, MA, 1993.