View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Kent Academic Repository

Kent Academic Repository

Full text document (pdf)

Citation for published version

Smith, Connor Lane (2017) Optimal Sharing Graphs for Substructural Higher-order Rewriting Systems. Doctor of Philosophy (PhD) thesis, University of Kent,.

DOI

Link to record in KAR

https://kar.kent.ac.uk/63884/

Document Version

UNSPECIFIED

Copyright & reuse

Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research

The version in the Kent Academic Repository may differ from the final published version.
Users are advised to check **http://kar.kent.ac.uk** for the status of the paper. **Users should always cite the published version of record.**

Enquiries

For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at **http://kar.kent.ac.uk/contact.html**

KAR Kent Academic Repository

University of Kent

OPTIMAL SHARING GRAPHS FOR SUBSTRUCTURAL HIGHER-ORDER REWRITING SYSTEMS

a thesis submitted to
The University of Kent
in the subject of computer science
for the degree
of doctor of philosophy.

By
Connor Lane Smith
January 2017

Contents

List of Figures

Abstract

The notion of *optimal reduction* was introduced by Lévy (1980) in the context of the untyped λ-calculus, based on the concept of *families* of reducible expressions. It took more than a decade for an algorithm achieving this optimal reduction to be discovered, introduced by Lamping (1990) and then refined by Gonthier, Abadi & Lévy (1992). The existence of an analogous algorithm for higher-order term rewriting systems was later theorised by Van Oostrom (1996), but has of yet been unrealised.

We provide such an algorithm by defining a class of higher-order rewriting systems having Intuitionistic Linear Logic (Benton, Bierman, de Paiva & Hyland 1992) as a substitution calculus, in the sense of Van Oostrom (1994), and introduce a method of translating terms and rules into equivalent Lamping–Gonthier sharing graphs. Our system thus offers a generalisation of the mechanism for optimal reduction from second- to higher-order term rewriting systems. Moreover, in the case of match-sequential systems, we provide a specific reduction strategy, as we are able to effectively identify needed redexes.

Finally, we explore briefly the subtleties and complexities of applying the technique to various other term rewriting system, such as those with alternative substructural or polymorphic type systems, those with generalised patterns on the left-hand side, and those with rationally infinite terms. All these systems are built upon the same fundamental translation of λ-terms to sharing graphs.

Acknowledgements

First of all, I would like to thank my supervisor, Stefan Kahrs, for offering me the opportunity to work on this research project, and for indulging my interest in the admittedly niche topic of higher-order Lévy optimality. It has certainly been very interesting, albeit much harder work than I had imagined.

I would also like to thank Simon Thompson and Scott Owens, the remainder of my supervisory panel, for their time and input. Thanks too to my examiners, Maribel Fernández and Olaf Chitil, whose feedback on this thesis has improved it significantly.

I am deeply grateful to my parents, Philip and Gina, to whom I owe everything, for their endless support and encouragement.

Finally, my greatest thanks to Julie Skevik, *min kjæreste*, for her patience and support while I was working on this research, through good times and bad.

In memory of Randi Bauge Skevik (1952 – 2015).

Chapter 1

Introduction

The notion of optimal reduction was first conceived by Lévy (1980) for the untyped λ-calculus. It took more than a decade for an effective algorithm to be developed to provide the sophisticated sharing structures needed for such optimal reduction, introduced in an incomplete form by Lamping (1990) and then refined by Gonthier et al. (1992). This evaluator was later generalised to a particular class of second-order term rewriting systems, called Interaction Systems (Laneve 1993). A translation also exists from match-sequential constructor systems to Discrete Interaction Systems, the first-order fragment of Interaction Systems (Fernández & Mackie 1996). However, these techniques remain limited to first- and second-order term rewriting.

The question of whether it is possible to implement an evaluator for higher-order rewriting systems[1] has, to our knowledge, remained an open problem since it was posed by Van Oostrom (1996) — see §1.2.1. We provide such an evaluator by defining a class of higher-order rewriting systems having Intuitionistic Linear Logic (Benton et al. 1992) as a substitution calculus, in the sense of Van Oostrom (1994), and introduce a method of translating terms and rules into equivalent Lamping–Gonthier sharing graphs. Our system thus generalises this optimal evaluator from second-order Interaction Systems and match-sequential constructor systems, to substructural higher-order rewriting systems.

[1] We use "higher-order" in the sense of *arbitrary* orderedness, not restricted to (e.g.) second order rewriting; see §1.2.

1.1 Lévy optimality

When we wish to discuss whether a given algorithm is "efficient," we must always first adopt some particular *cost model* for use as a measurement of this efficiency. A common approach is to classify an algorithm according to bounds on the growth rate of its function, generally with Bachmann–Landau notation; but this describes only the shape of the curve, leaving undefined the actual values of constants involved, so two algorithms of the same complexity class cannot then be compared and contrasted. When it comes to term rewriting systems we might define the cost model in terms of numbers of rewrite steps, although this assumes that the cost of all rewrite steps is effectively the same.

The notion of *optimal reduction* was formalised for the λ-calculus by Lévy (1980) in terms of 'redex families', which comprise the residuals of a single 'ancestral' redex, which may be duplicated (or indeed discarded) over the course of a term's reduction. Instead of dealing with the reduction of individual redexes at each rewrite step, Lévy considered the reduction of all redexes in a single family in one single step, called a 'family reduction'. These reductions are then considered to be *optimal* if only those families containing *needed* redexes are reduced, i.e. those that must be reduced in order for the term to reach its normal form (if it has one).

The mechanism for performing substitutions in the λ-calculus — called its *substitution calculus* by Van Oostrom (1994) — may be formalised in any number of ways, as evidenced by the various alternative calculi for 'explicit substitution' (Abadi, Cardelli, Curien & Lévy 1991). The implicit nature of substitution is one of λ-calculus's strengths, but also means that a cost model which has no specific substitution calculus in mind cannot take into account these 'hidden costs' behind each rewrite step (Lawall & Mairson 1996). In the case of optimal reduction of the λ-calculus, the hidden cost of the computation required to identify needed redex families may in fact swamp the cost of any β-reductions thus avoided.

Yet whilst the meaningfulness of Lévy optimality has been called into question with respect to the λ-calculus, this complaint does not necessarily hold for other rewriting systems. Even in the case of the λ-calculus there are cases in which

optimal reduction, including all underlying costs, is of a lower-bounded complexity class — $O(n)$ compared to $O(n^2)$ — than any more conventional reduction strategy (Asperti & Guerrini 1998). This fact is amplified in the case of arbitrary rewriting systems, where the complexity of a given rewrite step may not be a cheap operation, as in the case of the λ-calculus,[2] but rather may require vast amounts of computation (i.e. where the right-hand side of the rewrite is not yet known). The general technique for optimal reduction is therefore free of such criticism: although it might be unwise to strive for absolute Lévy optimality in every case, it is still of clear practical benefit in situations where it is reasonable to expect the cost of reduction steps to outweigh the costs of the sharing. It is in this context which our work is framed.

1.1.1 Optimal sharing

In the case of first-order term rewriting systems, assuming we can identify needed redexes during normalisation, a sharing mechanism for upholding optimal reduction is quite straightforward: terms are not represented as trees as they commonly are, but as *dags* (directed acyclic graphs). Residuals of a given subterm are represented as a shared subdag, and so any redexes contained within it may be reduced in that one place only, and hence as a family reduction. This is actually a more natural representation for terms in computer implementations, with a dag's edges represented as pointers into shared memory. If sharing is made explicit then the point at which a subdag is shared by two superdags is called a *fan*, and during reduction this fan may need to be moved out of the way to allow a redex to fire, an unsharing process termed 'unfolding'.

Wadsworth (1971) formalised a term sharing mechanism for the λ-calculus using dags, since termed 'lazy evaluation', which likely introduced the notion of using term graphs as a sharing mechanism. This form of sharing is illustrated in Figure 1. However, Wadsworth's mechanism for the substitution of bound variables was shown to be suboptimal in the sense of Lévy.[3] Specifically, when contracting a

[2]Cheap, that is, with respect to the β-step $(\lambda x.t)u \to t[x/u]$ without subsequent substitution propagation, α-renaming, etc., which is handled by the optimal reduction apparatus.

[3]Lazy evaluation has been shown to be optimal in the restricted case of the *weak* λ-calculus (Balabonski 2013), but we are interested in the 'full', strong λ-calculus.

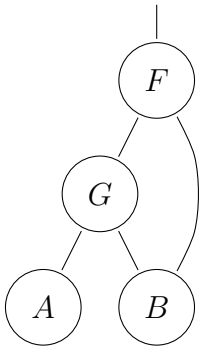


Figure 1: Wadsworth sharing of $F(G(A, B), B)$

β-redex whose abstraction and argument are separately shared, in order to perform the substitution of the bound variable in the body of the abstraction, it is necessary to duplicate the entire subdag. Of course, this also duplicates all redexes contained within that subdag, which undermines the optimality of the technique. Indeed, at the time when optimal reduction was introduced, no sharing mechanism was known which was Lévy-optimal.

It was another decade before a suitable optimal reduction algorithm was found for the λ-calculus, first devised by Lamping (1990)[4], and subsequently refined by Gonthier et al. (1992) in terms of the contemporary Interaction Nets (Lafont 1990). The essence of Lamping's algorithm is that as well as the 'positive' fans common to dags, which enable the sharing of common subterms, it also features 'negative' fans, separating distinct subterms within otherwise common subterms such that it is in fact *slices* of the term that are shared — see §4.2.1 for more details. This algorithm was later generalised from the λ-calculus to *Interaction Systems* (Laneve 1993), a class of term rewriting systems for which there exists a direct translation into Interaction Nets and thereby for which Lamping–Gonthier sharing is optimal.

Yet the rewrite rules of an Interaction System are rather overly restrictive, with its symbols being split into disjoint sets of constructors and destructors and each rule describing an interaction between exactly one constructor and one destructor, with no other symbols allowed on the left-hand side. Compared with the full freedom of orthogonal rewriting systems, this can be very limiting in its

[4]An alternative algorithm was independently discovered at the same time by Kathail (1990), although this has had relatively little impact on the field.

expressiveness — although still Turing-complete, individual rules are lacking in expressive power, requiring many individual rewrite steps in order achieve much computation, thus being something of a 'Turing tar-pit' (Perlis 1982). Moreover, Interaction Systems, being only second-order, lack the expressiveness required to define *third- or higher-order* rewrite rules.

1.2 Higher-order rewriting

Term rewriting systems have historically been described as "higher-order" with reference to a number of different attributes — and some systems that would be described as "higher-order" under one of its definitions would not qualify as being so under another definition — so it is important that we be clear on what it means in the present context. For instance, Interaction Systems, for which an optimal reduction strategy is already known, are described by Asperti & Laneve (1994) as "higher-order," appealing to an earlier claim by Klop (1980):

> It has been claimed by Klop that Combinatory Reduction Systems are the higher-order generalisation [of] Term Rewriting Systems. Then, Interaction Systems are to CRSs as Discrete Interaction Systems are to TRSs.

The term "higher-order" is used here to refer to how those systems have meta-level 'meta-variables' distinct from their terms' object-level bound variables — a distinction that first-order TRSs, having no bound variables, do not have. However, in the terminology of the later *higher-order term rewriting systems* (Wolfram 1990, Nipkow 1991, Van Oostrom 1994), CRSs and ISs are not truly higher-order, as the 'substitutes' (Kahrs 1994) used in these systems' substitution calculi are λ-functions of at most *second-order* (Van Oostrom & Van Raamsdonk 1994). Higher-order rewriting systems, whose substitution calculi permit as substitutes any simply-typed λ-function, are in this sense of a strictly higher order.

The confusion in terminology is exacerbated by the distinction between the object-level rewrite steps and meta-level substitutions. Second-order term rewriting is sufficient for the definition of untyped λ-calculus, and so Combinatory Reduction Systems can be used to *simulate* the higher-order, but if it were sufficient

to call any term rewriting system able to simulate the untyped λ-calculus "higher-order" then first-order rewriting systems would also suffice. Indeed, explicit substitution (Abadi et al. 1991) is precisely the 'demystification' of variable substitution by encoding it explicitly to a first-order term rewriting system. However, in being explicit about the substitution mechanism, with each rewrite step 'nudging' the substitution a little bit further down the term, then unless this mechanism is itself optimal it will undermine any optimality we may attempt to achieve with respect to the substitution of bound variables.

If we instead define rewriting in terms of higher-order rules, with implicit substitution, then sharing graphs are able to perform the role of substitution calculus, in the sense of Van Oostrom (1994), and thus do so optimally. This is, when all is said and done, the whole benefit of higher-order rewriting: the mechanics of variable substitution (as opposed to its semantics) are allowed to remain implicit. By using Lamping–Gonthier graphs for this mechanic instead of more straightforward substitution like that of the λσ-calculus, the system as a whole may benefit from their structural properties with regards to Lévy optimality.

1.2.1 Higher-order optimality

The theoretical groundwork for higher-order optimality was laid by Van Oostrom (1996), who generalised Lévy's (1980) redex families from the λ-calculus to higher-order term rewriting systems (§2.2.1). At the same time, he drew attention to the absence of an algorithm for optimal reduction in higher-order rewriting, and identified some potential avenues of research:

> The next question is whether it is possible to implement an evaluator for orthogonal Higher-order Rewrite Systems (HRSs) in which redexes in the same family are actually represented by the same structure, and which only ever contracts redexes which will contribute to the final result of the computation.
>
> For the second, a generalisation to the higher-order case of the theory of *sequentiality* and *neededness* as founded in Huet & Lévy (1991) for orthogonal TRSs, is required.

We propose a candidate for the higher-order generalisation of sequentiality and neededness, as well as an appropriate evaluator for family reductions in orthogonal higher-order rewriting systems, which is optimal for sequential HRSs as described by Van Oostrom. Moreover, we provide an effective algorithm for such an evaluator, for those systems that are 'match-sequential' (§5.4).

1.3 Substructural rewriting

Our approach here is to define *Linear HRSs* (§5.1) — higher-order term rewriting systems with Intuitionistic Linear Logic (Benton et al. 1992) as a substitution calculus, in the sense of Van Oostrom (1994). These systems allow us to explicitly manage the contraction and weakening (and promotion and dereliction) of subterms and variables within rules, which plays well with the known relationship between optimal sharing and linear logic. Linear HRSs are expected to be of broader general use, but here we apply them only to the problem of optimal reduction.

Lamping's (1990) major innovation was to add sharing operators not to a directed acyclic graph, as Wadsworth (1971) had done, but to Bourbaki graphs, which encode second-order terms with bound variables that arc back to their respective binders. We generalise this technique from second-order terms to higher-order abstract syntax, providing translations from linear λ-terms of any order to Lamping–Gonthier sharing graphs. This translation is not at the level of a link signature (mapping syntactic abstractions and applications in the term to links, or nodes, in the graph), but rather at the level of a substitution calculus, mapping HRSs over a linear λ-calculus to HRSs over Lamping–Gonthier sharing graphs. This means it is constants within the term that are mapped to a link, as opposed to elements of the λ-calculus syntax.

We provide a translation of match-sequential Linear HRSs to Interaction Nets (Lafont 1990), as well as a simple generalisation of Interaction Net rewriting to match-sequential systems, wherein sequential contexts are used in place of the 'agents' of Interaction Nets, and their sequential indices the 'principal ports' at which interactions may take place. Lastly, we propose various generalisations of this approach to similar term rewriting systems and substitution calculi.

1.4 Thesis structure

This thesis comprises the following chapters. The dependencies for these chapters are essentially linear: each will only really make sense in the context of those previous.

1. **Introduction** — This very introduction; a general overview of our work.
2. **Preliminaries** — Covers the terminology and notation used for λ-calculus, term rewriting, and intuitionistic linear logic. For more thorough an explanation of this material, the reader should look to the sources cited.
3. **Term graphs** — Introduces 'higher-order term graphs' as a graphical representation of higher-order abstract syntax, generalising both first-order term trees and second-order 'Bourbaki graphs'.
4. **Sharing graphs** — Augments the previous term graphs with sharing operators, thus generalising Lamping–Gonthier sharing beyond second order.
5. **Rewriting** — Describes several methods of defining graph rewriting systems modulo Guerrini's π-interaction. Two of these, restricted to match-sequential higher-order rewriting systems without rigid variable occurrences on the left-hand side, are effectively computable.
6. **Extensions** — Describes ways in which the previous effective rewriting techniques may be extended to cover more general systems — additive types, polymorphic types, other substructural logics, fixed points, etc.
7. **Conclusion** — An overall summary of our work, its relation to prior work, and potential future work.

Furthermore, there are two appendices.

A. **Equivalence proofs** — Figures for term graph equivalence proofs that would take up too much space had they been included in the main text.

B. **Krivine machines** — Related but tangential work on implementing strong β-reduction by combining abstract machines with explicit substitution.

1.4.1 Contributions

Our primary contributions are the following:

- §3.3 — A generalisation of first-order term trees and second-order Bourbaki graphs to the case of higher-order abstract syntax. Also, a translation from purely-linear λ-terms to these graphs.
- §4.3 — A generalisation of Lamping–Gonthier sharing from second-order Bourbaki graphs to higher-order abstract syntax. A translation from modal linear λ-terms to these graphs, which together with the previous forms a complete translation of linear λ-calculus to sharing graphs.
- §5 — Linear HRSs: higher-order rewriting systems having linear λ-calculus as a substitution calculus. A translation of these term rewriting systems to Lamping–Gonthier sharing graphs, resulting in a Lévy-optimal evaluator.
- §6 — Various extensions of the previous higher-order rewriting systems to different type systems, term constructs, pattern forms, and so on.

1.4.2 Other publications

In addition to this thesis and its contributions thus detailed, the author has also during this research produced a number of other papers in the field, namely:

- *Abstract machines for higher-order term sharing* (Smith 2014) — On strong abstract machines for the λ-calculus, forming the basis of Appendix B, as well as an earlier exploration into higher-order term sharing. This was presented at, and published in the pre-proceedings of, the IFL conference.
- *Normal forms and infinity* (Kahrs & Smith 2014) — Describes a technique for deriving term rewriting systems through signature embedding (cf. §5.3), in this case for constructing normal forms from infinitary reduction sequences.
- *Non-ω-overlapping TRSs are UN* (Kahrs & Smith 2016) — Proves that first-order term rewriting systems that are non-ω-overlapping (i.e. do not overlap with respect to unification by rationally infinite terms) have unique normal forms, closing a 27-year-old open problem.

Chapter 2

Preliminaries

2.1 Term Rewriting Systems

The definitions in this section are standard and are largely taken from Terese (2003). Certain choices of definition, notation, and terminology, are however different, drawing from alternative conventions where we find them preferable.

Definition 1. An *Abstract Rewriting System* (ARS) is a binary relation on a set of *objects*. We use arrows to denote ARSs, which allows for some special rewriting notation: if $\to$ is ARS then $\leftarrow = \to^{-1}$ and $\twoheadrightarrow = \to^*$.

Concatenating rewrite steps, we have (possibly infinite) *reduction sequences*, $t_0 \to t_1 \to t_2 \to \cdots$, or *reductions* for short. If $t_0 \to \cdots \to t_n$ $(n \geq 0)$ we also write $t_0 \twoheadrightarrow t_n$, and t_n is called a *reduct* of t_0. If an object cannot be reduced, it is called a *normal form*; we may write $t_0 \downarrow t_n$ if t_n is a normal form.

Terms

Definition 2. A *term signature* Σ consists of a non-empty set of *operator symbols* $F, G, \ldots$, each equipped with a fixed *arity*. The arity of a operator symbol F is a natural number, indicating the number of arguments it is supposed to have. Thus we may have nullary, unary, binary, ternary, etc., operator symbols. We write $\#F$ to denote the arity of operator symbol F.

Terms are expressions constructed from symbols in an *alphabet*, consisting of the signature and a countably infinite set *Var* of *variables*. The set of variables is

assumed to be disjoint from the operator symbols in the signature Σ. Variables will be denoted by x,y,z, etc., if needed with indices: x_0, x_1, x_2, x_3, ...

Definition 3. The set of *terms* over Σ is indicated as $Ter(\Sigma)$ and is defined inductively:

1. $x \in Ter(\Sigma)$ for every $x \in Var$.

2. If F is an n-ary operator symbol and $t_1, \ldots, t_n \in Ter(\Sigma)$, then $F(t_1, \ldots, t_n) \in Ter(\Sigma)$. If an operator symbol is nullary then we may write A instead of $A()$.

The terms t_i are called the *arguments* of the term $F(t_1, \ldots, t_n)$, and the symbol F the *root symbol*, written $F \equiv root(t)$.

Terms not containing a variable are called *closed* terms or *ground* terms, and $Ter_0(\Sigma)$ is the set of closed terms. Terms in which no variable occurs more than once are called *linear*.

By $Var(t)$ we denote the set of variables that occur in t. So t is a closed term if $Var(t) = \emptyset$. If V is a set of variables, then the notation $Ter(\Sigma, V)$ is often used to denote the set of terms t with $Var(t) \subseteq V$. According to this notation we have $Ter_0(\Sigma) = Ter(\Sigma, \emptyset)$.

The *length* of the term t, denoted by $|t|$, is defined as the number of occurrences of operator symbols and variables in t. So we have the inductive definition $|x| = 1$ for variable x, and $|F(t_1, \ldots, t_n)| = |t_1| + \cdots + |t_n| + 1$, for an operator symbol F with arity n.

Identity of terms is called *syntactic identity* and is denoted by $\equiv$.

Contexts

A context can be viewed as an incomplete term, which may contain several empty places, or *holes*.

Formally a context can be defined as a term containing zero, one, or more occurrences of a special nullary operator symbol $\Box$, i.e. a term over the extended signature $\Sigma \cup \{\Box\}$. If C is a context containing exactly n holes, and $t_1, \ldots, t_n$ are terms, then $C[t_1, \ldots, t_n]$ denotes the result of replacing the holes of C from left to right with $t_1, \ldots, t_n$. If $t \in Ter(\Sigma)$ can be written as $t \equiv C[t_1, \ldots, t_n]$, then the

context C will also be called a *prefix* of t. If $t \equiv D[C[t_1, \ldots, t_n]]$ for some prefix D, then C is a *slice* of t.

An important special case is when there is exactly one occurrence of $\Box$ in C. Then C is called a *one-hole context*, also denoted by $C[]$. If $t \in \mathit{Ter}(\Sigma)$ can be written as $t \equiv C[s]$, then the term s is said to be a *subterm* of t, written $s \subseteq t$. Since $\Box$ is itself a context, the *trivial context*, we also have $t \subseteq t$. Other subterms s of t than t itself are called *proper* subterms of t, written $s \subset t$.

Since contexts have been introduced as terms over an extended signature, they automatically inherit the inductive definition of terms.

The same term s may occur more than once as a subterm in a term t. It is often important to distinguish between different occurrences of a subterm (or symbol) in a term. The ordered pair $(s, C[])$ uniquely determines the *occurrence* of s in $C[s]$ with prefix $C[]$. Analogously, the pair $(F, C[])$ determines a unique occurrence of the operator symbol F in the term $C[F(t_1, \ldots, t_n)]$, namely the root symbol of the occurrence $(F(t_1, \ldots, t_n), C[])$. We may identify occurrences within a term by underlining.

For the relative position of two occurrences in a given term t there are two possibilities. Either one occurrence *contains* the other, or the two occurrences are completely separate. In the latter case we call them *disjoint*.

Definition 4. Consider two occurrences $(s, C[])$ and $(s', C'[])$ in a term t.

1. We say that $(s', C'[])$ is *contained* in $(s, C[])$, written $(s', C'[]) \leq (s, C[])$, if for some context $D[]$ we have $C'[] \equiv C[D[]]$. $(s', C'[])$ is called the *inner* occurrence, $(s, C[])$ the *outer* occurrence.

2. We say that $(s, C[])$ and $(s', C'[])$ are *disjoint* if for some two-hole context E either $C[] \equiv E[\Box, s']$ and $C'[] \equiv E[s, \Box]$, or, in the reverse order, $C[] \equiv E[s', \Box]$ and $C'[] \equiv E[\Box, s]$.

Rewriting

Definition 5. A *substitution* is a map $\sigma : \mathit{Var} \to \mathit{Ter}(\Sigma)$. A substitution may be lifted to a map $\hat{\sigma} : \mathit{Ter}(\Sigma) \to \mathit{Ter}(\Sigma)$ by

$$\hat{\sigma}(x) \equiv \sigma(x)$$
$$\hat{\sigma}(F(t_1, \ldots, t_n)) \equiv F(\hat{\sigma}(t_1), \ldots, \hat{\sigma}(t_n))$$

A substitution σ that replaces distinct variables with distinct variables (i.e. σ is injective and $\hat{\sigma}(x)$ is a variable for every x) is called a *renaming*. If $\sigma|_{\mathit{Var}(t)}$ is a renaming, then σ is called a *renaming for* t.

Definition 6. A *rewrite rule* for a signature Σ is a pair (l, r) of terms of $\mathit{Ter}(\Sigma)$. It will be written as $l \to r$. Often a rewrite rule will be given a name, e.g. ρ, and we write $\rho : l \to r$. Two restrictions on rewrite rules are imposed:

1. the left-hand side l is not a variable,
2. every variable occurring in the right-hand side r occurs in l as well.

An *instance* of ρ is obtained by applying a substitution σ. The result is an *atomic* rewrite step $\hat{\sigma}(l) \to_\rho \hat{\sigma}(r)$. The left-hand side $\hat{\sigma}(l)$ is called a *(ρ-)redex*. The right-hand side $\hat{\sigma}(r)$ is called its *contractum*.

The words redex and contractum will also be used for occurrences of a redex and a contractum. Replacing a redex $\hat{\sigma}(l)$ with its contractum is called *contraction*.

Given a term, it may contain one or more occurrences of redexes. A rewrite step consists of contracting one of these, i.e. replacing the redex with its contractum.

Definition 7. A *rewrite step* according to the rewrite rule $\rho : l \to r$ consists of contracting a redex within an arbitrary context:

$$C[\hat{\sigma}(l)] \to_\rho C[\hat{\sigma}(r)]$$

We call $\to_\rho$ the *rewrite relation* generated by ρ.

Definition 8. A *Term Rewriting System* (TRS) is a pair $T = (\Sigma, R)$ of a signature Σ and a set of rewrite rules R for Σ.

The *rewrite* relation of T, denoted by $\to$ (or $\to_R$), is defined as the union $\bigcup\{\to_\rho \mid \rho \in R\}$. So we have $t \to_R s$ when $t \to_\rho s$ for one of the rewrite rules $\rho \in R$.

If $T = (\Sigma, R)$, then we may also write $\mathit{Ter}(T)$ for $\mathit{Ter}(\Sigma)$, and likewise $\mathit{Ter}_0(T)$. We may specify a TRS just by its set of rewrite rules R, in which case the signature Σ is understood to consist solely of the operator symbols used in R.

2.1.1 Interaction Systems

Definition 9. An Interaction System (IS) *term signature* Σ consists of a non-empty set of operator symbols $F, G, \ldots$, each equipped with a fixed *arity*. The arity of an operator symbol F is a finite (possibly empty) sequence of natural numbers $a \in \mathbb{N}^*$. We write $\#F$ to denote the arity of operator symbol F.

In contrast to first-order term rewriting systems, the arity indicates not only how many arguments an operator symbol is supposed to have, but also how many variables are bound over each argument. A TRS operator symbol of arity n would translate into an IS operator symbol of arity $(0, 0, \ldots, 0)$, with n zeroes.

The term signature is partitioned into two disjoint sets Σ^+ and Σ^-, representing *constructors* (denoted by C) and *destructors* (denoted by D), respectively. Destructors have the added constraint that they must be of some arity $(0, k_2, \ldots, k_n)$ where $n \geq 1$. That is, they must have at least one first argument, which cannot have a variable binder.

Definition 10. The set of *IS terms* over Σ is indicated as $\mathit{Ter}(\Sigma)$ and is defined inductively:

1. $x \in \mathit{Ter}(\Sigma)$ for every $x \in \mathit{Var}$.

2. If F is an operator symbol of arity $k_1, \ldots, k_n$, and $t_1, \ldots, t_n \in \mathit{Ter}(\Sigma)$, then $F(\vec{x}^1_{k_1}.t_1, \ldots, \vec{x}^n_{k_n}.t_n) \in \mathit{Ter}(\Sigma)$, where '$\vec{x}_k.t$' is syntactic shorthand for '$(x_1, \ldots, x_k).t$', an abstraction binding k variables over t. If an operator symbol is nullary, i.e. $n = 0$, then we may write A instead of $A()$.

Definition 11. Given an arbitrary but fixed set of *meta-variables* X, an *IS rewrite rule* for a signature Σ is a pair (H_l, H_r) of *meta-expressions*, written as $H_l \to H_r$.

The left-hand meta-expression H_l then has the following format:

$$D(C(\vec{x}^1_{k_1}.X_1, \ldots, \vec{x}^m_{k_m}.X_m), \ldots, \vec{x}^m_{k_n}.X_n)$$

where $D \in \Sigma^+$, $C \in \Sigma^-$, $\#D = (0, k_{m+1}, \ldots, k_n)$, $\#C = (k_1, \ldots, k_m)$, and $i \neq j$ implies $X_i \neq X_j$ (i.e. left linearity).

The right-hand meta-expression H_r is a *closed* meta-expression whose meta-variables are already in the left-hand side of the rule, built up according to the following syntax:

$$H ::= x \mid F(\vec{x}^1_{a_1}.H_1, \ldots, \vec{x}^j_{a_j}.H_j) \mid X_i[H_1/x^i_1, \ldots, H_{k_1}/x^i_{k_i}]$$

Finally, the set of rewrite rules must be *non-ambiguous*, i.e. there exists at most one rewrite rule for each pair D–C.

Example 1. The following Interaction System defines β-reduction on the untyped λ-calculus:

$$\mathrm{App}(\mathrm{Abs}(x.M), N) \to M[N/x]$$

The symbols in this system's signature are App : $(0, 0)$ and Abs : (1); M and N are meta-variables.

2.1.2 Sequentiality

We generalise the definition of sequentiality slightly to arbitrary monotonic functions on contexts, rather than only predicates. This allows us to formalise a definition of match sequentiality that we feel is a little smoother than the standard one (Thatte 1987) for systems that are not orthogonal; for orthogonal TRSs the two definitions coincide. We otherwise follow the presentation of Huet & Lévy (1991) and Terese (2003).

Definition 12. Let f be a monotonic function on contexts (on $\leq$). Position p in a context C is an *index* of f in C iff $C|_p$ is a hole and $\forall D$ such that $C \leq D$ then $f(C) < f(D)$ implies that $D|_p$ is *not* a hole. f is said to be *sequential* iff whenever $\exists D$ such that $C \leq D$ then $f(C) < f(D)$ implies there exists an index of f in C.

Definition 13. Given a TRS R, let *matches*(C) equal the number of distinct rules in R for which C is a redex. A TRS is *match sequential* iff *matches* is sequential.

Example 2. In the following non-left-linear, overlapping TRS:

$$F(A, x, y) \to r_1$$
$$F(x, x, y) \to r_2$$

The context $C = F(A, \Box_1, \Box_2)$ matches only the first rule, so matches$(C) = 1$. C does not match the second rule because $A \neq \Box_1$. In contrast, *matches*$(C[A, \Box]) = 2$, because $F(A, A, \Box)$ also matches the second rule, with $x = A$ and $y = \Box$.

For orthogonal TRSs match sequentiality also coincides with strong sequentiality (Huet & Lévy 1991). Additionally, these have a decidable and normalising *index reduction* strategy, whereby each index is reduced until they form a match, or they reach head normal form and do not match.

Thatte (1987) instead uses a monotonic predicate '*isredex*' in the definition of match-sequential TRSs. Let *isredex*(C) equal whether or not C matches any rule in R, where Booleans have the ordering false $\leq$ true.

Theorem 1. *The monotonic functions* matches *and* isredex *coincide in the case of orthogonal TRSs.*

Proof. The maximum number of rules a context can possibly match at the root, given an orthogonal TRS R, is 1 (Terese 2003). Thus the two functions are isomorphic given false $= 0$ and true $= 1$. □

Definition 14. The *prefix* function $\omega(\cdot)$ maps a pattern to the context with holes substituted for its free variables:

$$\omega(x) \equiv \Box$$
$$\omega(F(t_1, \ldots, t_n)) \equiv F(\omega(t_1), \ldots, \omega(t_n))$$

With our definition of match sequentiality, if R has two rules (l_1, r_1) and (l_2, r_2) where $\omega(l_1) \leq \omega(l_2)$, the predicate *matches*(C, l_i) to determine whether C matches a specific rule $(l_i, r_i) \in R$ must still also be sequential. In contrast, with Thatte's

definition, the second rule can be ignored when determining whether the whole TRS is match-sequential, since $\text{isredex}(\omega(l_1)) \not\leftarrow \text{isredex}(\omega(l_2))$ — both are simply true — and thus there need not be an index in $\omega(l_1)$. Our match sequentiality does not allow this 'obscuring' of non-sequentiality, so that if $R' \subseteq R$ where R is match sequential then R' is also match sequential.

Example 3. 'Gustave's TRS' (Huet & Lévy 1991) is *not* match-sequential, since it is not known which subterm of F to check first; any that we check may not form a part of the redex, and yet the term may still be a redex. We write r_i to stand for an arbitrary (but valid) right-hand side.

$$F(A, B, x) \to r_1$$
$$F(x, A, B) \to r_2$$
$$F(B, x, A) \to r_3$$

2.2 Higher-order Rewrite Systems

Structured Rewriting Systems (Van Oostrom 1996), previously called Higher-Order Rewriting Systems (Van Oostrom 1994), make an explicit division between the systems' combinatorial component (rules) and structural component (substitutions). The latter component is called the rewrite sytem's *substitution calculus*, and is usually set in place before specific rewrite rules are laid on top. Structured rewriting can then be viewed as 'rules modulo substitution'.

Definition 15. A *substitution calculus* $\mathcal{SC}$ consists of:

- A set $\rho, \sigma, \tau, \upsilon, \ldots \in \mathcal{T}$ of *types*.
- A set $s, t, u, \ldots$ of *prestructures* inhabiting types in $\mathcal{T}$. We assume an alphabet of atomic prestructures: *variables* $x, y, z, \ldots$ for each type, and a *signature* Σ of *operator symbols* or *constants* $F, G, H, \ldots$
- An ARS $\to_{\mathcal{SC}}$ on prestructures.

A *structure* is a prestructure in $\to_{\mathcal{SC}}$-normal form. A rewrite rule $\aleph$ is a triple (l, r, τ), often written $\vdash l \to r : \tau$, of *closed* structures (i.e. containing no free

variables) l and r of the same type τ. The ARS $\to_{\aleph}$ on structures is defined by $s \to_{\aleph} t$, if $s \leftrightarrow^* C[l]$ and $C[r] \leftrightarrow^* t$, where C is an $\mathcal{SC}$-normal unary context. The rewrite step is said to *contract* the $(\aleph)$-redex $(C, \aleph)$.

Definition 16. A *Structured Rewriting System* (SRS) $\mathcal{H}$ is a triple $(\mathcal{SC}, \mathcal{A}, \mathcal{R})$ consisting of a substitution calculus $\mathcal{SC}$, an alphabet $\mathcal{A}$, and a set $\mathcal{R}$ of rewrite rules. The ARS associated with $\mathcal{H}$ is a relation $\mathbb{PT}$ defined by $\to_{\mathcal{H}} = \bigcup_{\aleph \in \mathcal{R}} \to_{\aleph}$.

We define Structured Rewriting Systems having the simply-typed λ-calculus as a substitution calculus. In their more general form, these systems are Wolfram's (1990) Higher-Order Term Rewriting Systems (HOTRSs), but can be restricted to admit only higher-order patterns (Miller 1991) on the left-hand side, which are then Nipkow's (1991) Higher-order Rewrite Systems (HRSs).

HRSs were limited to patterns in this way at least in part for worry that higher-order matching may in general be undecidable; Miller had proved the decidability of these higher-order patterns. It has since been shown that $\beta\eta$-matching is decidable in general for the simply-typed λ-calculus with one base type (Stirling 2009) — although not β-matching without η-extensionality (Loader 2003), which serves to demonstrate how fragile this property is. It is still not yet known whether $\beta\eta$-matching is decidable for the simply-typed λ-calculus with more than one base type. In any case, we choose to play it safe and, without need for more general HRSs, follow Nipkow in restricting their left-hand sides to Miller's higher-order patterns.

Definition 17. Given a set A of *type atoms*, the set T of *simple types* is defined:

$$\alpha \in A \implies \alpha \in T$$
$$\sigma, \tau \in T \implies \sigma \to \tau \in T$$

Definition 18. The set Λ of *simply-typed λ-preterms* is defined inductively as some t such that $\Gamma \vdash t : \tau$, for some *basis* set Γ of variables, can be derived by the following rules:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{\Gamma \vdash t_1 : \sigma \to \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1 t_2 : \tau} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x^{\sigma}.t : \sigma \to \tau}$$

The rewrite relation $\rightarrow_{\mathcal{SC}}$ on λ-preterms modulo the usual α-renaming of bound variables (Barendregt 1984) is generated by the following rules:

$$(\lambda x.t)u \rightarrow_\beta t[u/x]$$
$$t \rightarrow_{\bar{\eta}} \lambda x.tx$$

where $\bar{\eta}$ is not allowed to create β-steps and x does not occur in t.

Definition 19. A *Higher-Order Term Rewriting System* (Wolfram 1990) is a structured rewriting system having simply-typed λ-calculus as a subsitution calculus.

A higher-order pattern (Miller 1991) is a simply-typed term of the form $\lambda\vec{x}.t$ such that:

1. t is of the form $Ft'_1 \ldots t'_n$, where F is called the *head* of the term.
2. Each occurrence of a *parameter* x_i in $\vec{x}$ in t only has (η-normal forms of) pairwise distinct variables not in $\vec{x}$ as arguments.

A *Higher-order Rewrite System* $\mathcal{H}$ is an HOTRS in which all rules' left-hand sides are higher-order patterns.

Definition 20. An HRS is *orthogonal* if it is:

1. *Left-linear*, i.e. each parameter in a pattern on the left-hand side of a rule occurs only once.
2. *Non-overlapping*, i.e. the left-hand sides of a rule cannot be unified with that of another rule.

In this thesis we only concern ourselves with orthogonal HRSs.

2.2.1 Optimality

Here we use Van Oostrom's (1996) definitions of *redex families* and *labellings* of a Higher-order Rewrite System, generalising those of Lévy (1980) for the λ-calculus. Intuitively, a redex family is a set of redexes that are created 'in the same way'. Families specify which redexes should be shared in an optimal evaluator.

Definition 21. Given an set L of *label atoms*, the set L_λ of λ-*labels* is defined inductively:

$$\alpha \in L \implies \alpha \in L_\lambda$$
$$\ell_1, \ell_2 \in L_\lambda \implies \ell_1 ; \ell_2 \in L_\lambda$$

The *labelled* λ-calculus Λ^ℓ is obtained from the unlabelled λ-calculus Λ by labelling elements (nodes and edges) of terms' (Bourbaki) graph representation (Bourbaki 1954, cf. §3.2.2).

Definition 22. Steps in a labelled HRS are either labelled β-steps, as defined above, or labelled partial HRS-steps.

The set L of *rule-labels* is defined inductively:

$$n \in \mathbb{N} \implies n \in L_{\mathcal{H}}$$
$$\vec{\alpha} \subseteq L_\lambda \wedge \vdash l, r : \tau \wedge n \in \mathbb{N} \implies (\vec{\alpha}, l \to r, n) \in L_{\mathcal{H}}$$

For the construction of the labelled partial HRS-step, first perform the labelled β-expansion as defined above (this only depends on the left-hand side). Then perform the (unlabelled) HRS replacement step and after that label all created (either by the β-expansion or by the replacement) elements by $(\vec{\alpha}, l \to r, n)$ where α is an enumeration (fixed by l) of the labels in the left-hand side, and n is a sequence number in an enumeration of the elements (fixed by r and the β-expansion). Finally, the edges connecting the created symbols to its surroundings are provided with a sequence number (establishing the interface).

Definition 23. In an *initial labelling* of a preterm, all elements of the labelled preterm have distinct atomic labels.

Definition 24. A *locked* sequence is a structure $\sigma \lhd \partial$, where $\sigma : s \twoheadrightarrow t$ is a sequence (on preterms) and ∂ a part of t, meaning that ∂ is a pair consisting of a position p and a λ-term u, such that $t = C[\hat{\theta}(u)]_p$ for some context C and substitution θ. The sequence is said to be *locked onto* u (at p).

Definition 25. A *family relation* $\simeq$ is an equivalence relation on coinitial rewrite sequences locked onto the same lock (i.e. their λ-terms are identical).

Let $\sigma \lhd \partial$, $\tau \lhd \partial$ be two locked rewrite sequences. They are in the same redex family, $\sigma \lhd \partial \simeq \tau \lhd \partial$, if their initial labellings are locked onto the same labelled part.

Definition 26. A *family reduction* is a $\multimap\!\!\!\!\rightarrow$-rewrite in which in each step a non-empty *family* of redexes is contracted. A family reduction ρ to normal form is *optimal* if $|\rho| \leq |\sigma|$ for all $\multimap\!\!\!\!\rightarrow$-rewrites σ starting and ending in the same preterms as ρ. A rewrite strategy is optimal if it constructs optimal family reductions.

2.3 Intuitionistic Linear Logic

Intuitionistic Linear Logic is a refinement of Intuitionistic Logic in which formulae must be used exactly once. Girard's (1987) linear logic removes from intuitionistic logic the structural rules of *weakening* and *contraction*:

$$\frac{\Gamma \vdash \psi}{\Gamma, \phi \vdash \psi}\ \text{Weakening} \qquad \frac{\Gamma, \phi, \phi \vdash \psi}{\Gamma, \phi \vdash \psi}\ \text{Contraction}$$

To regain the expressive power of Intuitionistic Logic, these rules are then returned, but in a controlled manner. A logical *exponentiation* operator '!' ("of course") is introduced, which allows a formula to be used as many times as required (including zero). Following Girard's original presentation of linear logic by way of sequent calculus, '!' is introduced by the rules,

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma, !\phi \vdash \psi}\ !_L \qquad \frac{!\Gamma \vdash \psi}{!\Gamma \vdash !\psi}\ !_R$$

where $!\Gamma$ means that all formulae in multiset Γ are of the form $!\psi_i$.

The structural rules are then permitted only on formulae of the form $!\phi$:

$$\frac{\Gamma \vdash \psi}{\Gamma, !\phi \vdash \psi}\ \text{Weakening} \qquad \frac{\Gamma, !\phi, !\phi \vdash \psi}{\Gamma, !\phi \vdash \psi}\ \text{Contraction}$$

Throughout this thesis, we shall consider only the *multiplicative* fragment of Intuitionistic Linear Logic (i.e. including not only linear implication '$\multimap$' and exponentiation '!', but also multiplicative conjunction '$\otimes$'), as shown in Figure 2.

$$\frac{}{\psi \vdash \psi}\ \text{Identity} \qquad \frac{\Gamma, \phi, \varphi \vdash \psi}{\Gamma, \varphi, \phi \vdash \psi}\ \text{Exchange}$$

$$\frac{\Gamma \vdash \phi \quad \Delta, \phi \vdash \psi}{\Gamma, \Delta \vdash \psi}\ \text{Cut} \qquad \frac{\Gamma \vdash \phi \quad \Delta, \varphi \vdash \psi}{\Gamma, \Delta, \phi \multimap \varphi \vdash \psi}\ \multimap_L \qquad \frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \multimap \psi}\ \multimap_R$$

$$\frac{\Gamma, \phi, \varphi \vdash \psi}{\Gamma, \phi \otimes \varphi \vdash \psi}\ \otimes_L \qquad \frac{\Gamma \vdash \phi \quad \Delta \vdash \varphi}{\Gamma, \Delta \vdash \phi \otimes \varphi}\ \otimes_R \qquad \frac{\Gamma \vdash \psi}{\Gamma, I \vdash \psi}\ I_L \qquad \frac{}{\vdash I}\ I_R$$

$$\frac{\Gamma, !\phi \vdash \psi}{\Gamma, \phi \vdash \psi}\ \text{Dereliction} \qquad \frac{!\Gamma \vdash \psi}{!\Gamma \vdash !\psi}\ \text{Promotion}$$

$$\frac{\Gamma \vdash \psi}{\Gamma, !\phi \vdash \psi}\ \text{Weakening} \qquad \frac{\Gamma, !\phi, !\phi \vdash \psi}{\Gamma, !\phi \vdash \psi}\ \text{Contraction}$$

Figure 2: (Multiplicative) Intuitionistic Linear Logic

This is quite standard in the literature, as although these term calculi are able to model the additive fragment of Intuitionistic Linear Logic (i.e. including additive conjunction '&' and possibly disjunction '$\oplus$'), doing so ends up with rather a lot of cases to handle, and so they are elided for brevity. A (rarer) example of a term calculus which does explicitly feature an additive fragment is that of Wadler (1993). We consider the additive fragment as something of an aside (§6.1.1), but otherwise focus on the multiplicative, as additive types cause certain complications during the translation to Lamping–Gonthier graphs, which might obscure the simplicity of the principal algorithm.

2.3.1 Linear λ-calculus

Abramsky (1993) introduced a variant of the λ-calculus having the intuitionistic fragment of linear logic as a type system, much as the simply-typed λ-calculus of Church (1940) has the 'minimal logic' of Johansson (1936) as its type system. Unfortunately, Abramsky's calculus was found to be incoherent with the *de facto* categorical semantics of Seely (1989), as substitution (i.e. Gentzen's *Hauptsatz*, or cut elimination) did not commute with promotion (Wadler 1991). This allowed for there be more than one derivation of a single syntactic term, which were yet not semantically equivalent.

This realisation led to a number of calculi intending to amend the problem, with the most prominent being those of Benton et al. (1992) and Barber & Plotkin (1997). In the former case, an explicit closure is formed at the promotion site, which is then converted into a substitution when the promotion is eliminated. In the latter, free variables are stored in two separate 'dual' type contexts (resulting in the name 'DILL', *Dual Intuitionistic Linear Logic*): one for linear variables and one for intuitionistic, with variables being moved from the former to the latter by way of a *let* expression. Both methods make use of a 'thunk' in order to protect the promotion from substitutions; the latter is essentially a generalisation of the former wherein the thunks may be distributed throughout the term rather than being required to close the promotion site.

The term calculus used here is that of Benton et al. (1992). Since this calculus was first formalised, other, less syntactically explicit term calculi have generally seen more use. One such term calculus is Dual Intuitionistic Linear Logic, or 'DILL' (Barber & Plotkin 1997), which performs promotion explicitly, and uses a kind of 'pattern matching' to perform dereliction, but otherwise allows the structural rules of weakening and contraction to occur implicitly without syntactic consequence. While such a calculus may well be favourable in many circumstances, we find the explicit syntax of Benton et al. to be particularly suited to our usage. In this term calculus, each subexpression corresponds directly to an operation in the corresponding logic, which allows us to map a inductive definition of terms directly to an inductive definition of term graphs. A more implicit syntax being used would obstruct and complicate this approach, in fact obscuring clarity.

It is worth emphasising that despite their syntactic differences, these two calculi, 'ILL' and 'DILL', are in fact equivalent, as they each express the semantics of Seely (1989). Thus, in a practical implementation of this system, it is entirely likely that the chosen syntax would be that of DILL, or a term calculus at least in the same vein, as most modern developments have tended to be. Indeed, our earlier attempts at both formalising and implementing our algorithm used the 'DILL' syntax. Nevertheless, for the purposes of this thesis, such implicit syntax proved more of a hindrance than a help.

Definition 27. Given a set A of *type atoms*, we define the set $Typ(A)$ of *linear types on* A inductively:

$$\begin{aligned} \alpha \in A &\implies \alpha \in Typ(A) \\ \sigma, \tau \in Typ(A) &\implies \sigma \multimap \tau \in Typ(A) \\ \sigma, \tau \in Typ(A) &\implies \sigma \otimes \tau \in Typ(A) \\ \tau \in Typ(A) &\implies !\tau \in Typ(A) \\ &\quad I \in Typ(A) \end{aligned}$$

We may also write Typ to mean $Typ(\{0\})$, and will generally assume this to be the case.

The rules for constructing linear λ-terms are defined in Figure 3. To this we add a rule for variable substitution, or 'cut', in Figure 4. Substitution is not a part of the syntax, but of the metasyntax, the propagation of which is defined in Figure 5. We assume here the so-called 'Barendregt convention' (Barendregt 1984), whereby α-renaming is performed so as to avoid any illegal variable capture.

A linear λ-term may be β-reduced by the rules given in Figure 6. Note that these rules are termed β-reduction even though they include additional rules to the traditional β-reduction $(\lambda x.t)u \to t[u/x]$ of intuitionistic λ-calculi. To these we add the commutation rules in Figure 7, which although not 'contractive' in the sense of the β-rules, are still required in order to reach a normal form. Finally, the rules for term equality, whereby $t_1 \leftrightarrow^* t_2$ implies $t_1 = t_2$, are given in Figure 8. All of these relations are as defined by Benton et al. (1992).

A term in the simply-typed λ-calculus may be translated into ILL in a number of ways, the most common of which are the *standard* translation and *steadfast* translation, both due to Girard (1987). The simply-typed λ-calculus and the term calculus of Intuitionistic Linear Logic are computationally equivalent, although the addition of multiplicative and unit types result in a more expressive calculus. The latter also, of course, has a more sophisticated type system for expressing the substructural properties of certain variables, although the standard and steadfast translations do not take advantage of this in any respect.

$$\frac{}{x : \tau \vdash x : \tau} \qquad \frac{}{\vdash F : \tau}\ F : \tau \in \Sigma$$

$$\frac{\Gamma \vdash t : \sigma \multimap \tau \quad \Delta \vdash u : \sigma}{\Gamma, \Delta \vdash tu : \tau} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x^{\sigma}.t : \sigma \multimap \tau}$$

$$\frac{\Gamma \vdash s : \sigma \quad \Delta \vdash t : \tau}{\Gamma, \Delta \vdash s \otimes t : \sigma \otimes \tau} \qquad \frac{\Gamma \vdash s : \rho \otimes \sigma \quad \Delta, x : \rho, y : \sigma \vdash t : \tau}{\Gamma, \Delta \vdash \text{let } s \text{ be } x \otimes y \text{ in } t : \tau}$$

$$\frac{}{\vdash * : I} \qquad \frac{\Gamma \vdash s : I \quad \Delta \vdash t : \tau}{\Gamma, \Delta \vdash \text{let } s \text{ be } * \text{ in } t : \tau}$$

$$\frac{\Delta_1 \vdash s_1 : !\sigma_1 \quad \ldots \quad \Delta_n \vdash s_n : !\sigma_n \quad x_1 : !\sigma_1, \ldots, x_n : !\sigma_n \vdash t : \tau}{\Delta_1, \ldots, \Delta_n \vdash \text{promote } s_1, \ldots, s_n \text{ for } x_1, \ldots, x_n \text{ in } t : !\tau}$$

$$\frac{\Gamma \vdash t : !\tau}{\Gamma \vdash \text{derelict } t : \tau} \qquad \frac{\Gamma \vdash s : !\sigma \quad \Delta \vdash t : \tau}{\Gamma, \Delta \vdash \text{discard } s \text{ in } t : \tau}$$

$$\frac{\Gamma \vdash s : !\sigma \quad \Delta, x : !\sigma, y : !\sigma \vdash t : \tau}{\Gamma, \Delta \vdash \text{copy } s \text{ as } x, y \text{ in } t : \tau}$$

Figure 3: Linear λ-terms

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

Figure 4: Variable substitution rule

$$
\begin{aligned}
x[u/x] &\equiv u \\
(\lambda y.t)[u/x] &\equiv \lambda y.t[u/x] \quad \text{if } y \notin \mathit{Var}(u) \\
(t_1 t_2)[u/x] &\equiv \begin{cases} (t_1[u/x])t_2 & \text{if } x \in \mathit{Var}(t_1) \\ t_1(t_2[u/x]) & \text{if } x \in \mathit{Var}(t_2) \end{cases} \\
(t_1 \otimes t_2)[u/x] &\equiv \begin{cases} t_1[u/x] \otimes t_2 & \text{if } x \in \mathit{Var}(t_1) \\ t_1 \otimes t_2[u/x] & \text{if } x \in \mathit{Var}(t_2) \end{cases} \\
(\text{let } s \text{ be } y \otimes z \text{ in } t)[u/x] &\equiv \begin{cases} \text{let } s[u/x] \text{ be } y \otimes z \text{ in } t & \text{if } x \in \mathit{Var}(s) \\ \text{let } s \text{ be } y \otimes z \text{ in } t[u/x] & \text{if } x \in \mathit{Var}(t) \end{cases} \\
(\text{let } s \text{ be } * \text{ in } t)[u/x] &\equiv \begin{cases} \text{let } s[u/x] \text{ be } * \text{ in } t & \text{if } x \in \mathit{Var}(s) \\ \text{let } s \text{ be } * \text{ in } t[u/x] & \text{if } x \in \mathit{Var}(t) \end{cases} \\
(\text{promote } r, \vec{s} \text{ for } y, \vec{z} \text{ in } t)[u/x] &\equiv \begin{cases} \text{promote } r[u/x], \vec{s} \text{ for } y, \vec{z} \text{ in } t & \text{if } x \in \mathit{Var}(r) \\ \text{promote } r, \vec{s} \text{ for } y, \vec{z} \text{ in } t[u/x] & \text{if } x \in \mathit{Var}(t) \end{cases} \\
(\text{derelict } t)[u/x] &\equiv \text{derelict } t[u/x] \\
(\text{discard } s \text{ in } t)[u/x] &\equiv \begin{cases} \text{discard } s[u/x] \text{ in } t & \text{if } x \in \mathit{Var}(s) \\ \text{discard } s \text{ in } t[u/x] & \text{if } x \in \mathit{Var}(t) \end{cases} \\
(\text{copy } s \text{ as } y, z \text{ in } t)[u/x] &\equiv \begin{cases} \text{copy } s[u/x] \text{ as } y, z \text{ in } t & \text{if } x \in \mathit{Var}(s) \\ \text{copy } s \text{ as } y, z \text{ in } t[u/x] & \text{if } x \in \mathit{Var}(t) \end{cases} \\
t[u_1/x_1, \ldots, u_n/x_n] &\equiv t[u_1/x_1] \ldots [u_n/x_n]
\end{aligned}
$$

Figure 5: Substitution metasyntax

$$(\lambda x.t)u \to t[u/x]$$
$$\text{let } * \text{ be } * \text{ in } t \to t$$
$$\text{let } r \otimes s \text{ be } x \otimes y \text{ in } t \to t[r/x, s/y]$$

$$\text{derelict (promote } \vec{u} \text{ for } \vec{x} \text{ in } t) \to t[\vec{u}/\vec{x}]$$
$$\text{discard (promote } \vec{s} \text{ for } \vec{x} \text{ in } u) \text{ in } t \to \text{discard } \vec{s} \text{ in } t$$

$$\text{copy (promote } \vec{s} \text{ for } \vec{x} \text{ in } u) \text{ as } y, z \text{ in } t$$
$$\to \text{copy } \vec{s} \text{ as } \vec{y}, \vec{z} \text{ in } t[\text{promote } \vec{y} \text{ for } \vec{x} \text{ in } u/y, \text{promote } \vec{z} \text{ for } \vec{x} \text{ in } u/z]$$

Figure 6: β-contraction rules

We distinguish two fragments of the linear λ-calculus: the *pure* and *modal* fragments. The former are those involving types and type operators '$\multimap$', '$\otimes$', and 'I', whilst the latter are those involving the modal type operator '!' — i.e. 'promote', 'derelict', 'discard', and 'copy' expressions. Dividing the calculus in this way allows us to deal with the simpler, purely-linear subset of the calculus separately from terms that require contraction and weakening.

$$(\text{let } s \text{ be } * \text{ in } t)u \to \text{let } s \text{ be } * \text{ in } tu$$
$$\text{let } (\text{let } r \text{ be } * \text{ in } s) \text{ be } * \text{ in } t \to \text{let } r \text{ be } * \text{ in let } s \text{ be } * \text{ in } t$$
$$\text{let } (\text{let } r \text{ be } * \text{ in } s) \text{ be } x \otimes y \text{ in } t \to \text{let } r \text{ be } * \text{ in let } s \text{ be } x \otimes y \text{ in } t$$
$$\text{discard } (\text{let } r \text{ be } * \text{ in } s) \text{ in } t \to \text{let } r \text{ be } * \text{ in discard } s \text{ in } t$$
$$\text{copy } (\text{let } r \text{ be } * \text{ in } s) \text{ as } x, y \text{ in } t \to \text{let } r \text{ be } * \text{ in copy } s \text{ as } x, y \text{ in } t$$

$$(\text{let } s \text{ be } x \otimes y \text{ in } t)u \to \text{let } s \text{ be } * \text{ in } tu$$
$$\text{let } (\text{let } r \text{ be } x \otimes y \text{ in } s) \text{ be } * \text{ in } t \to \text{let } r \text{ be } x \otimes y \text{ in let } s \text{ be } * \text{ in } t$$
$$\text{let } (\text{let } r \text{ be } w \otimes x \text{ in } s) \text{ be } y \otimes z \text{ in } t \to \text{let } r \text{ be } w \otimes x \text{ in let } s \text{ be } y \otimes z \text{ in } t$$
$$\text{discard } (\text{let } r \text{ be } x \otimes y \text{ in } s) \text{ in } t \to \text{let } r \text{ be } x \otimes y \text{ in discard } s \text{ in } t$$
$$\text{copy } (\text{let } r \text{ be } w \otimes x \text{ in } s) \text{ as } y, z \text{ in } t \to \text{let } r \text{ be } w \otimes x \text{ in copy } s \text{ as } y, z \text{ in } t$$

$$(\text{discard } s \text{ in } t)u \to \text{discard } s \text{ in } tu$$
$$\text{let } (\text{discard } r \text{ in } s) \text{ be } * \text{ in } t \to \text{discard } r \text{ in let } s \text{ be } * \text{ in } t$$
$$\text{let } (\text{discard } r \text{ in } s) \text{ be } x \otimes y \text{ in } t \to \text{discard } r \text{ in let } s \text{ be } x \otimes y \text{ in } t$$
$$\text{discard } (\text{discard } r \text{ in } s) \text{ in } t \to \text{discard } r \text{ in discard } s \text{ in } t$$
$$\text{copy } (\text{discard } r \text{ in } s) \text{ as } x, y \text{ in } t \to \text{discard } r \text{ in copy } s \text{ as } x, y \text{ in } t$$

$$(\text{copy } s \text{ as } x, y \text{ in } t)u \to \text{copy } s \text{ as } x, y \text{ in } tu$$
$$\text{let } (\text{copy } r \text{ as } x, y \text{ in } s) \text{ be } * \text{ in } t \to \text{copy } r \text{ as } x, y \text{ in let } s \text{ be } * \text{ in } t$$
$$\text{let } (\text{copy } r \text{ as } w, x \text{ in } s) \text{ be } y \otimes z \text{ in } t \to \text{copy } r \text{ as } w, x \text{ in let } s \text{ be } y \otimes z \text{ in } t$$
$$\text{discard } (\text{copy } r \text{ as } x, y \text{ in } s) \text{ in } t \to \text{copy } r \text{ as } x, y \text{ in discard } s \text{ in } t$$
$$\text{copy } (\text{copy } r \text{ as } w, x \text{ in } s) \text{ as } y, z \text{ in } t \to \text{copy } r \text{ as } w, x \text{ in copy } s \text{ as } y, z \text{ in } t$$

Figure 7: Commutation rules

$$
\begin{aligned}
(\lambda x.t)u &= t[u/x] \\
\lambda x.tx &= t \\
\text{derelict (promote } \vec{s} \text{ for } \vec{z} \text{ in } t) &= t[\vec{s}/\vec{z}] \\
\text{promote } t \text{ for } x \text{ in derelict } x &= t \\
\text{discard (promote } \vec{s} \text{ for } \vec{z} \text{ in } u) \text{ in } t &= \text{discard } \vec{s} \text{ in } t \\
\text{promote } u, \vec{s} \text{ for } y, \vec{z} \text{ in discard } y \text{ in } t &= \text{discard } u \text{ in promote } \vec{s} \text{ for } \vec{z} \text{ in } t \\
\text{copy } s \text{ as } x, y \text{ in discard } x \text{ in } t &= t[s/y] \\
\text{copy } s \text{ as } x, y \text{ in discard } y \text{ in } t &= t[s/x] \\
\text{copy } s \text{ as } x, y \text{ in } t &= \text{copy } s \text{ as } y, x \text{ in } t \\
\text{copy } s \text{ as } w, x \text{ in copy } w \text{ as } y, z \text{ in } t &= \text{copy } s \text{ as } w, z \text{ in copy } w \text{ as } x, y \text{ in } t
\end{aligned}
$$

$$
\begin{aligned}
&\text{promote (promote } \vec{r} \text{ for } \vec{x} \text{ in } u), \vec{s} \text{ for } y, \vec{z} \text{ in } t \\
&= \text{promote } \vec{r}, \vec{s} \text{ for } \vec{y}, \vec{z} \text{ in } t[\text{promote } \vec{y} \text{ for } \vec{x} \text{ in } u/y]
\end{aligned}
$$

$$
\begin{aligned}
&\text{copy (promote } \vec{s} \text{ for } \vec{z} \text{ in } u) \text{ as } x, y \text{ in } t \\
&= \text{copy } \vec{s} \text{ as } \vec{x}, \vec{y} \text{ in } t[\text{promote } \vec{x} \text{ for } \vec{z} \text{ in } u/x, \text{promote } \vec{y} \text{ for } \vec{z} \text{ in } u/y]
\end{aligned}
$$

$$
\begin{aligned}
&\text{promote } u, \vec{s} \text{ for } y, \vec{z} \text{ in copy } y \text{ as } w, x \text{ in } t \\
&= \text{copy } u \text{ as } w, x \text{ in promote } w, x, \vec{s} \text{ for } w, x, \vec{z} \text{ in } t
\end{aligned}
$$

$$
\begin{aligned}
t[\text{let } s \text{ be } * \text{ in } u/x] &= \text{let } s \text{ be } * \text{ in } t[u/x] \\
t[\text{let } s \text{ be } y \otimes z \text{ in } u/x] &= \text{let } s \text{ be } y \otimes z \text{ in } t[u/x] \\
t[\text{discard } s \text{ in } u/x] &= \text{discard } s \text{ in } t[u/x] \\
t[\text{copy } s \text{ as } y, z \text{ in } u/x] &= \text{copy } s \text{ as } y, z \text{ in } t[u/x]
\end{aligned}
$$

Figure 8: Term equalities

Chapter 3

Term graphs

3.1 Link graphs

Term graphs are often used in order to reflect a notion of shared data in the underlying term structure, as first formalised by Wadsworth (1971) in the context of the λ-calculus. We however will delay any analysis of the graphs' applicability to term sharing until the next chapter (§4), and focus only on the structure of the graphs *sans* sharing. In the specific context of first-order terms these graphs are uninteresting, being essentially a tree of function symbols, as can be formalised with respect to the terms' *positions* (Terese 2003). With second-order terms these become more distinct in their structure, corresponding to so-called *Bourbaki notation* (Bourbaki 1954). As we progress further on towards third- and higher-order terms their structure becomes more interesting again, as it becomes clear that they reflect a particular representation of higher-order abstract syntax — one which, despite its being a generalisation of those aforementioned notations, has to the best of our knowledge not yet been explored in the literature.

We use the 'structures' of Guerrini (1996) to describe all of the forms of graph we will encounter during this investigation. Guerrini defines his term graphs, the most general of which are called *structures*, as forms of hypergraph. Each arrow in the term graph is treated as a vertex, and each link a hyperarc, which at first may seem counterintuitive. However, this presentation is also used for the more

common 'jungle' term graphs (Plump 1993), and is in fact more appropriate a representation. When it comes to the encoding of terms into these graphs (see §3.2, §3.3) an n-ary constant may be realised as a single hyperedge, whereas with 'ordinary' graphs a vertex together with n additional edges must be used. Hypergraphs thereby lead to much cleaner translations.

Our definitions are also based on later work by Accattoli (2011), who uses instead the term *link graph*. We use 'structure', 'ℓ-structure', '$s\ell$-structure', etc., when referring to a specific data type, and 'link graph' or 'sharing graph' when referring to these graphs in a broader, collective sense, whether with or without sharing operators.

3.1.1 Hypergraphs

Definition 28. An *undirected hypergraph* is a pair $G = (V, E)$ where $V(G) = V = \{v_1, \ldots, v_n\}$ is a set of objects, the *vertices* of G, and $E(G) = E = \{e_1, \ldots, e_n\}$ is a set of finite sequences of distinct vertices of G, the *hyperedges* of G.

Definition 29. A directed hyperedge, or *hyperarc*, is a pair $e = (\partial_t(e), \partial_h(e))$ of (possibly empty) disjoint ordered sets of vertices, $\partial_t(e)$ being the set of tail vertices, or *tail* of e, and $\partial_h(e)$ the set of head vertices, or *head* of e. The undirected hyperedge associated with a hyperarc e is the concatenation of its tail and head: $\partial(e) = \partial_t(e)\partial_h(e)$.

Definition 30. A hypergraph G is *directed* when all of its hyperedges are hyperarcs, *i.e.* a directed hypergraph is a pair $G = (V, E)$ of vertices V and hyperarcs E such that $(V, \{\partial(e) \mid e \in E\})$ is an undirected hypergraph.

Some of the links we will consider have a natural orientation independent from that of the hyperarc given by the cardinalities of their head or tail: they have a unique head arrow and a tail of arbitrary cardinality, or vice versa.

Definition 31. A k-ary *backward* hyperarc e is a hyperarc where $|\partial_t(e)| = k$ and $|\partial_h(e)| = 1$. A k-ary *forward* hyperarc e is a hyperarc where $|\partial_t(e)| = 1$ and $|\partial_h(e)| = k$.

The hyperarc e is an *incoming* hyperarc of the vertex v when $v \in \partial_h(e)$; it is an *outgoing* hyperarc when $v \in \partial_t(e)$. We assume that the incoming and outgoing hyperarcs of a vertex are ordered, and denote by $\partial_{\text{in}}(v)$ and $\partial_{\text{out}}(v)$ the respective sequences of these. Incoming and outgoing hyperarcs of a vertex are both said to be *incident* to it.

Definition 32. A *source root* is a vertex v with no incoming hyperarcs, $\partial_{\text{in}}(v) = \emptyset$; a *target root* a vertex with no outgoing hyperarcs, $\partial_{\text{out}}(v) = \emptyset$; and an *isolated* vertex one with no incident hyperarcs.

The set $\partial_{V_s}(G)$ is the set of source roots of G; $\partial_{V_t}(G)$ the target roots of G; and the union $\partial_V(G) = \partial_{V_s}(G) \cup \partial_{V_t}(G)$ the roots of G.

Definition 33. The *border* of a hypergraph G is the set $\partial_E(G)$ of hyperarcs incident to a root of G.

3.1.2 Link graphs

The vertices of our hypergraphs are of a particular shape: they may be seen as *arrows* representing directional one-to-one connections between pairs of *links*.

Definition 34. A *link type* is a triple comprising a *type name*, a set of named *input ports*, and a set of named *output ports*.

Definition 35. A *link* of a given type is a hyperarc labelled with the corresponding type name and with a tail (*resp.* head) vertex for each of the input (*resp.* output) ports of the type. The tail vertices $\partial_t(e)$ and head vertices $\partial_h(e)$ of a link e are together said to be the *doors* of e. We assume that isolated links, with no doors, are not allowed.

Definition 36. An *arrow* is a vertex v with at most one incoming link and one outgoing link, *i.e.* $\partial_{\text{in}}(v) \cup \partial_{\text{out}}(v) \neq \emptyset$, with $|\partial_{\text{in}}(v)| \leq 1$ and $|\partial_{\text{out}}(v)| \leq 1$.

Consider the dual G^* of a hypergraph G in which all the vertices are arrows. We see that G^* is a directed graph (*not* a hypergraph), since all of its edges — the arrows of G — have only one source and one target node. This is the reason vertices are called “arrows” in our hypergraphs.

Definition 37. A *link signature* Σ is a set of link type names. A Σ-naming for the links of the hypergraph G is a map $\tau_G : E(G) \to \Sigma$ that assigns a link type to each hyperedge of G.

Definition 38. A *structure* $\mathscr{G}$ over the link signature Σ is a pair (G, τ_G) in which: the hypergraph G has at least one root; all vertices of G are arrows; and the map τ_G is a Σ-naming for the links of G.

Let $\mathscr{G}$ be a structure over Σ and let $\star \in \Sigma$, then the set $E_\star(G)$ is the set of links in $\mathscr{G}$ of type $\star$, *i.e.* $E_\star(G) = \{e \in E(G) \mid \tau_G(e) = \star\}$.

As the border of our structure has vertices with no source or no target (or both), if we chose to work with the dual of our hypergraphs then there would be graphs in which the source and target maps of the edges were partial, or else we would have had to append a root node to each dangling edge represented by a root. This is another reason why these structures are better presented as hypergraphs.

Definition 39. A *substructure* $\mathscr{R}$ of a structure $\mathscr{G}$, or $\mathscr{G}$*-substructure*, is a structure such that $V(R) \subseteq V(G)$ and $E(R) \subseteq E(G)$, and with $\tau_R = \tau_G|_{E(R)}$.

Since a structure does not contain isolated vertices, the $\mathscr{G}$-substructures may also be seen as the parts of $E(G)$. In fact, not only is any $\mathscr{G}$-substructure $\mathscr{R}$ uniquely determined by the set $E(R) \subseteq E(G)$, but also, given a set of links $E \subseteq E(G)$, there exists a unique $\mathscr{G}$-substructure such that $E(R) = E$.

With this in mind, the inclusion relation and set operations may be extended to $\mathscr{G}$-substructures:

- $\mathscr{R}_0 \subseteq \mathscr{R}_1$ *iff* $E(R_0) \subseteq E(R_1)$;
- $\mathscr{R} = \mathscr{R}_0 \cap \mathscr{R}_1$ is the $\mathscr{G}$-substructure such that $E(R) = E(R_0) \cap E(R_1)$;
- $\mathscr{R} = \mathscr{R}_0 \cup \mathscr{R}_1$ is the $\mathscr{G}$-substructure such that $E(R) = E(R_0) \cup E(R_1)$;
- $\mathscr{R} = \mathscr{R}_0 \setminus \mathscr{R}_1$ is the $\mathscr{G}$-substructure such that $E(R) = E(R_0) \setminus E(R_1)$;

where $\mathscr{R}_0$ and $\mathscr{R}_1$ are $\mathscr{G}$-substructures.

These considerations do not apply if we replace links with vertices. In fact, even if a $\mathscr{G}$-substructure is uniquely determined by the set of its vertices, it is no longer true that for each set of vertices $V \subseteq V(G)$ there exists a $\mathscr{G}$-substructure $\mathscr{R}$

with $V = V(R)$. In any case, to each set of vertices $V \subseteq V(G)$ we may associate the $\mathscr{G}$-substructure $\{e \in E(G) \mid \partial(e) \subseteq V\}$ composed of those links whose doors are all contained in V, *i.e.* the biggest $\mathscr{G}$-substructure $\mathscr{R}$ such that $V(R) \subseteq V$.

The border $\partial_E(G)$ of a structure $\mathscr{G}$ is a $\mathscr{G}$-substructure, and its complement $\partial_E(\mathscr{G}) = \mathscr{G} \setminus \partial_E(G)$ is the *interior* of $\mathscr{G}$. We say that the links and arrows of the interior of $\mathscr{G}$ are the *interior* links and arrows of $\mathscr{G}$.

3.1.3 Interaction Nets

Interaction Nets (Lafont 1990) are a class of graph rewriting system in which links (called 'agents') may interact with one another in a binary fashion, such that rewrite steps may occur whereby a pair of links are rewritten to form a new subgraph comprising a network of agents, which likewise go on to interact. These Nets have been shown to have certain properties which will prove useful later on, most notably forming the basis of Lévy optimality in Lamping–Gonthier sharing graphs (§4).

Definition 40. Each link type in an Interaction Net system has one designated *principal port*, and n *auxiliary ports*. A pair of links that are connected via a single vertex by their respective principal ports are called an *active pair*, the Interaction Net analogue of a redex.

An *interaction rule* is a graph rewrite rule (l, r) where l comprises only an active pair. A border vertex occurs in r iff it occurs in l, but two vertices may also be *unified* so as to connect an incoming arrow to a outgoing one, causing the arrow to pass through the resulting structure unabated. In this case the incoming arrow's source port and outgoing arrow's target port are connected.

An *Interaction Net system* (Σ, R) is a set R of interaction rules with link graphs over the link signature Σ, whose left-hand sides each have a unique pair of links (i.e. a unique pair of link labels). There is therefore no ambiguity as to which interaction rule applies (or, indeed, does not) for any pair of facing links.

3.2 Graphical abstract syntax

There are two complementary ways one can formalise combinatory logic as a rewrite system, one having implicit applications and the other explicit. In the former case one would define the S combinator as $Sxyz \to xz(yz)$; in the latter case, the applications are explicit and must be included in both sides of the rule, as in $@(@(@(S,x),y),z) \to @(@(x,z),@(y,z))$. Indeed, the second-order combinatory reduction systems were originally defined with implicit application (Klop 1980), and were later reformulated to have fixed arity and so explicit application (Klop, Van Oostrom & Van Raamsdonk 1993). One can see the difference between these two formalisations as the latter having a type system (i.e. fixed arities), and the former not. One holdout even in these later combinatory reduction systems, however, is that abstractions do not contribute to this type, so they are not quite identical to the second-order fragment of Nipkow's (1991) HRSs.

When dealing with higher-order rewrite systems in general, the standard choice of the simply-typed λ-calculus to be used as a substitution calculus is technically unnecessary, as in fact the untyped λ-calculus would be quite sufficient in theory. Yet, just as having type constraints, such as symbols with fixed arities, is useful in first- and second-order rewriting, the simple type system is useful for the same purpose in higher-order rewriting. This relationship between arity constraints and the type system of a substitution calculus will become quite crucial in § 3.3, with simple types seeming to be the purest higher-order counterpart of arity constraints, which deals only with the first-order notion of subterms.

Of course, the fact that $\beta\eta$-matching is decidable for the simply-typed λ-calculus (with one base type) also goes a long way to inducing a coherent higher-order rewrite relation, especially if as in the untyped λ-calculus a term may have meaningless subterms, i.e. unlabelled leaves in its Böhm tree (Barendregt 1984). It is for these reasons that substitution calculi are generally assumed to have decidable equality between terms, for instance. A type system may therefore be useful to a substitution calculus for its properties alone. Although the previously discussed similarity between types and arities applies to simple types in general, nothing can be done to exploit this relationship that cannot with 'intuitionistic' HRSs, unless we move to a more expressive type system which allows us to describe substructural

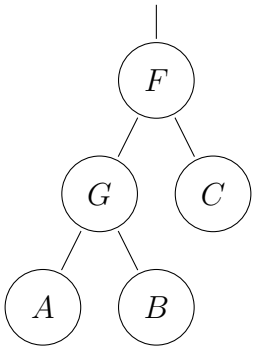


Figure 9: Term tree of $F(G(A, B), C)$

properties of subtypes. This will be explored later (see §5).

3.2.1 Term trees

Term trees are the simplest form of link graph we deal with, and which serve as the abstract syntax tree for first-order terms.

Definition 41. A *term tree* is an acyclic structure whose links each have exactly one tail vertex.

In order to translate a first-order term signature into a link graph signature, each n-ary function symbol is mapped to a correspondingly labelled hyperarc with n heads (and 1 tail).

Example 4. Figure 9 illustrates a tree representing the term $F(G(A, B), C)$. F and G each corresponding to links with 2 heads and 1 tail, whilst A, B, and C each correspond to links with only 1 tail, as determined by their respective arities. The lines between these links are arrows, or vertices. At the root of the tree is a boundary arrow (vertex) with only a single outgoing link, F.

3.2.2 Bourbaki notation

It is not clear at first how these structures ought to be generalised so as to account for terms of the λ-calculus, having the added complication of bound variables. One

method, following the classical representation of terms, is to straightforwardly declare a variable's name at its abstraction, and to perform α-renaming as appropriate during reduction of the term. This was the method used in Wadsworth's (1971) original λ-calculus formalisation. Another method is to omit the name of the variable at its declaration, and to instead use De Bruijn indices (De Bruijn 1972), which during reduction are adjusted by offsets so as to always refer to the correct relative scope. Both of these approaches retain the term representation's tree structure, though at the cost of requiring adjustments in order to correct the variable occurrences as new scopes are introduced during reduction.

As first shown by Bourbaki (1954), a term of the λ-calculus may be represented not as a tree containing named variables, but instead as a kind of graph ("*assemblages*") in which bound variables are drawn as edges ("*liens*") that arc back to their binding abstraction. Translating (linear) Combinatory Reduction Systems is fairly straightforward. Function symbols, as in first-order term rewriting systems, are again represented as links with 1 tail and n heads, with its tail corresponding to its context and its heads corresponding to each of its arguments. Abstraction links, on the other hand, have 2 tails and 1 head: one tail represents its context, the other its bound variable, whilst the head represents its body.

CRS rules are not necessarily possible to translate into Interaction Nets, however, due to the strict constraints on the nets' interaction rules. As a result, a subclass of CRSs was introduced as a class of term rewriting systems with Interaction Nets as a compilation target, which were termed Interaction Systems (Asperti & Laneve 1994).

Whereas Interaction Nets in general need not have any particular form corresponding to a term structure, Bourbaki notation does require that a variable be 'in scope'. This means the binder must *dominate* the variable occurrence: there must be a path through the symbol's subterm arrow and back through the corresponding binder arrow (Kahl 1998). This is illustrated in Figure 10: in the graph on the left, this means there must be a path from G through its southern arrow, returning through its southwestern arrow; compare the term on the right, which does not properly represent a second-order term due to the wrongly-bound variable occurrence, where there is no such path for F.

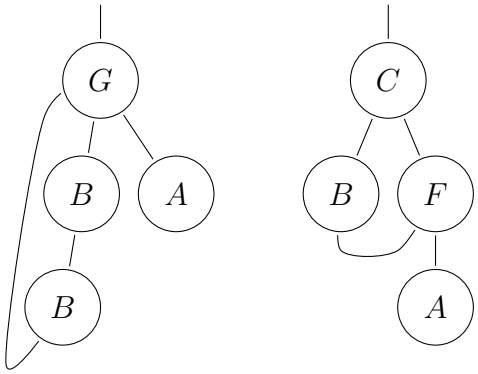


Figure 10: Bourbaki graph $G(x.B(B(x)), A)$, and invalid $C(B(x), F(x.A))$

3.3 Higher-order abstract syntax

Although this representation does account for the bound variables of the λ-calculus, from the perspective of higher-order abstract syntax (Pfenning & Elliot 1988), wherein λ-terms are only second-order, this is its limitation: terms within third- and higher-order abstract syntax cannot be represented. For example, Okasaki (1998) describes a practical set of parser combinators from third- up to sixth-order, which cannot be represented as a Bourbaki graph without first embedding them into a lower-order syntax such as the untyped λ-calculus. We shall show that these Bourbaki graphs are a second-order fragment of a more general graphical representation of higher-order abstract syntax, which we call *well-typed* link graphs. Moreover, when dealing with match sequential systems these graphs are equivalent to a particular class of Interaction Net, which we call 'higher-order syntax nets'.

In this section we only consider purely linear terms — i.e. without any of the structural expressions 'promote', 'derelict', 'copy', or 'discard' — which means that each variable must occur exactly once within a term. The structural expressions require additional infrastructure, and so are dealt with in §4.

Definition 42. Link graphs are constructed inductively in a way that tightly mirrors the construction of terms. Just as terms are derived by a formula $\Gamma \vdash t : \tau$, a link graph is derived by a formula $[\![\Gamma]\!] \vdash [\![t]\!] : [\![\tau]\!]$. $[\![t]\!]$ is the link graph itself, which we are constructing. $[\![\tau]\!]$, and the members of the set $[\![\Gamma]\!]$, are *arrow types*, which describe the *interface* (i.e. the margin) of the link graph. An arrow type is

a linear type whose type atom occurrences each correspond to vertices within the link graph. Furthermore, all marginal vertices are present in the interface.

Theorem 2. *A derivation* $[\![\Gamma]\!] \vdash [\![t]\!] : [\![\tau]\!]$ *implies* $\partial_E([\![t]\!])$ *is the smallest set such that* $[\![\Gamma]\!] \cup \{[\![\tau]\!]\} \subseteq \mathit{Typ}(\partial_E([\![t]\!]))$.

Proof. By construction, no link graph $[\![t]\!]$ where $[\![\Gamma]\!] \vdash [\![t]\!] : [\![\tau]\!]$ may contain marginal vertices not occurring in $[\![\Gamma]\!]$ or $[\![\tau]\!]$; nor can internal vertices so occur. □

For each arrow α occurring in an arrow type $[\![\tau]\!]$, whether that arrow is outgoing or incoming (with respect to a link graph $[\![t]\!]$ derived in part from that type) depends on the *polarity* of the occurrence in $[\![\tau]\!]$.

Definition 43. We define the *polarity*, *positive* (P) or *negative* ($\neg P$), of a type occurrence $(\tau, C[])$ inductively as follows, where $P(\tau) = P(\tau, \Box)$.

$$\begin{aligned}
& P(\tau, \Box) \\
P(\sigma \otimes \tau, C[]) &\implies P(\sigma, C[\Box \otimes \tau]) \wedge P(\tau, C[\sigma \otimes \Box]) \\
P(\sigma \multimap \tau, C[]) &\implies \neg P(\sigma, C[\Box \multimap \tau]) \wedge P(\tau, C[\sigma \multimap \Box]) \\
\neg P(\sigma \otimes \tau, C[]) &\implies \neg P(\sigma, C[\Box \otimes \tau]) \wedge \neg P(\tau, C[\sigma \otimes \Box]) \\
\neg P(\sigma \multimap \tau, C[]) &\implies P(\sigma, C[\Box \multimap \tau]) \wedge \neg P(\tau, C[\sigma \multimap \Box])
\end{aligned}$$

The notion of polarity is also extended to atom occurrences within a type. So, for instance, the type $0 \multimap 0$ has a positive atom on the right and a negative atom on the left. We may annotate these atoms' polarities by $0^- \multimap 0^+$.

The negativity of ψ in a type occurrence $\psi \multimap \phi$ can be understood through the encoding of linear implication in classical linear logic as $\psi^\perp \parr \phi$. Polarity is defined the same way in classical logic, where $\psi \to \phi$ may be encoded as $\neg\psi \vee \phi$. In each case the negation of ψ leads to a reversal in polarity.

Type occurrences in the type context of a term derivation are negative at their root, while those in the type itself are positive.

$$\Gamma \vdash t : \tau \implies \Big(\bigwedge_{x:\sigma \in \Gamma} \neg P(\sigma) \Big) \wedge P(\tau)$$

This can be seen most clearly from the abstraction rule,

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x^{\sigma}.t : \sigma \multimap \tau}$$

where σ occurs negatively in $\sigma \multimap \tau$ and therefore must be effectively negative in the type context.

Thus it could be said that the basis provides the 'input' arrows for the term graph, and the term type the 'output' arrows. However, these 'input' arrows need not all be incoming, since with a higher-order variable in the basis, e.g. $x : 0 \multimap 0 \vdash t : \tau$, the basis contains both an incoming and an outgoing arrow — the right and left type atoms of $0 \multimap 0$, respectively. This is because although the basis does act as an 'input', it requires a value to be provided in order to provide its value, hence the variable's type $0 \multimap 0$.

3.3.1 Constants

When a term is translated to a link graph, constants in the term are represented as links, with each distinct symbol in the term signature corresponding to a distinct label in the link signature.

Definition 44. A symbol $F : \tau \in \Sigma$ is mapped to a link label $[\![F]\!] : (|\tau|^+, |\tau|^-) \in [\![\Sigma]\!]$, where $|\tau|^+$ and $|\tau|^-$ are the number of positive and negative atoms in τ, respectively.

Note that positive atoms correspond to the tail, and negative atoms the head, of a link.

Example 5. If $G : (0 \multimap 0) \multimap 0 \in \Sigma$, then this maps to the link $G : (1, 2) \in [\![\Sigma]\!]$. With polarity annotations, the type of G is $(0^+ \multimap 0^-) \multimap 0^+$. The term G then maps to the link graph whose leftmost and rightmost arrows (vertices) are at the two tails of the hyperarc — corresponding to the constant's type's positive atom occurrences — and its middle arrow (vertex) at its head — corresponding to the negative type atom occurrence(s). This graph can be illustrated as in Figure 11.

Notation 1. It is worth taking a moment to explain the diagrammatic notation used here, as it is our own and will be commonly used in the remainder of this

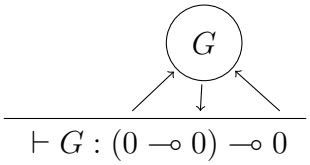


Figure 11: $\vdash G : (0 \multimap 0) \multimap 0$

thesis. As the reader may already have gleaned, this notation takes inspiration from Gentzen's standard notation for natural deduction, where a derivation comprises zero or more premises, which are underscored by a horizontal line, under which the conclusion is placed. A constant has no premises, and so in the standard notation has nothing above this horizontal score. However, we opt to place above it an illustration of the term graph constructed by the given rule. In this case a link with label G is connected by its head and two tails to three vertices that are, as vertices in the margin of the link graph, identified by the type atoms in the judgement $\vdash G : (0 \multimap 0) \multimap 0$, which forms its interface. Those rules that also include premises require further notation, as we will see in the next section.

3.3.2 Expressions

It is only constants that are represented in the graph as links — variables, abstractions, and applications, as well as expressions dealing with $\otimes$-types, only affect the ways in which the constants' links are arranged. However, expressions dealing with exponentiation, for which translations are defined later (§4.3), will result in other (special) links being added to the link graph. Putting those aside for the time being, we now deal with these expressions and how they may be used to inductively construct link graphs, so as to comprise the link graphs of constants as already described, as well as those of other expressions.

Definition 45. The rules for translating expressions in purely linear λ-terms are illustrated in Figure 12. In each case, each formula $\Gamma \vdash t : \tau$ maps to a link graph $[\![\Gamma \vdash t : \tau]\!]$.

Notation 2. In cases where a term graph also has premises, since both they and the constructed link graph would be placed above the score, we choose to cleave

the score in two, so that the conclusion is placed below a lower score and the premises above an upper score, between the two of which is placed the link graph, with its vertices associated with atoms in each, which together form its interface. Figure 12, for instance, comprises a number of these illustrations, most of which have premises and therefore include both scores.

Another sophistication is that although we draw a single arrow to represent the vertices of type τ, any arrow corresponding to a type may stand for some finite number of arrows in one particular instantiation of the rule. For example, the rule $\vdash F : \tau$ has only one arrow drawn between the link labelled F and the interface at τ. However, in Figure 11, which is a particular instantiation of this rule for the constant $G : (0 \multimap 0) \multimap 0$, there is not only one arrow, but three, as here there are three type atoms in $\tau = (0 \multimap 0) \multimap 0$. These arrows may also differ in directionality as necessary for the polarity of the type atoms to which it is associated, as is the case for G.

Example 6. We will step through a simple example of term construction: that of the identity function $\lambda x^0.x$. As with ordinary construction of terms by inductive rules, we begin here with the judgement $x : 0 \vdash x : 0$. This rule, sometimes called *Axiom*, has no premises, and so relies on no pre-existing proposition. Thus, $x : 0 \vdash x : 0$ maps to the link graph in Figure 13.

This link graph comprises a single vertex, which occurs in its interface twice, written each time as 0. We then construct an abstraction expression using this proposition as a premise; we write the entire inductive construction of the term *à la* natural deduction, as on the left of Figure 14. It may alternatively be drawn as on the right, removing intermediary steps used to construct the graph.

Notation 3. We have extended our diagrammatic notation here in the natural way, allowing us to construct larger term graphs from smaller ones and their interfaces, which form a tree of judgements in much the same way as in standard natural deduction, albeit with the constructed term graph drawn between these as well. These composite term graphs can be redrawn, often much more simply, by removing these intermediary steps so that only the interface of the final term itself is illustrated. The resulting link graphs are clearly isomorphic to those with their intermediary interfaces still included.

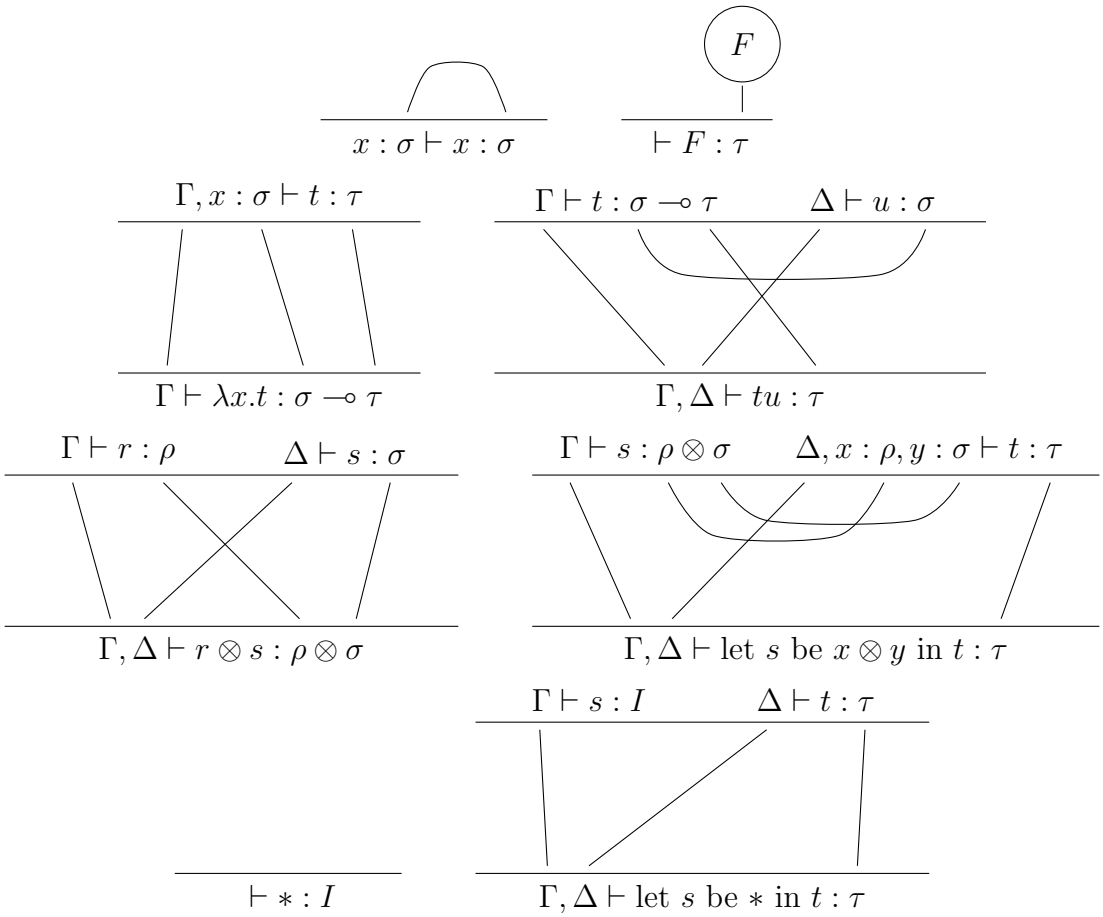


Figure 12: Translations from purely linear λ-terms to link graphs

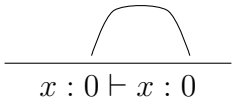


Figure 13: $x : 0 \vdash x : 0$

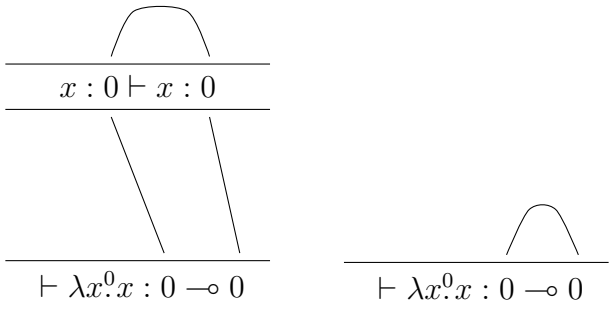


Figure 14: $\vdash \lambda x^0_!x : 0 \multimap 0$

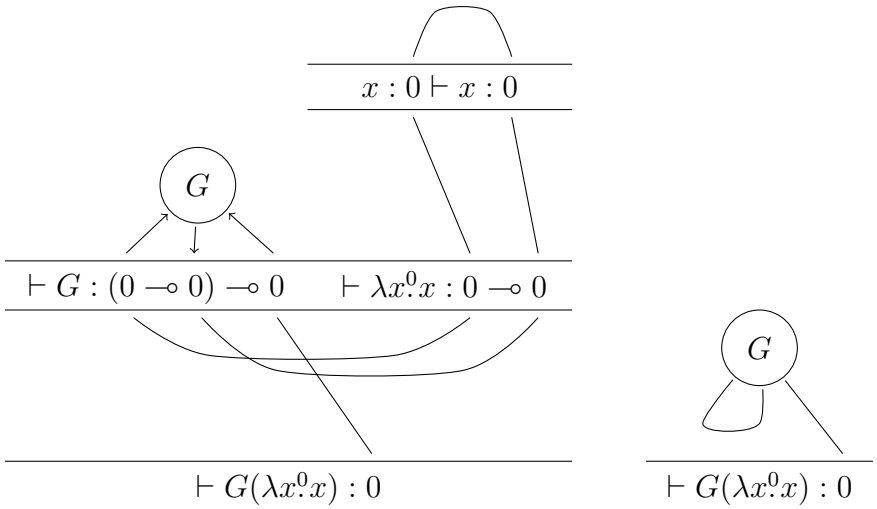


Figure 15: $\vdash G(\lambda x^0_!x) : 0$

Example 7. Now we demonstrate how these expressions may be used alongside constants in order to form link graphs comprising several constant links. Here we use the constant $\vdash G : (0 \multimap 0) \multimap 0$ of Example 5, together with the identity function $\vdash \lambda x^0_!x : 0 \multimap 0$ of Example 6, in order construct the term $\vdash G(\lambda x^0_!x) : 0$. This is illustrated in Figure 15; as before, this link graph may be equivalently redrawn as on the right.

Although the application rule in Figure 12 features only one arrow drawn between $t : \sigma \multimap \tau$ on the one hand, and $u : \sigma$ on the other, in this instantiation of the rule σ stands for $0 \multimap 0$, in which each of the two occurrences of the type atom 0 corresponds to a distinct arrow. Notably, this demonstrates how the two arrows drawn incident to σ and to τ in the abstraction rule may in fact

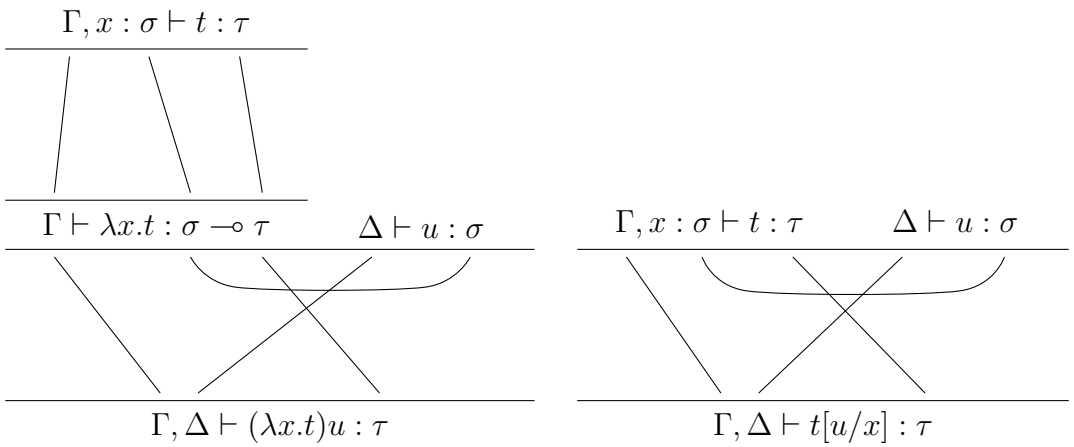


Figure 16: $(\lambda x.t)u$ and its β-reduct $t[u/x]$

represent two ends of the same arrow in a particular instantiation, specifically in this case by the fact that the *Axiom* rule introduces both ends of a single arrow into the interface, one in the type and the other in the type context.

3.3.3 Equivalence

There are two additional inductive rules for term graphs that are crucial for determining the equivalence of terms and their respective link graphs. These are the *substitution* and *identity* graphs. The necessity of these rules is made clear from their use in establishing the correctness of β- and η-reduction on (linear) functions. For the former, consider the term graph on the left of Figure 16, which forms the β-redex $(\lambda x^{\sigma}.t)u$.

As we know, $(\lambda x.t)u$ reduces to $t[u/x]$, where the substitution is not syntactic but is rather applied to the term so as to substitute, in t, u for each (free) occurrence of x. Since we know that $(\lambda x^{\sigma}.t)u = t[u/x]$, we are free to define a term graph for the substitution $t[u/x]$ which is equivalent to that of $(\lambda x.t)u$. Thus, this substitution, although not a syntactic construct, may be translated to a link graph as on the right of Figure 16. This, however, only serves to substitute a single term for a single variable; substitutions in general may substitute any number of terms for any (equal) number of variables. The general form for substitutions of any arity is illustrated in Figure 17.

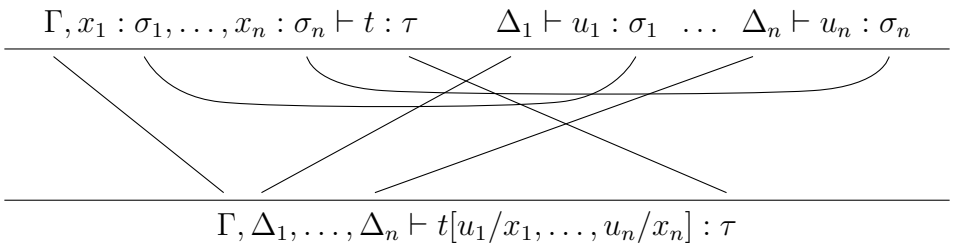


Figure 17: Substitution on link graphs

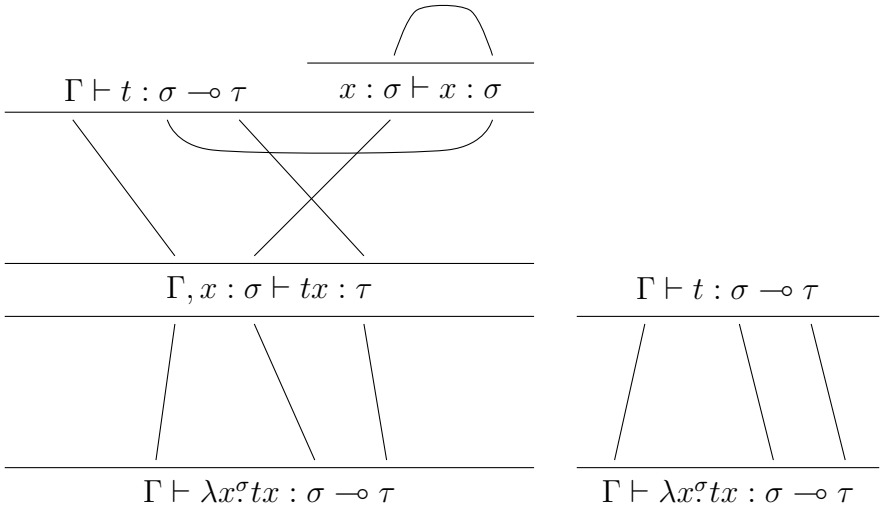


Figure 18: Isomorphic syntax graphs representing the term $\lambda x^{\sigma}.tx$

In a similar fashion, the term graph on the left of Figure 18 forms the η-redex $\lambda x.tx$. By straightening out the arrow that makes a detour through $x : \sigma \vdash x : \sigma$, this link graph may equivalently be drawn as on the right, with no effect on either the link graph or its interface. The link graph $[\![\lambda x.tx]\!]$ can thus be seen to be an identity on $[\![t]\!]$, meaning the two terms' graphs (as well as the terms themselves) are equivalent: any link graph $[\![C[(\lambda x.tx)]]\!]$ must be isomorphic to $[\![C[t]]\!]$. We also use this notion of identity on link graphs for the purpose of determining their equivalence. The rule, which equates any term graph $[\![t]\!]$ with itself, is illustrated in Figure 19.

Example 8. Another example of identity on link graphs can be seen when the earlier term graph $[\![\lambda x^{0}.x]\!]$ is applied to some term graph $[\![t]\!]$; see Figure 20. As before, this link graph may be drawn equivalently to the identity on $[\![t]\!]$.

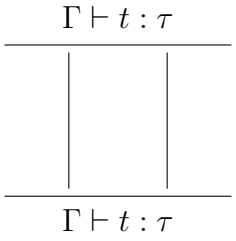


Figure 19: Identity on link graphs

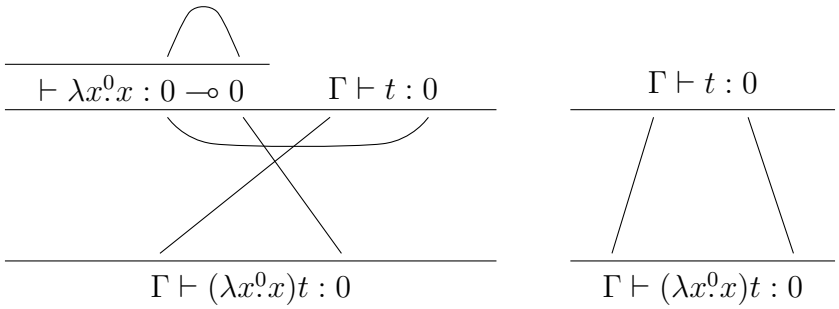


Figure 20: $\Gamma \vdash (\lambda x^0_!x)t : 0$

Given any term t, the link graph $[\![t]\!]$, constructed in the same manner, is a unique link graph representative of the equivalence class of t, up to isomorphism. When we say 'isomorphism' in this sense, we mean with respect to an isomorphism on link graphs (Guerrini 1996).

Lemma 1. *Let t_1 and t_2 be purely linear λ-terms; then $t_1 = t_2 \implies [\![t_1]\!] \equiv [\![t_2]\!]$.*

Proof. Case by case, the following graphs are isomorphic, pairwise. Referenced figures have been relegated to Appendix A.

1. $(\lambda x.t)u = t[u/x]$

   Figure 50.

2. $\lambda x.tx = t$

   Figure 51.

3. let $*$ be $*$ in $t = t$

   Figure 52.

4. let $r \otimes s$ be $x \otimes y$ in $t = t[r/x, s/y]$

   Figure 53.

5. $t[\text{let } s \text{ be } * \text{ in } u/x] = \text{let } s \text{ be } * \text{ in } t[u/x]$

   Figure 54.

6. $t[\text{let } s \text{ be } y \otimes z \text{ in } u/x] = \text{let } s \text{ be } y \otimes z \text{ in } t[u/x]$

   Figure 55.

These can be immediately observed from the respective figures through the removal of any intermediary graph construction step. □

Lemma 2. *Let t_1 and t_2 be purely linear λ-terms; then $[\![t_1]\!] \equiv [\![t_2]\!] \implies t_1 = t_2$.*

Proof. In any other case but those covered by Lemma 1, if $t_1 = t_2$ then they differ by a constant within the term. Let F and G be constants in signature Σ; then $F = G \iff F \equiv G \iff [\![F]\!] \equiv [\![G]\!]$. □

Theorem 3. $t_1 = t_2 \iff [\![t_1]\!] \equiv [\![t_2]\!]$.

Proof. Follows from Lemmas 1 and 2. □

Chapter 4

Sharing graphs

4.1 Jungles

If we relax our link graphs' constraint on the possible number of incoming hyperarcs' heads that may be incident to a given vertex, forming not a tree but a directed acyclic graph, then the resulting graphs would be equivalent to the *jungles* of Plump (1993). It may be noted that this corresponds to the intuitionistic counterpart to our essentially linear structures. Although this does not appear to have been remarked upon in the literature, it is straightforward that a vertex may thereby have constraints on its structural rules analogous to those in substructural type systems: similarly, if n is the number of incoming hyperarcs then the constraint $n \leq 1$ corresponds to an affine type system, and $n \geq 1$ a relevant type system (Walker 2005), e.g. the λI-calculus (Barendregt 1984).

These jungles may be reduced by performing rewrite steps modulo an *unfolding* relation, which may incrementally 'unfold' a shared subdag into multiple copies, each of whose own subdags are yet shared. This permits a rewrite step which occurs wholly within a shared subdag to be performed only once for all of the term positions in which it is shared. Only when a redex lies actually 'on the fold', on the boundary point at which a subdag is shared, is the unfolding step strictly necessary; if unfolding happens only at these times then the reduction process is known in functional programming languages as 'lazy evaluation'. This definition of unfolding is sufficient only for first-order terms, however, as the syntax graphs

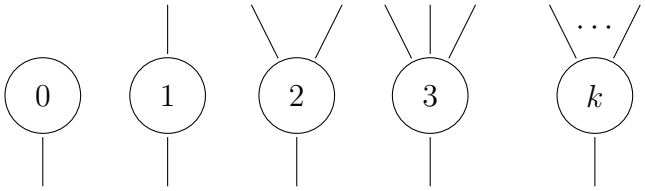


Figure 21: Eraser, indirector, duplicator, triplicator, and k-ary replicator

of higher-ordered terms require more sophisticated a sharing mechanism than is provided by jungles alone (§4.2, §4.3).

4.1.1 Replicators

Given that jungles are not sufficient for higher-order terms, it would be beneficial to use a sharing mechanism even in the first-order case which is as explicit as that required for such terms (§4.2.1), so that the two mechanisms may be better compared, 'like with like'. We will therefore make use of an explicit sharing operator called a *replicator* (Terese 2003). A replicator is in effect a dedicated operator representing a 'fold' in a jungle, a hyperarc projecting one subterm vertex at its head into several occurrence vertices at its tail, rather than having it as an implicit artefact of vertices' having multiple incoming hyperarcs.

We use particular terminology for the cases when k is: (0) an *eraser*; (1) an *indirector*; (2) a *duplicator*. Replicators where $k = 3$ are also sometimes referred to as *triplicators*, and so on. See Figure 21.

The purpose of the replicator is to share a single subterm between two superterms. It does this by *propagating* the fan through the arrow at its principal port, duplicating the other link incident to that arrow, and also duplicating itself so that there is a new duplicator at each other arrow to which that other link was incident. This process is illustrated in Figure 22.

4.1.2 Graph rewriting

In this thesis, from now on, we will make use of various graph rewriting systems, which generalise the notions of context and rewrite step from terms to directed graphs.

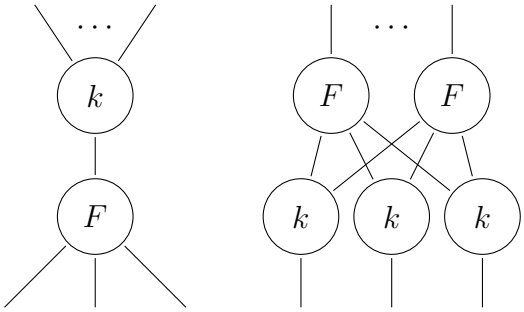


Figure 22: Propagation of k-ary replicator through link F

Definition 46. A *graph context* is a graph whose border vertices are each labelled with a unique label. This labelling is called the context's *interface*. Two graph contexts are called *complementary* if a source vertex in one corresponds to a target vertex with the same label in the other, and vice versa.

Definition 47. A graph G is a *subgraph* of a graph H if the sets of vertices and edges of G are subsets of the vertices and edges of H.

Note that if G is a subgraph of H, then H is isomorphic to $C[\bar{G}]$, where $\bar{G}$ is a context having G for a graph and the vertices "cut free" from H for an interface.

Definition 48. A graph rewrite rule comprises two graphs (L, R), written $L \to R$. A rule induces a relation $C[L] \to C[R]$ for any graph context C.

A graph rewrite system is a set of rules, inducing the union of its rules.

Replicators, as described in the previous section, will add to a graph rewrite system an infinite set of graph rewrite rules for handling propagation, one for each arity k and each other symbol F, as per Figure 22. However, they apply only to dags, and not to arbitrary graphs, since they only "make sense" so long as two replicators do not meet. A generalisation for this case is the subject of the following section.

4.2 Lamping–Gonthier graphs

Wadsworth's (1971) algorithm works with the λ-calculus in a similar fashion to the unfolding of jungles (§4.1). However, there is no way for Wadsworth's dags

to share terms which may differ in their subterms. This means that if a variable substitution is made within a subterm, by the rewriting of an abstraction, then the entire subterm must be duplicated in each position that it occurs, since the value that is substituted for the variable may differ between each position.

At POPL 17, Lamping (1990) first introduced an algorithm capable of performing optimal reduction in the sense of Lévy (1980). At this same conference, Lafont (1990) introduced his 'Interaction Nets', "a new kind of programming language" founded on Girard's (1989) *Geometry of Interaction*. Two years later, at POPL 19, Gonthier et al. (1992) fused these two concepts, refining Lamping's algorithm in the context of Lafont's new language. We retrace these steps, with a retrospective analysis of Lamping's optimal reduction algorithm, and of the motivations behind Gonthier's refinements thereof. The canonical reference work on Lévy-optimal sharing graphs is Asperti & Guerrini (1998), where more information on this subject may be sought.

4.2.1 Lamping fans

Lamping (1990) too uses backward binding arcs, *à la* Bourbaki notation (§3.2.2), in the term graphs of his sharing mechanism. Yet, due to the restrictions on structures whereby an arrow may be incident to at most two vertices, it is not possible on the face of it to use this method to represent non-linear functions, *i.e.* those abstractions which do not bind exactly one variable occurrence. The solution is to introduce a *contraction link*, a hyperarc incident to the abstraction's binding vertex as well as any number of vertices corresponding to occurrences of its bound variable.

Lamping's (1990) breakthrough was a technique to share not subterms but *contexts*, terms with an unspecified part, a *hole*. Hence, for any shared context there are several 'access pointers' to it and, for each hole, a set of possible choices to fill it — a choice for each instance of the context. As a result, Lamping's graphs used not only one sharing operator as described for jungles, but two: a *fan-in* collecting the pointers to a context, and a *fan-out* collecting the ways in which a hole may be filled (the exit pointers from the context). Assuming that each pointer collected by a fan-in or fan-out is associated to a named port, the

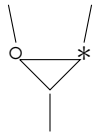


Figure 23: Lamping's fan operator

correspondence between entering and exiting pointers is kept provided there is a way to pair fan-ins and fan-outs: the instance of the context corresponding to the pointer connected to a fan-in port with name a fills its holes with the substructure accessed through the ports with name a of the matching fan-outs.

Lamping's (1990) fans, as well as some other links we consider later, have a natural orientation independent of that of the hyperarc, given by the cardinalities of the head and tail: they have a unique head arrow and a tail of arbitrary cardinality, or *vice versa*.

Definition 49. A *fan* is a backward or forward link with one principal port and two auxiliary ports. A fan is said to be a *fan-in* when it is backward, and a *fan-out* when it is forward.

A fan is drawn as illustrated in Figure 23.

A fan may *propagate* through another link, as illustrated in Figure 24. This is essentially the same process as for the simpler replicators; the only distinction is that fans may, as well as replicating links, loop back around and interact with another fan coming in the other direction. This is due to the characteristic loops of Bourbaki graphs, and more generally those that may occur in our higher-order abstract syntax graphs — note that the bottom edge in Figure 24 may, as before, represent multiple arrows, or indeed even the head and tail of the same arrow.

The two possible actions upon meeting another fan are *annihilation* and *swapping*. Intuitively, two fans annihilate one another if they originate from the same fan, having been duplicated through propagation; they swap, on the other hand, if they do not originate from the same fan. This notion of descendants mirrors the fans' role as delimiters of Lévy's redex families. The two actions, annihilation and swapping, are depicted in Figure 25.

Lamping's (1990) original algorithm was initially incomplete, as he pointed out himself: when two fans met, the algorithm could not determine which of two

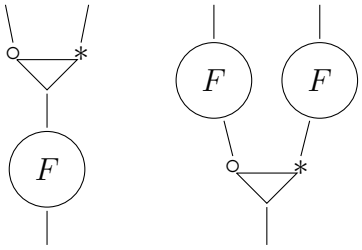


Figure 24: Fan propagation

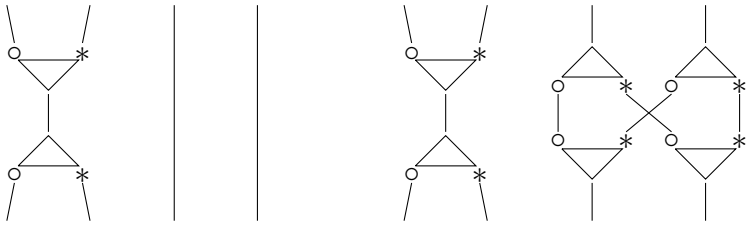


Figure 25: Fan annihilation and swapping

possible actions was the correct one. Although the correct course of action could always be determined through external analysis of its original and current states, the information contained within the sharing graphs themselves — as originally defined by Lamping — does not quite support the operations required for his sharing mechanism. With the solution to this problem not having yet presented itself, Lamping simply proposed a so-called 'oracle', which through some undisclosed means would determine the correct action to take. The proper formalisation of this oracle, and its relationship with linear logic (Girard 1987), was the real gem of Gonthier et al. (1992).

4.2.2 Gonthier operators

The key to Gonthier's implementation of the 'oracle' is the notion of Lamping's fans having *levels*; two fans at the same level will annihilate one another when they interact, whereas fans at differing levels will instead 'swap', passing through one another. The question, then, is how these levels are determined. This is the purpose of Gonthier's *control operators*: to adjust the levels of other operators, including Lamping's fans, so that they will interact with one another correctly, according to their provenance.

Definition 50. A *nesting levels assignment* for a hypergraph G is a map $\iota_G : V(G) \to \mathbb{N}$ assigning a non-negative *level* to each vertex of G.

Definition 51. An ℓ-structure over the link signature Σ is a triple (G, τ_G, ι_G), where (G, τ_G) is a structure of Σ, and ι_G is a nesting levels assignment for G.

The operators novel to Gonthier are *brackets*, which increment other links' levels, and *croissants*, which decrement them. However, in order to simplify the definitions of each of these operators, we will use Guerrini's (1996) generalised *multiplexers*, of which each of Lamping's and Gonthier's operators are a particular instantiation. The *triangles* of BOHM, the *Bologna Optimal Higher-order Machine* (Asperti & Guerrini 1998), are likewise a specialisation of Guerrini multiplexers, albeit a polymorphic one. We cover all of these operators in the next section.

4.2.3 Guerrini multiplexers

Guerrini (1996) generalises both Lamping's fans and Gonthier's control operators into a single sharing operator called the *multiplexer*. These multiplexers combine the levelling capabilities of brackets and croissants with the sharing capabilities of fans — as well as generalising the one or two auxiliary ports of these operators to an arbitrary (though fixed) n auxiliary ports.

Definition 52. A k-ary multiplexer, or *k-mux*, is a backward or forward link with:

1. one *principal port*;
2. a sequence of k secondary or *auxiliary ports*, the names of which $a_1, \ldots, a_k$ are chosen over a denumerable set with the proviso that $a_i = a_j$ iff $i = j$;
3. an associated non-negative integer m, called the *theshold* of e; and
4. an associated sequence of k integers $q_1, \ldots, q_k$, called the auxiliary port *offsets*, such that $\forall i \in \{1, \ldots, k\}.\ q_i \geq -1$.

A k-ary multiplexer is *positive* when it is backwards, and *negative* when it is forward.

Definition 53. There are three interactions involving multiplexers which we are interested in, called π-*interactions* ($\rightarrow_\pi$). These are defined as follows, and behave like the fan interactions already described, with the added subtlety of Gonthier's levels.

1. **Propagation** — the interaction of a k-mux e_γ with a second link $e_\star$ having h auxiliary doors (w.r.t. the redex). Its execution
   (a) creates k new instances of $e_\star$ and h copies of e_γ;
   (b) connects the j^{th} auxiliary port of the i^{th} instance $e_\star^i$ of the $\star$ link to the i^{th} port of the j^{th} copy e_γ^j of the mux; and
   (c) lifts the levels of all doors of $e_\star^i$ by the offset q_i of the i^{th} auxiliary port of e_γ.
2. **Annihilation** — the interaction between two k-ary muxes at the same level. The two multiplexers are removed from the structure, each auxiliary ports being linked to the corresponding port of the other.
3. **Swapping** — the interaction between two muxes with different thresholds. Consequently, the muxes must be swapped, in much the same way as a propagation. The thresholds of the new instances of the mux with the higher threshold are lifted by the offset of the auxiliary ports of the copies of the mux with the lower threshold to which they are connected.

Lamping's fans and Gonthier's operators are instantiations of Guerrini's multiplexers, with their attributes set as follows:

- A fan is a binary multiplexer whose auxiliary port offsets are both 0.
- A bracket is a unary multiplexer whose auxiliary port offset is 1.
- A croissant is a unary multiplexer whose auxiliary port offset is -1.
- An eraser is a nullary multiplexer.

These operators are drawn as illustrated in Figure 26.

Definition 54. An $s\ell$-structure is an ℓ-structure with the addition of Guerrini multiplexers.

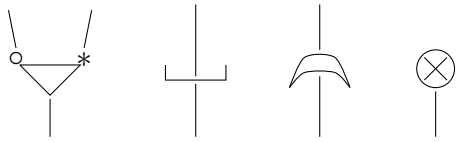


Figure 26: Fan, bracket, croissant, and eraser operators

Contraction links

Guerrini (1996) also introduces the *contraction link*, which differs from multiplexers in two distinct ways:

1. As Guerrini writes, "while the contraction links are only 'positive', the multiplexers may also be 'negative'." What this means is that a multiplexer may be a backward hyperarc ('positive'), as with contraction links, but may also be a forward hyperarc ('negative'). This is the key to Lamping's algorithm, as it is necessary for the 'dynamic' aspect of optimal sharing.

2. No distinction is made between the different tail vertices of a contraction link, whereas each auxiliary port of a multiplexer has its own name by which it may be distinguished. This is because, again, for Lamping's algorithm, the ports of a positive and negative multiplexer must line up if the two encounter each other.

Definition 55. A k-ary *contraction link* is a k-ary backward hyperarc in which:

1. all k premise ports have the same name; and

2. all the doors have the same level.

During a rewrite step involving its binding link, a contraction link is 'activated', triggering a substitution as illustrated in Figure 27.

Contraction links provide Guerrini's (1996) sharing graphs with a definition of *normal form*. A subclass of $s\ell$-structures called $u\ell$-structures — for 'unshared, unlabelled' — do not permit multiplexers of arity $n > 1$, so there may only be lifts (cf. "Erasers and garbage" below). Yet contraction links, which are used to contract numerous occurrences of a variable, may still be used, as they are static sharing

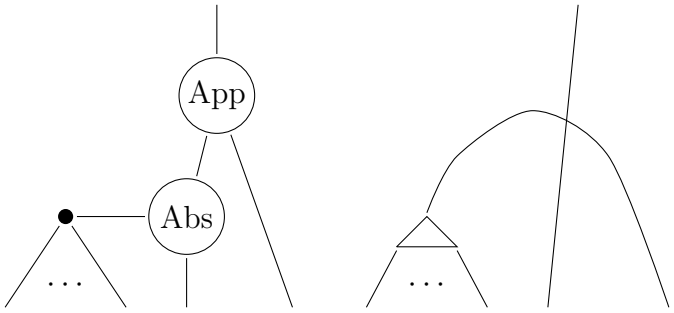


Figure 27: Activation of contraction link in $\mathrm{App}(\mathrm{Abs}(x.M), N) \to M[N/x]$

operators compared to the dynamic sharing provided by multiplexers. These structures may be obtained by π-reducing all sharing multiplexers, and have a normal form, which provides not only a representative structure for each π-equivalence class, but also provides the 'read-back' functionality needed to convert a graph back into a λ-term.

Definition 56. A $u\ell$-structure is an $s\ell$-structure whose signature includes contraction links, yet is free of Guerrini multiplexers. Each $s\ell$-structure has at least one $u\ell$-structure in its π-equivalence class, and the π-normal form of these serves as a normal form for the π-equivalence class.

Erasers and garbage

It is possible that theoretical problems with admitting weakening into sharing graphs (in the form of erasers) may yet be discovered, although in practice it does not seem that there are any such problems. After all, the inclusion of weakening 'eraser' operators is almost universal both in the literature and in concrete implementations, as they are necessary for the evaluation of the untyped λ-calculus. Nevertheless, Guerrini (1996) forbids nullary multiplexers in $s\ell$-structures, although they are present in BOHM, which is in part based on his work on $s\ell$-structures (Asperti & Guerrini 1998). To quote the appendix of his thesis, dealing with the behaviour of weakening and erasers:

> Even if no detailed study on the subject has been published yet, the relevant literature agrees that the problem should not have consequences for the soundness of the λ-calculus implementation. In spite of this, we

> prefer to not get involved in the study of a particular case, but to try to solve the problem more in general changing the shape of the weaning [*sic*] and of the eraser according to the work of Banach (1995). Namely, since the problem connected by the use of weakenings and erasers is the loss of the connectedness of the structure, we add to each weakening link has a port connecting it to a link which precedes it in a correct sequentialization of the net. In this way, in MELL a weakening component would be always connected to the main part of the boxes containing it, that is, to the part of the box which is a proof net by itself. According to this, the eraser should be transformed in a sort of lifts having a particular behavior on the logical nodes (erasing them).

To rely upon Banach's (1995) solution, as described by Guerrini, in a concrete implementation of $s\ell$-structures would however lead to unconstrained memory leakage, as 'eraser' operators are required for such an implementation to benefit from any form of garbage collection. An alternative is to use a notion of *operational equivalence* (Fernández & Mackie 1998), whereby a subset of the vertices in the graph's margin are specified as its 'observable interface', and any subgraph not connected to this interface is not considered for the purposes of equality. This reflects how, in practical terms, the portions of link graph through which an eraser has propagated are garbage, and have no further effect on the resulting computation.

4.3 Modal term graphs

We extend the earlier purely-linear term graphs (§3.3) to include the *modal* (i.e. non-purely-linear) expressions of the term calculus for Intuitionistic Linear Logic. As the name suggests, the key to $s\ell$-structures is in both *sharing* and *levelling*. We manage the sharing aspect — contraction and weakening of subgraphs — through fans and erasers, respectively, and manage the levelled aspect — dereliction and promotion of subgraphs — through croissants and brackets.

In addition to the term graphs we have already discussed, we may now annotate translations with levels, indicating that all nodes in the subgraph must be at

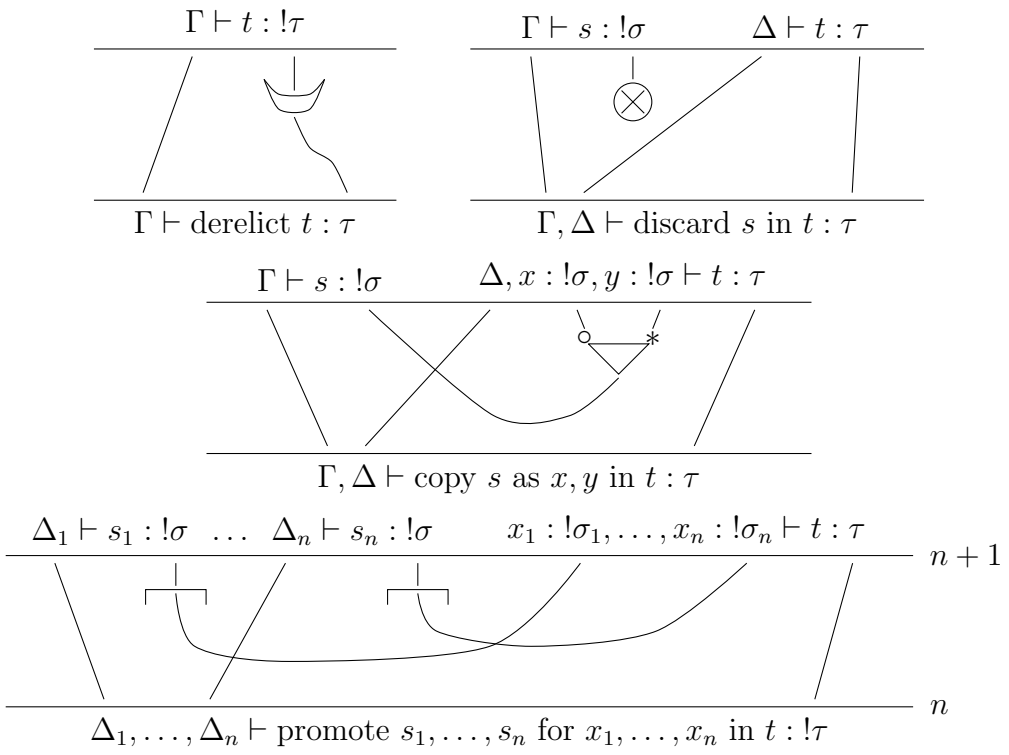


Figure 28: Translations from modal linear λ-terms to link graphs

a particular level. However, often we elide these annotations, as they may be inferred from context. Only once multiplexers have begun to propagate through one another are annotations truly necessary, for purposes of disambiguation.

4.3.1 Expressions

Figure 28 illustrates the inductive rules required for modal expressions. Dereliction makes use of a croissant, weakening an eraser, contraction a fan, and promotion a set of brackets. Additionally, a promotion $[\![\text{promote } s_1, \ldots, s_n \text{ for } x_1, \ldots, x_n \text{ in } t]\!]$ requires that the promoted subgraph $[\![t]\!]$ be one level higher than the resulting link graph. The promotion's arguments — $s_1, \ldots, s_n$ — however, are at the same level as the surrounding term.

Theorem 4. *Given a translated term $[\![\lambda\xi_1. \ldots \lambda\xi_n.t]\!]$, the vertices corresponding to some ξ_i are all at level 0.*

Proof. Only a promotion expression will increase a level, and only for its body, which must be closed. Therefore, in order for ξ_i to be used at a higher level of sharing, there must be an expression 'promote ξ_i for x_i in t', and in this case the vertices of $[\![\xi_i]\!]$ remain at level 0, albeit bracketed for use at the higher level. □

This property will be crucial when we come to translations of higher-order term rewriting systems (§5), as with a rule $\lambda\xi_1.\ldots\lambda\xi_n.t \to \lambda\xi_1.\ldots\lambda\xi_n.t'$, all vertices for each ξ_i must be at the same level in both $[\![t]\!]$ and $[\![t']\!]$.

Example 9. The term $\lambda x.!^0\text{copy } x \text{ as } y, z \text{ in } Fyz$ is translated into a term graph in Figure 29, which may alternatively be drawn as illustrated in Figure 30.

4.3.2 Contraction links

Before we can come to the question of equivalence of these modal term graphs, we need to consider how Guerrini's (1996) notion of contraction links, which are used to provide a normal form for sharing graphs, can be adapted to the higher-order case. As we will explain, the definition given by Guerrini is sufficient for second-order but not higher-order term graphs.

Orders of sharing

A first-order term rewriting system requires no sharing operators when translating its terms into a link graph, as its (first-order) abstract syntax tree is strictly a tree: there are no cycles, such as those used by Bourbaki graphs for variable occurrences, and no subterm is yet shared. It is only once reductions have taken place that, through non-right-linear rewrite rules, a replicator may be used to share multiple instances of a term. Even so, despite their newfound sharing, these derivative structures still have no cycles — they are directed acyclic graphs — and so this sharing may be represented by replicators alone.

Once we move up to second-order term rewriting systems, the presence of a variable will produce a cycle, by the fact that a variable occurrence is represented by an arrow arcing back to the variable's binding link. In the case of a contracting variable, this will need to be augmented by a contraction link, which performs

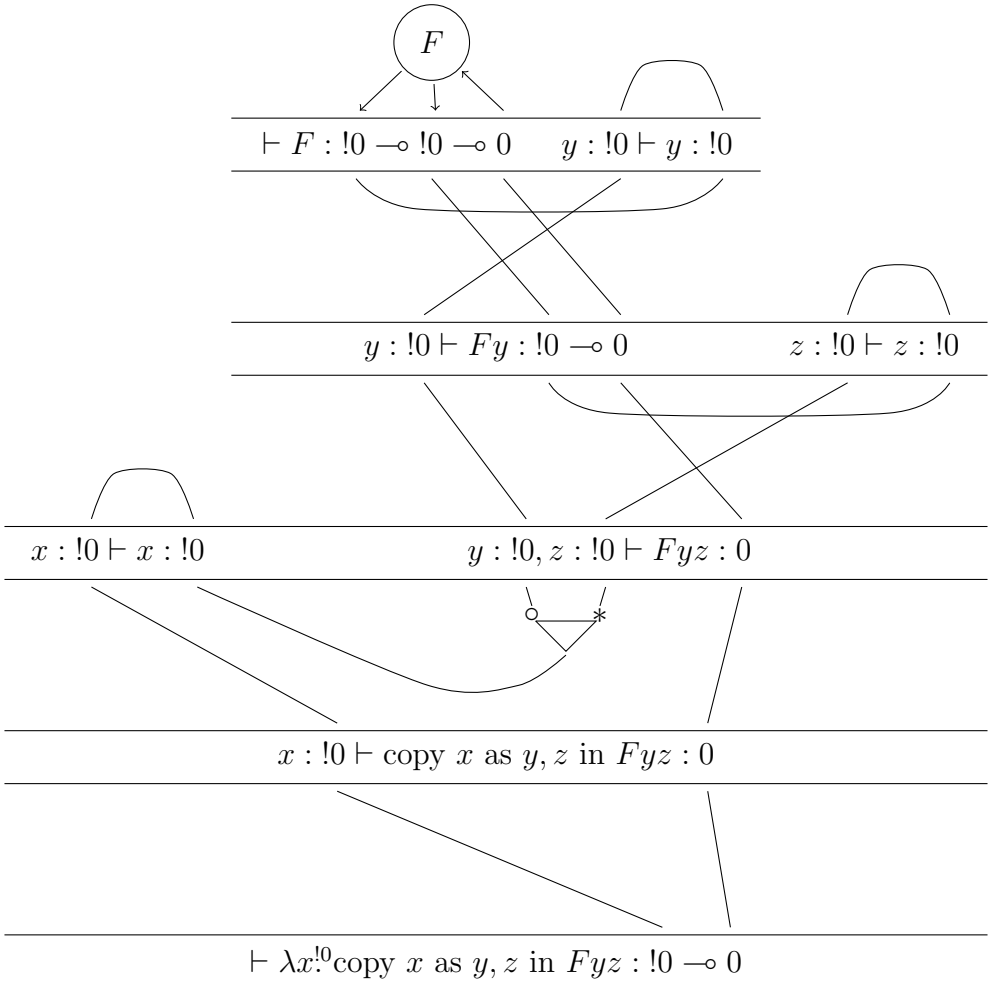


Figure 29: Construction of $\lambda x{:}^{!0}\text{copy } x \text{ as } y, z \text{ in } Fyz$

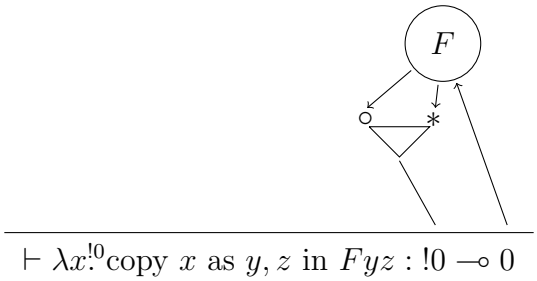


Figure 30: Alternative representation of $\lambda x{:}^{!0}\text{copy } x \text{ as } y, z \text{ in } Fyz$

essentially the same role as a replicator: it contracts, or replicates, the variable to each of its occurrences. Indeed, these two sharing operators — replicators and contraction links — are functionally interchangeable: a contraction link is merely a replicator for the specific case of a replicated variable occurrence. In either case, we need not distinguish between the tail ports of the link, as each occurrence of a shared term or variable is observationally equivalent to all other occurrences, by their 'shared image' (Guerrini 1996). However, once reductions have taken place, replicators or contraction links are no longer enough, and we must turn to the more sophisticated multiplexers to provide optimal sharing in the higher-order case.

With these sharing behaviours in mind, we can posit a notion of 'orders of sharing' in a sharing graph. A first-order term rewriting system begins without any sharing, and then as reductions take place may acquire replicators, from non-linearity on the right-hand side of rewrite rules, which introduces what we might call *first-order sharing*. Contrastingly, a second-order term rewriting system begins with first-order sharing, necessary for contracting its non-linear variable occurrences, and as reductions take place it may acquire multiplexers, from non-linear function application on the right-hand side, which introduces what we might call *higher-order sharing*.

Note that this notion of higher-orderedness in sharing is a semantic one, as opposed to syntactic, by which we mean that although the untyped λ-calculus is a second-order term rewriting system, when considered from the perspective of successive rewrite steps modulo substitution, its functions may, in a semantic sense, be of unbounded order. For instance, the following HRS term represents the untyped λ-term $(\lambda x.xx)(\lambda x.xx)$:

$$\text{App (Abs } (\lambda x^{0}.\text{App } x\ x)) \text{ (Abs } (\lambda x^{0}.\text{App } x\ x))$$

In the rewrite system for untyped β-reduction, this term has no normal form. Through successive rewrite steps its functions may be recursively applied to one another, continually and infinitely, even though within the substitution calculus itself they are of finite order. In this way, it is as if that term were recursively typed $\vdash (\lambda x.xx)(\lambda x.xx) : \mu\tau.(\tau \to \tau)$, which is of order ∞. So whilst first-order term rewriting can lead only to first-order sharing, as there are no functions in terms

at all, second-order term rewriting may lead to second- or higher-order sharing, depending on the orders of functions applied to one another during reduction, from an 'object level' perspective.

The reason this is important is that, whereas first-order terms begin with no sharing, when first translated into link graphs, and second-order terms begin with first-order sharing of their variable occurrences, third- and higher-order terms begin with higher-order sharing. This higher-order sharing is necessary in order to represent not only variable occurrences but also the terms to which those variables are applied, if they are not first-order: we must be able to associate each occurrence of a contracted variable with that occurrence's argument.

The consequence of this is that the specific distinctions made by Guerrini (1996) between multiplexers and contraction links are an artefact of the second-order case, and are not so useful as a distinct form of link in the third- and higher-order cases. The trouble being that, with a contraction link, each tail port has the same name, so they cannot be used to represent higher-order bound variables, whose occurrences must be associable with their respective arguments. Therefore, just as contraction links must be replaced with multiplexers during reduction in second-order rewriting, for third- and higher-order term graphs they are not sufficient even for the initial encoding of a term.

Contraction sets

In order to achieve sharing for higher-order term graphs without losing the desirable properties contraction links bestow upon $s\ell$-structures, we must generalise contraction links from the first-order form (i.e. equivalent to a replicator for a bound variable) to a higher-order one. To this end, we claim that contraction link as defined by Guerrini (1996) might be thought of as a contraction link *on type* 0, i.e. the base type, meaning it serves to contract the variables of a function of type $!0 \multimap \tau$. This makes sense in the context of our translations from terms to sharing graphs, which encode each atomic (base) type as a single vertex, and it is a single vertex for each variable occurrence that may be contracted by Guerrini's contraction link. What remains is to generalise this to the remaining type constructs required for Intuitionistic Linear Logic.

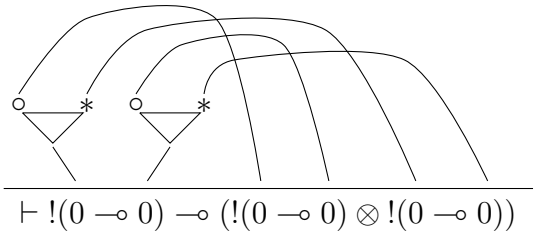


Figure 31: Sharing graph requiring paired contraction links

We begin with the multiplicative conjunction operator, '⊗'. A single contraction link, as defined, is unable to contract the variable occurrences in a function of type $!(0 \otimes 0) \multimap \tau$. The closest we could manage with this is to have two separate contraction links, one for the left-hand type and another for the right, as if the function were 'curried' to one of type $!0 \multimap !0 \multimap \tau$. Although this would work well enough in the multiplicative case — albeit not in the implicative case we have yet to broach — we would like to distinguish between the 'curried' and 'uncurried' graph forms, at least from a theoretical perspective. Consequently, we choose to 'pair up' two contraction links that are used in tandem to contract the left- and right-hand sides of a variable of a product type, in a sense treating the contraction links as two elements of a single 'contraction set'. A function of type $!(0 \otimes 0 \otimes 0) \multimap \tau$ would similarly have a triad of contraction links working in concert to contract each factor of the product type.

As for the linear implication operator, '⊸', things are made a little more complicated by the fact that the left- and right-hand sides of the implication are of opposite polarities, and so may be linked. By this we mean that two contraction links, one for an atomic type on the left and another on the right, may be activated as multiplexers which may then meet. If this happens then the auxiliary ports of the two multiplexers must be correctly associated with one another.

Example 10. Consider the following (β-normal) λ-term:

$$\vdash \lambda x.^{!(0 \multimap 0)}\text{copy } x \text{ as } y, z \text{ in } y \otimes z : !(0 \multimap 0) \multimap (!(0 \multimap 0) \otimes !(0 \multimap 0))$$

This is translated into the sharing graph in Figure 31, which requires a correct pairing of multiplexers, and the same is true of contraction links. An example

usage of this function would be,

$$(\lambda x.^{!(0\multimap 0)}\text{copy } x \text{ as } y, z \text{ in } y \otimes z)(\text{promote} - \text{for} - \text{in } \lambda x.^{0} x)$$

The correct normal form for this term is,

$$(\text{promote} - \text{for} - \text{in } \lambda x.^{0} x) \otimes (\text{promote} - \text{for} - \text{in } \lambda y.^{0} y)$$

However, an incorrect pairing of contraction links would result in illegal variable capture, with the left-hand function binding the variable in the body of the right, and vice versa.

Our solution, then, is to provide 'higher-order contraction links' with some limited notion of labelled tail vertices, greater than that provided to the ('first-order') contraction links of Guerrini (1996), but less than that of the dynamic multiplexers. In a similar spirit to the 'paired' contraction links for 'uncurried' functions, we initialise contraction links not individually but in sets, and do so with respect to some type σ. A contraction set over σ can then serve to contract the variable occurrences in a function of type $!\sigma \multimap \tau$. The key, then, is that each contraction set has its own *internal* labelling which keeps its links correctly paired yet does not have any external influence. The labelling of a contraction set is therefore arbitrary, and contraction links are equivalent modulo relabelling, just as with first-order contraction links, yet the relabelling of individual contraction links within the set does not preserve equivalence.

Definition 57. Given a set L of k distinct labels, a k-ary (higher-order) *contraction link* is a k-ary backward or forward hyperarc in which all k ports at its tail have the same level, each labelled with a pairwise-distinct label from L.

Note how these contraction links may be backward or forward hyperarcs, in contrast to the strictly backward hyperarcs for first-order contraction links.

Definition 58. A k-ary *contraction set* over a type σ is a set of k-ary contraction links, one per type atom occurrence in σ. A contraction set is equipped with an arbitrary set L of k distinct labels, which is used to label each of its contraction links. Contraction sets are equivalent modulo L.

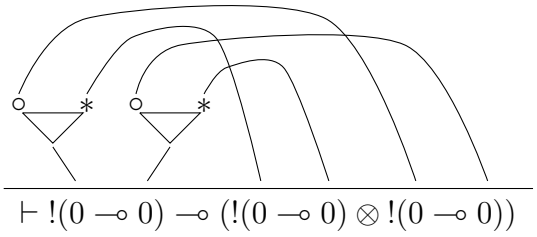


Figure 32: Alternative labelling for a contraction set

Example 11. The contraction set in Figure 31 could alternatively be labelled as in Figure 32, where the links' $\circ$ and $*$ ports have been switched by relabelling. However, one link could not be relabelled without the other being so too, as they both belong to the same contraction set and must be relabelled in the same way together, so that their respective tail arrows remain properly paired. These labels ($\circ$ and $*$) are arbitrary; other distinct labels from some set L may be used.

When, in our translation scheme, we construct Lamping fans over a set of edges, it may be understood as instantiating a binary contraction set over the given type. These contraction sets may be manipulated in much the same way as Guerrini's (1996) contraction links, with the proviso that the contraction links within each set remain correctly labelled with respect to one another. For instance, a two contraction sets of arity k_1 and k_2 over the same type τ may be merged into one contraction set of arity $k_1 + k_2$ over that type, by merging each corresponding contraction link, so long as the two have disjoint label sets.

Our definition of a contraction set over a type specialises to the case described for product types, exhibits the necessary behaviour in the case of linear implication, and is generalised trivially to exponential and unit types: for the former, a contraction set over $!\sigma$ is equal to that over σ; and for the latter, a contraction set over I is empty.

4.3.3 Equivalence

With higher-order contraction links at hand, we are now sufficiently well-equipped to prove the equivalence of sharing graphs with respect to the terms from which they are translated.

Theorem 5. $t \to t' \implies [\![t]\!] \to^* [\![t']\!]$

Proof. Case by case, the following graphs are π-equivalent, pairwise. We make reference to the six "safe rules" of Asperti & Guerrini (1998, p. 290). Referenced figures have been relegated to Appendix A.

1. derelict (promote $\vec{s}$ for $\vec{z}$ in t) $= t[\vec{s}/\vec{z}]$

   Figure 56 — In $[\![$derelict (promote $\vec{s}$ for $\vec{z}$ in $t)]\!]$, the croissant lowers the level of all links in $[\![t]\!]$, and then cancels out the bracket (by safe rule 2), resulting in a graph isomorphic with $[\![t[\vec{s}/\vec{z}]]\!]$.

2. promote t for x in derelict $x = t$

   Figure 57 — In $[\![$promote t for x in derelict $x]\!]$, the croissant cancels out the bracket (by safe rule 1), resulting in a graph isomorphic with $[\![t]\!]$.

3. discard (promote $\vec{s}$ for $\vec{z}$ in u) in $t =$ discard $\vec{s}$ in t

   Figure 58 — In $[\![$discard (promote $\vec{s}$ for $\vec{z}$ in u) in $t]\!]$, the eraser may propagate and erase both $[\![u]\!]$ and the bracket, resulting in a graph isomorphic with $[\![$discard $\vec{s}$ in $t]\!]$.

4. promote $u, \vec{s}$ for $y, \vec{z}$ in discard y in $t =$ discard u in promote $\vec{s}$ for $\vec{z}$ in t

   Figures 59 and 60 — In $[\![$promote $u, \vec{s}$ for $y, \vec{z}$ in discard y in $t]\!]$, the eraser may propagate and erase the bracket at $[\![u]\!]$, resulting in a graph isomorphic with $[\![$discard u in promote $\vec{s}$ for $\vec{z}$ in $t]\!]$.

5. copy s as x, y in discard x in $t = t[s/y]$

   Figure 61 — In $[\![$copy s as x, y in discard x in $t]\!]$, the eraser cancels out the fan (by safe rule 4), resulting in a graph isomorphic with $[\![t[s/y]]\!]$.

6. copy s as x, y in discard y in $t = t[s/x]$

   Figure 62 — In $[\![$copy s as x, y in discard y in $t]\!]$, the eraser cancels out the fan (by safe rule 5), resulting in a graph isomorphic with $[\![t[s/x]]\!]$.

7. copy s as x, y in $t =$ copy s as y, x in t

   Figure 63 — These graphs are equivalent by a relabelling of their respective contraction sets.

8. copy s as w, x in copy w as y, z in t = copy s as w, z in copy w as x, y in t

   Figures 64 and 65 — The two trees of fans are associative (by safe rule 6).

9. promote (promote $\vec{r}$ for $\vec{x}$ in u), $\vec{s}$ for $y, \vec{z}$ in t =
   promote $\vec{r}, \vec{s}$ for $\vec{y}, \vec{z}$ in t[promote $\vec{y}$ for $\vec{x}$ in u/y]

   Figures 66 and 67 — In $[\![$promote (promote $\vec{r}$ for $\vec{x}$ in u), $\vec{s}$ for $y, \vec{z}$ in $t]\!]$, a bracket at $[\![u]\!]$ may propagate through that term, lifting it one level higher, and then stack behind the bracket at $[\![s]\!]$, resulting in a graph isomorphic with $[\![$promote $\vec{r}, \vec{s}$ for $\vec{y}, \vec{z}$ in t[promote $\vec{y}$ for $\vec{x}$ in $u/y]]\!]$.

10. copy (promote $\vec{s}$ for $\vec{x}$ in u) as y, z in t =
    copy $\vec{s}$ as $\vec{y}, \vec{z}$ in t[promote $\vec{y}$ for $\vec{x}$ in u/y, promote $\vec{z}$ for $\vec{x}$ in u/z]

    Figures 68 and 69 — These graphs are isomorphic.

11. promote $u, \vec{s}$ for $y, \vec{z}$ in copy y as w, x in t =
    copy u as w, x in promote $w, x, \vec{s}$ for $w, x, \vec{z}$ in t

    Figures 70 and 71 — These graphs are equivalent following the propagation of the fan through $[\![u]\!]$.

12. t[discard s in u/x] = discard s in $t[u/x]$

    Figure 72 — These graphs are isomorphic.

13. t[copy s as y, z in u/x] = copy s as y, z in $t[u/x]$

    Figure 73 — These graphs are isomorphic.

□

Chapter 5

Rewriting

5.1 Linear HRSs

A *Linear Higher-order Rewrite System* is a higher-order term rewriting system in which the terms being rewritten, and so too the terms in the rules, are of the linear λ-calculus (§2.3.1), instead of the simply-typed λ-calculus as in Nipkow's (1991) HRSs. In the terminology of Van Oostrom (1994), a Linear HRS has the linear λ-calculus as its substitution calculus. As a result of this, terms are not simply-typed but rather linearly-typed, and this allows us to control the duplication and erasure of subterms during rewrite steps. Furthermore, these constraints are obtained 'for free', as the two sides of a rewrite rule must have the same type and type context.

The control we have over the duplication and erasure of subterms, as a result of the linear types of a Linear HRS, is much more powerful than if we were to attempt to apply those constraints to an HRS by way of the properties of rules. One may for instance require that a TRS be right-linear or non-erasing, or both, and may also apply this to individual variables, but with an HRS this is not enough. An HRS that is both right-linear and non-erasing may still duplicate or erase the terms corresponding to those variables during the course of the substitutive β-reduction. For example, the left- and right-linear, non-erasing HRS rule $\lambda\xi.\lambda\zeta.F\xi\zeta \to \xi\zeta$ will still duplicate ζ if ξ is a contractive function, or erase it if it is a weakening one. A Linear HRS, on the other hand, can enforce a restriction such that a variable which is not to be duplicated or erased cannot be used in any context where the resulting

β-reduction might possibly lead to the duplication or erasure of that term.

5.1.1 Patterns

We focus on a class of Linear HRSs whose left-hand sides are *purely-linear higher-order patterns*. As is the case for HRSs Nipkow's (1991), systems with more general left-hand sides are possible — all simply-typed λ-terms in the case of HRSs (i.e. HOTRSs), and all ILL terms in the case of Linear HRSs — but we restrict these for the sake of an effectively computable matching algorithm.

Definition 59. A linear λ-term l is a *purely-linear higher-order pattern* if it is a higher-order pattern and is purely linear. That is, it contains only expressions from the purely-linear fragment of the linear λ-calculus, i.e. without 'copy', 'discard', 'derelict', or 'promote' expressions.

Purely-linear higher-order patterns are therefore a subset of Miller's (1991) (simply-typed) higher-order patterns, except for the product and unit types and their related expressions, which are nevertheless occasionally included in definitions of simply-typed λ-calculi, and could easily be simulated in the latter by replacing the ⊗ operator with a constant. Consequently, there is an effective matching algorithm for these purely-linear pattern HRSs — it is just as though simply-typed λ-terms had been restricted to those with linear variable occurrences. We also

Note however that the 'purely linear' criterion is not strictly necessary for Linear HRSs; it simply happens to be a requirement for our work on optimality. This restriction also does not, of course, limit us from dealing with modal types altogether. For example, the following rule, that of β-reduction for the untyped λ-calculus, makes use of intuitionistic types but does not make use of any modal expression on the left-hand side.

$$\lambda\xi.^{!0\multimap 0}\lambda\zeta.^{!0}\mathrm{App}\ (\mathrm{Abs}\ \xi)\ \zeta \to \lambda\xi.^{!0\multimap 0}\lambda\zeta.^{!0}\xi\zeta$$

Further, modal expressions may occur on the right-hand side even if they may not

on the left, as in the following Linear HRS.

$$\lambda\xi.^{!(0\multimap 0)}\lambda\zeta.^{0}\text{Maybe } \xi \text{ (Some } \zeta) \to \lambda\xi.^{!(0\multimap 0)}\lambda\zeta.^{0}\text{Some ((derelict } \xi)\zeta)$$
$$\lambda\xi.^{!(0\multimap 0)}\text{Maybe } \xi \text{ None} \to \lambda\xi.^{!(0\multimap 0)}\text{discard } \xi \text{ in None}$$

The purely-linear criterion also takes care of variable containment (Kahrs 1996): a variable cannot occur on the right-hand side unless it occurs on the left. A rewrite rule $\lambda\vec{x}.t \to \lambda\vec{x}.t'$ has $\mathit{Var}(t) = \vec{x}$, since these variables cannot be weakened, and so $\mathit{Var}(t) \supseteq \mathit{Var}(t')$ — as well as left-linearity, since these variables cannot be contracted in t either. This would not be true if we permitted modal expressions on the left-hand side, as a rule like $\lambda x.^{!0}\text{discard } x \text{ in } A \to \lambda x.^{!0}x$ would be able to pluck a value for x 'out of thin air', and so could rewrite A to any other term.

Definition 60. A Linear HRS rule is a pair of closed linear λ-terms $\vdash l : \tau$ and $\vdash r : \tau$, written $l \to r$, where l is a purely-linear higher-order pattern. A rule ρ induces a relation $\forall C[].\ C[l] \to_\rho C[r]$ modulo equality, i.e. $t \to_\rho t'$ where $t = C[l]$ and $t' = C[r]$.

A Linear HRS comprises a set of rules inducing a union relation $\bigcup\{\to_\rho \mid \rho \in R\}$.

Owing to their similarity, we can apply same the definition of overlappingness for HRSs to Linear HRSs: two patterns overlap if they are unifiable. However, two patterns that would overlap in an HRS do not necessarily overlap in a Linear HRS, if linear variables are involved.

Example 12. The following Linear HRS is orthogonal; $\hat{\theta}_1(\xi x \otimes \zeta)$ and $\hat{\theta}_2(\xi \otimes \zeta x)$ cannot be unified because x is linear and so cannot be weakened: it must occur in one or other of the multiplicative subterms.

$$\lambda\xi.^{0\multimap 0}\lambda\zeta.^{0}F(\lambda x.^{0}(\xi x \otimes \zeta)) \to \lambda\xi.^{0\multimap 0}\lambda\zeta.^{0}C(\xi\zeta)$$
$$\lambda\xi.^{0}\lambda\zeta.^{0\multimap 0}F(\lambda x.^{0}(\xi \otimes \zeta x)) \to \lambda\xi.^{0}\lambda\zeta.^{0\multimap 0}D(\zeta\xi)$$

In an 'intuitionistic' HRS, on the other hand, the two terms would be unifiable to a term in which the bound variable x occurs in neither subterm.

Corollary 1. *Since any Linear HRS is left-linear, non-overlappingness implies orthogonality, and thereby confluence.*

5.1.2 Extendedness

One subtlety in higher-order rewriting that is not shared by first-order rewriting is that, whereas first-order rewriting can only create redexes in those contexts with which it overlaps, in the higher-order case the weakening of a bound variable can cause a redex to appear well outside the context of the rewrite step that caused that weakening to take place (Terese 2003). Every term in an orthogonal TRS that is not in normal form has an *external redex*, a redex which is 'persistently outermost' and so must need to be reduced in order for the term to be normalised. Intuitionistic HRSs, on the other hand, do not share this property in general, because the weakening of a bound variable can cause a redex to appear where there previously was none.

Example 13. Consider the following (non-linear) HRS, where $F : (0 \to 0) \to 0$ and $G : 0 \to 0$.

$$\lambda\xi^0.F(\lambda x.\xi) \to \lambda\xi^0.C\xi$$
$$\lambda\zeta^0.G\zeta \to \lambda\zeta^0.A$$

The term $F(\lambda x.G(Gx))$ contains two redexes, one for each symbol G; the outermost term, with F at its head, is not a redex since x occurs in its body, which the pattern $\lambda\xi.F(\lambda x.\xi)$ disallows. Yet if the inner redex is reduced, i.e. $F(\lambda x.G(\underline{Gx})) \to F(\lambda x.G\underline{B})$, then the outermost term suddenly becomes a redex by the weakening of x, despite not having any overlap with the term that was reduced. This means that the original term does not have any external redex.

In order to avoid this complication, *full extendedness* (Van Oostrom 1996) has been used to ensure that all variables are permitted in a subterm, and so the weakening of these variables cannot cause a redex to become active 'from a distance'. We propose a more general terminology when dealing with this phenomenon, specifically in higher-order rewriting systems with substructural substitution calculi, which we call *under-extendedness*.

Definition 61. A parameter in a pattern is *under-extended* if a bound variable is in scope which it does not 'extend over' — that is, which does not occur as one of the bound variables to which it is applied — and which may be weakened. A pattern is under-extended if any of its parameters are under-extended.

In normal HRSs, any parameter that is not fully-extended is therefore under-extended, as any bound variable may be weakened. This is not necessarily true in the case of Linear HRSs, as linear bound variables cannot be weakened. Speaking of substructural HRSs in general, an interesting pattern emerges in how bound variables may be used, and how this affects the extendedness and orderedness of matching in that system.

Variable occurrence checks are often ruled out, generally because the guarantee of external redexes drastically simplifies needed reductions, by requiring that all rules be fully-extended: every free variable is applied to every bound variable in scope, so that no variable occurrence check needs to be made and the weakening of a variable cannot cause a redex to lose its status as an outermost redex. Matching in an HRS which is fully-extended is equivalent to first-order matching, as no occurrence checks are required for bound variables.

However, this does not need to be enforced for linear variables in Linear HRSs, because a linear variable simply cannot weaken. A variable check may still be required to determine whether a term is a redex, as can be seen in Example 12, where the bound variable's being on the left- or the right-hand side of the product determines whether the term rewrites to $C(\xi\zeta)$ or $D(\zeta\xi)$, and so a variable check must be made to determine on which of the two sides the bound variable occurs. Yet, because the variable cannot *weaken*, the system's outermost redexes are, and will remain, external.

Theorem 6. *An HRS is* weakly-extended *if each free variable is applied to every weakening bound variable in scope (if any). Every Linear HRS is weakly-extended. Any term in a weakly-extended orthogonal HRS that is not in normal form has an external redex.*

If an HRS is both fully-extended and orthogonal then it is *local*. The matching process of such a system is equivalent to that of first-order TRSs, as there is no need for variable occurrence checks.

5.2 Sharing graph rewriting

Any HRS may be rewritten in the form of a sharing graph, a naïve strategy being to π-normalise the graph between each rewrite step. However, this will perform a full unsharing, which is clearly suboptimal in the sense of Lévy (1980). In fact, it is equivalent in its reduction behaviour to simply rewriting between β-normal forms, as is common in higher-order term rewriting. Rewriting a sharing graph whilst preserving the sharing provided by its multiplexers requires rewriting w.r.t. Guerrini's (1996) *sharing morphisms*, i.e. modulo π-equivalence. This amounts to somehow finding a sharing morphism that isolates a needed redex family to a single shared instance in the sharing graph, rewriting it, and then moving on to the next needed redex family. In the case of sequential systems, such a strategy of rewriting sharing graphs modulo π-equivalence would be optimal. However, this approach is clearly rather abstract, and is not an effective strategy that might be applied to an arbitrary HRS. Yet, by applying restrictions on the classes of HRS permitted, we can achieve an effective and optimal reduction strategy.

There are three classes of HRS that are particularly relevant, in decreasing order of size. In this chapter we describe effective optimal reduction strategies for each of these classes, which in the coming sections we shall introduce in reverse order, since, naturally, the smaller the class the easier it is to account for.

1. Match-sequential HRSs (§5.4.2) — Higher-order rewriting systems for which the pattern matching process is sequential. These translate into $s\ell$-structures for which we provide an effective optimal reduction strategy, but which are not Interaction Nets.

2. Non-rigid match-sequential HRSs (§5.3.3) — Match-sequential HRSs whose left-hand sides do not contain any rigid bound variable (that is not a parameter). These systems may be encoded into the next class of HRS by way of a signature morphism, or may be evaluated as is according to the same strategy as the previous class.

3. Higher-order Interaction Systems (§5.2.2) — Non-rigid match-sequential systems for which each left-hand side has only one sequential index. These

translate directly into Lafont's (1990) Interaction Nets, for which an optimal reduction strategy is already well understood.

We prove the optimality of the largest of these classes, and by extension all of its subclasses, towards the end of this chapter.

The fundamental basis of all of these systems is the notion of sequentiality (§2.1.2), albeit generalised to the higher-order case. We will consequently explore this higher-order generalisation before describing the various term rewriting systems. The distinction between the latter two systems is that patterns in the first of the two cannot contain variable occurrences in rigid position. A relaxing of this constraint yields, as a side-effect, further nuances with regards to higher-order sequentiality (§5.4.3).

5.2.1 Higher-order sequentiality

In order to achieve this we can define sequentiality not with respect to term contexts, but rather in term of the link graphs by which we represent higher-order terms. Sequentiality in link graphs thereby generalises first-order sequentiality in term contexts to the higher-order case.

Definition 62. Let f be a monotonic function on graphs (on $\leq$). Vertex v in a graph G is an *index* of f in G iff $v \in \partial_E(G)$, and $\forall H$ such that $G \leq H$ then $f(G) < f(H)$ implies that $v \notin \partial_E(G)$. f is said to be *sequential* iff whenever $\exists H$ such that $G \leq H$ then $f(G) < f(H)$ implies there exists an index of f in G. Given a Linear HRS R, the *prefixes* of R is defined as the set $\{C \mid \exists (l, r) \in R.\, C \sqsubseteq \omega([\![l]\!])\}$.

Essentially, whereas a TRS is sequential iff its tree structure may be explored in a sequential fashion, an HRS is sequential iff its *graph* structure — in the sense of higher-order abstract syntax as described earlier — may be explored in a sequential fashion. As the left-hand sides of Linear HRSs are purely linear, these sequential prefixes do not contain any multiplexers or other sharing structures — only interconnected links and their border vertices.

Example 14. Recall the Linear HRS for β-reduction in the untyped λ-calculus:

$$\lambda\xi.^{!0\multimap 0}\lambda\zeta.^{!0}\mathrm{App}\ (\mathrm{Abs}\ \xi)\ \zeta \to \xi\zeta$$

This system is match sequential, with the following normal prefixes:

$$\begin{aligned} \text{App} &: \underline{0} \multimap !0 \multimap 0 \\ \text{App}\ \Box^{\underline{0}} &: !0 \multimap 0 \\ \text{App}\ \Box^{\underline{0}}\ \Box^{!0} &: 0 \\ \text{Abs} &: (!0 \multimap 0) \multimap \underline{0} \\ \text{Abs}\ \Box^{!0 \multimap 0} &: \underline{0} \end{aligned}$$

Note how only those holes whose (lone) type atom is underlined need be inspected in order to determine a match.

In any case where there are more than one type atom, and one is not a sequential index, we can use $\beta\eta$-equivalence to isolate that type atom. The basic instances of this are the following:

$$\begin{aligned} \Box^{\sigma \multimap \tau} &\to (\lambda x^{\sigma}.\Box^{\tau}) \\ \Box^{\sigma \otimes \tau} &\to \Box^{\sigma} \otimes \Box^{\tau} \end{aligned}$$

There is an additional transformation which allows us to unfold a term not only from the context downwards, but also from a bound variable upwards, thus isolating the left-hand side of a linear implication. We will come to this transformation later (§5.4.3).

Variable occurrence checks

Match sequentiality in higher-order rewriting requires that the matching process not depend upon a variable occurrence check, unless we might make use of an auxiliary mechanism (an 'oracle') to perform variable occurrence checks without expanding context indices. We will explore this possibility later (§5.4), but lacking such a mechanism, we make the following proposition.

Proposition 1. A sequential HRS is fully extended.

This would not necessarily be true if the set of function symbols that might occur in that context, i.e. of the appropriate type, were all nullary or unary. However, we do not presume that the system's signature could be constrained in this

way, as we feel that the property then becomes a little too 'fragile', e.g. breaking on minor changes such as signature extension.

Proof. Suppose we have an HRS rule $F(\lambda x^0.\xi) \to \xi$; we begin with the trivial context $\Box$, and inspect $\Box$ to yield $F(\lambda x^0.\Box)$. At this point we do not know if the property holds, as x may or may not occur. If we inspect $\Box$ again and this results in $F(\lambda x.P\Box\Box)$, then we now have two holes, both of which may or may not contain x. If either does contain such an occurrence then checking the other term was unnecessary, but we cannot know beforehand which of them does or doesn't. Thus, the context is not sequential, and so the system is not sequential.

This may occur in any circumstance where a variable is not extended over, by likewise positing that the corresponding context's hole is filled by a term containing a function symbol of arity $n > 1$. Thus, any system that is not fully-extended is non-sequential. □

5.2.2 Higher-order Interaction Systems

Higher-order Interaction Systems are a higher-order generalisation of second-order Interaction Systems (Laneve 1993).

Definition 63. A Higher-order Interaction System is a match-sequential Linear HRS whose patterns comprise exactly two symbols, joined by a single sequential index atom — i.e. the conclusion of one is used as a premise for the other.

Lemma 3. *Each symbol must have exactly one sequential index atom in its type.*

Proof. Follows from match sequentiality: given an n-ary symbol F, for a context $F\Box_1 \dots \Box_n$, there must be an index $\Box_i$ to inspect first. As there is only ever a single other constant in a pattern, this must be the only index. □

Lemma 4. *Each rule must feature a unique pair of symbols.*

Proof. Follows from orthogonality: as each symbol has a single sequential index, the two constants must always be in the same arrangement, so two rules with the same two would overlap. □

The reason for defining Higher-order Interaction Systems as we have is in order to provide a straightforward translation from Higher-order Rewrite Systems to Interaction Nets (§3.1.3), for which optimal rewriting strategy is known (Asperti & Guerrini 1998).

Theorem 7. *The graph translation (see §3.3, §4.3) of a Higher-order Interaction System is an Interaction Net.*

Proof. By Lemmas 3 and 4, each translated link has a distinguished principal port corresponding to a sequential index, through which all interactions with other links will take place. This principal port has a set direction, according to the polarity of the type atom occurrence corresponding to its index. □

An IS symbol arity $(k_1, \ldots, k_n)$, where $k_i \in \mathbb{N}$, corresponds to an (intuitionistic) type $\sigma_1 \to \cdots \to \sigma_n \to 0$, where each σ_i is a type $0 \to \cdots \to 0 \to 0$ comprising k_i+1 type atoms. However, we focus on *linear* HISs, wherein by way of the 'Girard translation' the IS arity $(k_1, \ldots, k_n)$ is translated to $\sigma_i \multimap \cdots \multimap \sigma_n \multimap 0$ where $\sigma_i = !(!0 \multimap \cdots \multimap !0 \multimap 0)$, except for the first argument of a destructor σ_1 which is translated (for Interaction Net encoding reasons) to 0.

Given that existing algorithms for optimal reduction of term rewriting systems boil down to Interaction Systems, it is interesting to compare those systems with our Higher-order Interaction Systems. Aside from the obvious generalisation from second to higher order, Laneve (1993) places several restrictions on Interaction Systems that we do not apply to Higher-order Interaction Systems, essentially due to more expressive a type system. We will now explore two of these.

Multiplicative types

Given any IS symbol $F : (k_1, \ldots, k_n)$, a term $F(\vec{x}_1.t_1, \ldots, \vec{x}_n.t_n)$ must then be of an atomic type. For example, Asperti & Guerrini (1998) discuss "the case of *append*," in which they describe an Interaction Net agent 'Dec' (for 'de-cons'), featured in their implementation BOHM. The 'Dec' agent has two input ports and an output port; when it interacts with a 'Cons' agent, the two input ports of the former are joined with the outputs of the latter. This provides one independent access to the head and tail of a linked list without having to duplicate it and so manage it with

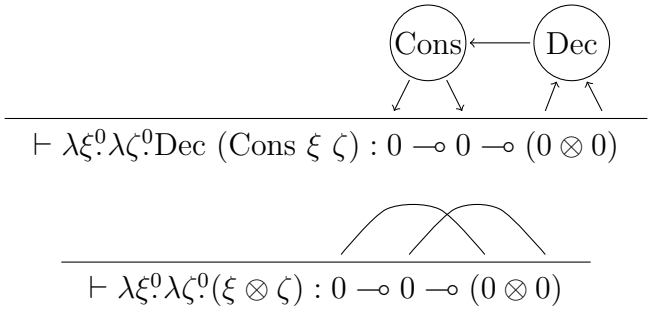


Figure 33: Interaction Nets for Dec–Cons HIS rule

sharing operators, as one would have to if one had only the separate destructors 'Head' and 'Tail'.

Yet, because the symbol does not have a single input port corresponding to its context — rather, it has two — this agent cannot be described in terms of an IS rule. In contrast, an HIS can represent such a symbol by way of a multiplicative type, in this case $\mathrm{Dec} : 0 \multimap (0 \otimes 0)$, and define the Interaction Net rule by way of an HIS rule:

$$\lambda\xi^0.\lambda\zeta^0.\mathrm{Dec}\ (\mathrm{Cons}\ \xi\ \zeta) \to \lambda\xi^0.\lambda\zeta^0.(\xi \otimes \zeta)$$

This rule is illustrated as an Interaction Net in Figure 33. Note that the multiplicative operator $\otimes$ is a part of the syntax of the substitution calculus itself, and unlike 'Cons' is not a constant, so there is no agent on the right-hand side of the translated graph rewrite rule.

There is however an implicit restriction on multiplicative types in Higher-order Interaction Systems, due to the requirement that the constructor and destructor be connected by a single sequential type atom. This constraint has the effect of forbidding rules where the same constant occupies both sides of a product type, as in the following example.

Example 15. The following ruleset is not a legal Higher-order Interaction System.

$$\begin{aligned} \lambda\xi^0.F(A \otimes \xi) &\to r_1 \\ \lambda\xi^0.F(\xi \otimes B) &\to r_2 \\ F(\text{let } C \text{ be } x \otimes y \text{ in } x \otimes y) &\to r_3 \end{aligned}$$

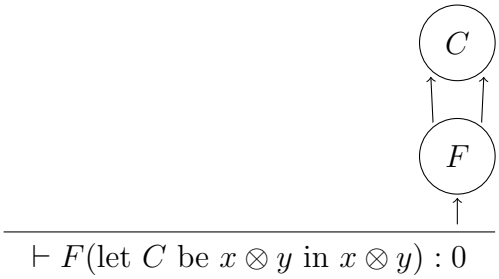


Figure 34: Parallel sequential type atoms between a pair of links

Note that the left-hand side of the third rule could also be written FC. But in any case, this pattern would require both atoms in the type of $C : 0 \otimes 0$ to be sequential, which is not permitted in an HIS. These parallel sequential indices are illustrated in Figure 34.

This restriction is an artefact of HISs' translation to Interaction Nets, which must have exactly one principal port. However, we may remove this constraint when working with sharing graphs that are not Interaction Nets, and the result will still achieve optimal reduction (§5.4).

Binding over constructors

Any IS destructor must be of arity $(0, \ldots)$, so that every pattern is of the form $\mathbf{d}(\mathbf{c}(\ldots), \ldots)$, i.e. with no variables bound by $\mathbf{d}$ over $\mathbf{c}$'s subterms. The reason for this restriction is that if such a binder were allowed then a rule could be written such as,

$$\mathbf{d}(x.\mathbf{c}(M, N)) \to \mathbf{g}(M[x/\mathbf{a}], N[x/\mathbf{b}])$$

Here, two different terms ($\mathbf{a}$ and $\mathbf{b}$) are substituted for x, but we cannot actually distinguish between variables occurring in one or other of the subterms corresponding to meta-variables M and N when rewriting in an Interaction Net. Instead, the occurrences in both must be substituted by the same term on the right-hand side.

However, this poses no problem for HISs, since modal expressions cannot occur on the left-hand sides of rules, as would be needed to write a direct translation of

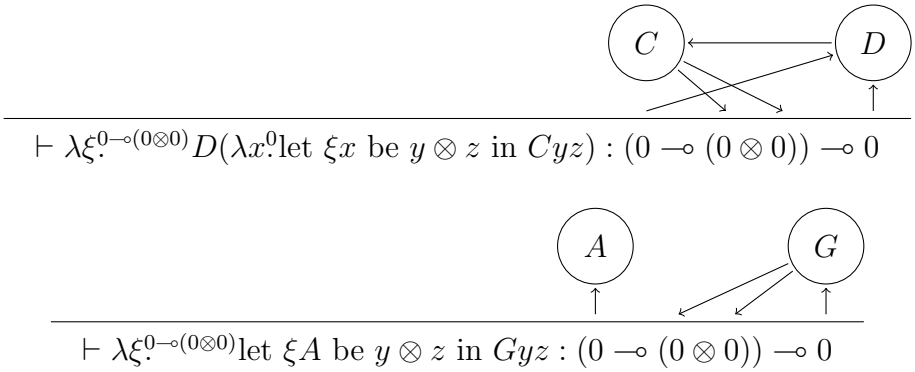


Figure 35: Interaction Nets for HIS rule with binder over constructor

the above rule:

$$\lambda\xi{:}^{!0\multimap 0}\lambda\zeta{:}^{!0\multimap 0}D(\lambda x{:}^{!0}\text{copy } x \text{ as } y, z \text{ in } C(\xi y)(\zeta z)) \to \lambda\xi{:}^{!0\multimap 0}\lambda\zeta{:}^{!0\multimap 0}G(\xi A)(\zeta B)$$

The only way to write a functionally equivalent left-hand side in an HIS, then, is to constrain the rule such that only one term may possibly be substituted for x on the right-hand side.

For example, in this case we substitute for x the constant A:

$$\lambda\xi{:}^{0\multimap(0\otimes 0)}D(\lambda x{:}^{!0}\text{let } \xi x \text{ be } y \otimes z \text{ in } Cyz) \to \lambda\xi{:}^{0\multimap(0\otimes 0)}\text{let } \xi A \text{ be } y \otimes z \text{ in } Gyz$$

We know here that x may occur in either of the arguments to C — and if x were linear that it must occur exactly once in exactly one of them — but we do not know which. Despite not knowing where the variable occurrence (or occurrences) are, we can still substitute for it a single term, wherever it happens to be. This demonstrates the usefulness of $\otimes$-types in Linear HRSs, and how they contribute crucial additional expressiveness.

Our approach to this problem may be contrasted to *hierarchical graph rewriting systems* (Fernández, Mackie & Pinto 2007), in which one may delineate a particular subgraph on the left-hand side of a rewrite rule, such as those corresponding to M and N in the original invalid IS, and thereby capture any arrows at their border. One could then substitute different terms on each meta-variable on the right-hand side. However, such systems may be incompatible with Lamping–Gonthier

sharing graphs. At any rate, it could be said that whereas Fernández et al. began with Combinatory Reduction Systems and designed hierarchical graph rewriting systems as a translation target, we have begun with Interaction Nets and designed Higher-order Interaction Systems as a corresponding translation source.

5.3 Interaction System translations

Having defined match sequentiality and Higher-order Interaction Systems, we will now introduce a number of effective algorithms for performing optimal reduction. The fundamental basis for this is a translation from a subclass of match-sequential Higher-order Rewrite Systems to Higher-order Interaction Systems, and from there a method of generalising this mechanism to the full class of match-sequential Higher-order Rewrite Systems. The most complicated hurdle in these strategies is, as is to be expected, the presence of bound variables; we therefore begin with a simplified algorithm for the first-order case, then generalise it to the higher-order case without rigid bound variables.

Later on we will move to the higher-order case with rigid bound variables (§5.4). However, these systems go beyond what can be defined in terms of Interaction Nets, and so cannot be translated into Interaction Systems. For these systems we must therefore handle the rewriting of such systems with an alternative to the translation approach described in this section.

5.3.1 Translating TRSs to DISs

Fernández & Mackie (1996) describe an algorithm for translating match-sequential constructor TRSs into Discrete Interaction Systems. This can be generalised to match-sequential TRSs in general by way of a signature morphism, as any orthogonal TRS can be converted into a constructor TRS (Kahrs & Smith 2014).[1] However, to convert an orthogonal TRS into a constructor TRS and thence into an Interaction System can, we feel, obscure some intuitiveness of the technique.

[1] This technique is also used in Kahrs & Smith (2016) to reduce the class of non-ω-overlapping TRSs to non-ω-overlapping constructor TRSs.

Building on the work of Fernández & Mackie, we present a generalised translation for all match-sequential orthogonal TRSs, which need not be constructor TRSs. We use the sequential contexts used in the definition of match sequentiality as actual function symbols in the signature of a Discrete Interaction System (DIS). This results in an intuitive translation in which the DIS's rewrite steps correspond to steps in the TRS's sequential redex matching procedure.

To translate TRSs we use the concept of signature morphism — see Sannella & Tarlecki (2012) for a more general and modern version of the concept; we specialise it here for the standard signatures in TRSs, as this concept rarely shows up in term rewriting literature.

Definition 64. A signature morphism between signatures Σ and Θ is a function $f : \Sigma \to \Theta$ such that $\#f(G) = \#G$. Each signature morphism $f : \Sigma \to \Theta$ induces a map $T_f : \mathit{Ter}(\Sigma) \to \mathit{Ter}(\Theta)$ as follows:

$$T_f(F(t_1, \ldots, t_n)) = f(F)(T_f(t_1), \ldots, T_f(t_n))$$
$$T_f(x) = x$$

Given a match-sequential TRS $T = (\Sigma, R)$, we may apply a signature morphism $f : \Sigma \to \Sigma \times \overline{\Sigma}$, where $\overline{\Sigma}$ is the set of match-sequential prefixes of R. This signature will be used as the signature of an Interaction System, where symbols in the original signature Σ will be used as constructors, and those in the prefix signature $\overline{\Sigma}$ as destructors.

The 'lifting' operation $\lceil \cdot \rceil$ is a map $\mathit{Ter}(\Sigma) \to \mathit{Ter}(\Sigma \times \overline{\Sigma})$, defined as follows:

$$\lceil F(t_1, \ldots, t_n) \rceil = \overline{\Box}(F(\lceil t_1 \rceil, \ldots, \lceil t_n \rceil))$$
$$\lceil x \rceil = \overline{\Box}(x)$$

This essentially 'primes' the matching process; each destructor $\overline{\Box}$ may be viewed as a position at which the matching procedure may yet find a redex, which it does by progressively inspecting its indices and incorporating them into itself before performing a step that is equivalent to a rewrite step in the source TRS. Thus $\overline{\Box}$ is the starting point for any potential match, a trivial context which is trivially match-sequential for any non-empty ruleset $R \neq \emptyset$.

This set-up may be compared to Fernández & Mackie's (1996) special-case destructor d_f, which is introduced to handle the case of a lone and spontaneously-rewriting TRS symbol f: a rule $f(x_1, \ldots, x_n) \to r$ becomes $d_f(f(x_1, \ldots, x_n)) \to r'$. In our system, $\overline{\Box}$ serves as the initial 'spark' for every rule, not only that of lone symbols. This can be brought closer to the algorithm of Fernández & Mackie, more clearly identifiable as a projection of each function symbol in Σ to a destructor or constructor, as appropriate, by way of an 'optimisation' step,

$$\lceil F(t_1, \ldots, t_n) \rceil = \begin{cases} \overline{\Box}(F(\lceil t_1 \rceil, \ldots, \lceil t_n \rceil)) & \text{if } F(\Box, \ldots, \Box) \to t' \\ \overline{F(\Box, \ldots, \Box)}(\lceil t_1 \rceil, \ldots, \lceil t_n \rceil) & \text{if } \overline{F(\Box, \ldots, \Box)} \in \overline{\Sigma} \\ F(\lceil t_1 \rceil, \ldots, \lceil t_n \rceil) & \text{otherwise} \end{cases}$$

However, this is unnecessary from a theoretical perspective.

For each match-sequential prefix symbol $\overline{C} \in \overline{\Sigma}$ and each function symbol $F \in \Sigma$ we generate an IS rewrite rule. In the following we assume that the sequential index of C is its first hole. There are three possible cases, depending on the status of the context $C' = C[F(\Box, \ldots, \Box), \Box, \ldots, \Box]$ as a prefix of R.

1. Match success: if C' is a redex, i.e. $C[F(x_1, \ldots, x_k), x_{k+1}, \ldots, x_n] \to r$, the term is rewritten to the right-hand side, though lifted to $\lceil r \rceil$ to allow for further rewriting steps to occur.

$$\overline{C}(F(x_1, \ldots, x_k), x_{k+1}, \ldots, x_n) \to \lceil r \rceil$$

2. No match yet: if C' is a strict prefix of R, the context consumes the symbol F, and will go on to inspect another subterm.

$$\overline{C}(F(x_1, \ldots, x_k), x_{k+1}, \ldots, x_n) \to \overline{C[F(\Box, \ldots, \Box), \Box, \ldots, \Box]}(x_1, \ldots, x_n)$$

3. Match failure: otherwise, C' is neither a prefix of R nor a redex; the term is replaced with the 'literal' filling of the context, as matching has failed and C may form part of the constructor component of some outer redex.

$$\overline{C}(F(x_1, \ldots, x_k), x_{k+1}, \ldots, x_n) \to C[F(x_1, \ldots, x_k), x_{k+1}, \ldots, x_n]$$

Example 16. The following TRS, due to Huet and Lévy (1991),

$$\begin{aligned} F(A, x) &\to r_1 \\ F(G(E, x), B) &\to r_2 \\ F(G(x, E), C) &\to r_3 \end{aligned}$$

Translates into the DIS:

$$\begin{aligned} \overline{\Box_1}(F(x, y)) &\to \overline{F(\Box_1, \Box_2)}(x, y) \\ \overline{F(\Box_1, \Box_2)}(A, x) &\to \lceil r_1 \rceil \\ \overline{F(\Box_1, \Box_2)}(G(x, y), z) &\to \overline{F(G(\Box_2, \Box_3), \Box_1)}(z, x, y) \\ \overline{F(G(\Box_2, \Box_3), \Box_1)}(B, x, y) &\to \overline{F(G(\Box_1, \Box_2), B)}(x, y) \\ \overline{F(G(\Box_1, \Box_2), B)}(E, x) &\to \lceil r_2 \rceil \\ \overline{F(G(\Box_2, \Box_3), \Box_1)}(C, x, y) &\to \overline{F(G(\Box_2, \Box_1), C)}(y, x) \\ \overline{F(G(\Box_2, \Box_1), C)}(E, x) &\to \lceil r_3 \rceil \end{aligned}$$

Plus various rules of the form e.g. $\overline{F(G(\Box_2, \Box_3), \Box_1)}(E, x, y) \to F(G(x, y), E)$, which act as match failure conditions.

Theorem 8. *Index reduction of the resulting IS simulates index reduction of the original TRS.*

Proof. Any needed TRS term $C[t_1, \ldots, t_n]$ maps to a needed IS term $\overline{C}(t'_1, \ldots, t'_n)$, where the reduction step then follows the exact procedure of searching for a redex in the appropriate index of C, as described by the index reduction algorithm for (match-sequential) orthogonal TRSs (Huet & Lévy 1991). □

Fernández & Mackie (1996) do not explicitly handle the second-order case as necessary for Interaction Systems proper, but the restrictions required of ISs make the extension trivial: no symbol may occur under a binder on the left-hand side, only a meta-variable. This ensures that we cannot have a prefix C whose sequential index binds a variable, in which case the corresponding IS symbol, $\overline{C} : (k_1, \ldots, k_n)$ where $k_1 > 0$, would not be a valid destructor. In addition, variables may not

occur on the left-hand side — except as arguments to a meta-variable — as these occurrences cannot be translated into Interaction Systems.

5.3.2 Encoding linear λ-contexts

We can translate Linear HRSs into Higher-order Interaction Systems, with certain constraints as before, by again introducing sequential contexts into its signature. Embedding contexts into a TRS signature was simple enough because TRS symbols only have arities, which for contexts is just the number of holes they contain. HRSs, on the other hand, have fully-fledged higher-order types, and so in order to embed contexts into the signature we have to assign each context symbol a linear type which properly encapsulates how many variables are bound by abstractions over each hole, bearing in mind that some holes may in fact be under the same abstraction.

Each type τ has its own corresponding hole symbol $\Box^\tau$, but we write simply $\Box$ when the type is not important or can be inferred from context. Each symbol $\overline{C} : \rho \multimap \tau$ corresponds to the context $C : \tau$, where ρ represents the types of all of the holes required to complete C into a term. This is extended even to contexts with no hole (i.e. a whole term), so that all take an initial first argument. For example, $A : 0$ has a corresponding context symbol $\overline{A} : I \multimap 0$, which may occur in a term $\overline{A}*$. This is not unlike the first-order case, where such a symbol would be of nullary type, taking in effect an empty tuple as its arguments with which to fill its (zero) holes. For instance, a first-order term $A()$ has a corresponding nullary context symbol $\overline{A()}$, which may then occur in a term $\overline{A()}()$.

Typing rules for Linear HRS context symbols, limited to the purely-linear fragment of the linear λ-calculus, are given in Figure 36. We extend this embedding, of contexts on Σ to symbols in $\overline{\Sigma}$, to the filling of those contexts' holes, written e.g. $\overline{C[t]}$. Rules for this hole-filling in the purely-linear fragment of the linear λ-calculus are given in Figure 37.

Theorem 9. $C[t_1, \ldots, t_n] \in \mathit{Ter}(\Sigma) \implies \overline{C[t_1, \ldots, t_n]} \in \mathit{Ter}(\overline{\Sigma})$

Proof. If $C[t_1, \ldots, t_n] \in \mathit{Ter}(\Sigma)$ then $\overline{C} \in \overline{\Sigma}$, and for any $\overline{C} \in \overline{\Sigma}$ a corresponding case $\overline{C[t_1, \ldots, t_n]} \in \mathit{Ter}(\overline{\Sigma})$ is given in Figure 37. □

$$\frac{}{\vdash \overline{\Box^{\tau}} : \tau \multimap \tau}\ \Box$$

$$\frac{F : \tau \in \Sigma}{\vdash \overline{F} : I \multimap \tau}\ \mathit{Sym}$$

$$\frac{}{x : \sigma \vdash \overline{x} : I \multimap \sigma}\ \mathit{Var}$$

$$\frac{\Gamma, x : \sigma \vdash \overline{C} : \rho \multimap \tau}{\Gamma \vdash \overline{\lambda x^{\sigma} C} : (I \multimap \rho) \multimap (\sigma \multimap \tau)}\ \multimap_{I_1}$$

$$\frac{\Gamma \vdash \overline{C} : \rho \multimap \tau \quad C \notin \mathit{Ter}(\Sigma)}{\Gamma \vdash \overline{\lambda x^{\sigma} C} : (\sigma \multimap \rho) \multimap (\sigma \multimap \tau)}\ \multimap_{I_2}$$

$$\frac{\Gamma_1 \vdash \overline{C_1} : \rho_1 \multimap (\sigma \multimap \tau) \quad \Gamma_2 \vdash \overline{C_2} : \rho_2 \multimap \sigma}{\Gamma_1, \Gamma_2 \vdash \overline{C_1 C_2} : (\rho_1 \otimes \rho_2) \multimap \tau}\ \multimap_{E}$$

$$\frac{}{\vdash \overline{*} : I \multimap I}\ I_I$$

$$\frac{\Gamma \vdash \overline{C} : \rho_1 \multimap I \quad \Delta \vdash \overline{D} : \rho_2 \multimap \tau}{\Gamma, \Delta \vdash \overline{\text{let } C \text{ be } * \text{ in } D} : (\rho_1 \otimes \rho_2) \multimap \tau}\ I_E$$

$$\frac{\Gamma \vdash \overline{C_1} : \rho_1 \multimap \tau_1 \quad \Delta \vdash \overline{C_2} : \rho_2 \multimap \tau_2}{\Gamma, \Delta \vdash \overline{C_1 \otimes C_2} : (\rho_1 \otimes \rho_2) \multimap (\tau_1 \otimes \tau_2)}\ \otimes_I$$

$$\frac{\Gamma \vdash \overline{C} : \rho_1 \multimap (\sigma_1 \otimes \sigma_2) \quad \Delta, x : \sigma_1, y : \sigma_2 \vdash \overline{D} : \rho_2 \multimap \tau}{\Gamma, \Delta \vdash \overline{\text{let } C \text{ be } x \otimes y \text{ in } D} : (\rho_1 \otimes ((I \otimes I) \multimap \rho_2)) \multimap \tau}\ \otimes_{E_1}$$

$$\frac{\Gamma \vdash \overline{C} : \rho_1 \multimap (\sigma_1 \otimes \sigma_2) \quad \Delta, y : \sigma_2 \vdash \overline{D} : \rho_2 \multimap \tau \quad D \notin \mathit{Ter}(\Sigma)}{\Gamma, \Delta \vdash \overline{\text{let } C \text{ be } x \otimes y \text{ in } D} : (\rho_1 \otimes ((\sigma_1 \otimes I) \multimap \rho_2)) \multimap \tau}\ \otimes_{E_2}$$

$$\frac{\Gamma \vdash \overline{C} : \rho_1 \multimap (\sigma_1 \otimes \sigma_2) \quad \Delta, x : \sigma_1 \vdash \overline{D} : \rho_2 \multimap \tau \quad D \notin \mathit{Ter}(\Sigma)}{\Gamma, \Delta \vdash \overline{\text{let } C \text{ be } x \otimes y \text{ in } D} : (\rho_1 \otimes ((I \otimes \sigma_2) \multimap \rho_2)) \multimap \tau}\ \otimes_{E_3}$$

$$\frac{\Gamma \vdash \overline{C} : \rho_1 \multimap (\sigma_1 \otimes \sigma_2) \quad \Delta \vdash \overline{D} : \rho_2 \multimap \tau \quad D \notin \mathit{Ter}(\Sigma)}{\Gamma, \Delta \vdash \overline{\text{let } C \text{ be } x \otimes y \text{ in } D} : (\rho_1 \otimes ((\sigma_1 \otimes \sigma_2) \multimap \rho_2)) \multimap \tau}\ \otimes_{E_4}$$

Figure 36: Rules for Linear HRS context symbols

$$\frac{}{\overline{\Box[t]} = \overline{\Box}t}\ \Box$$

$$\frac{}{\overline{F[]} = \overline{F}*}\ Sym$$

$$\frac{}{\overline{x[]} = \overline{x}*}\ Var$$

$$\frac{\overline{C[t_1,\ldots,t_n]} = \overline{C}u \quad x \notin Var(u)}{\overline{(\lambda x^{\sigma} C)[t_1,\ldots,t_n]} = \overline{(\lambda x^{\sigma} C)}(\lambda x.\text{let } x \text{ be } * \text{ in } u)}\ \multimap_{I_1}$$

$$\frac{\overline{C[t_1,\ldots,t_n]} = \overline{C}u \quad x \in Var(u)}{\overline{(\lambda x^{\sigma} C)[t_1,\ldots,t_n]} = \overline{(\lambda x^{\sigma} C)}(\lambda x.u)}\ \multimap_{I_2}$$

$$\frac{\overline{C_1[t_1,\ldots,t_k]} = \overline{C_1}u_1 \quad \overline{C_2[t_{k+1},\ldots,t_n]} = \overline{C_2}u_2}{\overline{(C_1 C_2)[t_1,\ldots,t_n]} = \overline{(C_1 C_2)}(u_1 \otimes u_2)}\ \multimap_{E}$$

$$\frac{}{\overline{*[]} = \overline{*}*}\ I_I$$

$$\frac{\overline{C[t_1,\ldots,t_k]} = \overline{C}u \quad \overline{C[t_{k+1},\ldots,t_n]} = \overline{D}v}{\overline{(\text{let } C \text{ be } * \text{ in } D)[t_1,\ldots,t_n]} = \overline{\text{let } C \text{ be } * \text{ in } D}(u \otimes v)}\ I_E$$

$$\frac{\overline{C_1[t_1,\ldots,t_k]} = \overline{C_1}u_1 \quad \overline{C_2[t_{k+1},\ldots,t_n]} = \overline{C_2}u_2}{\overline{(C_1 \otimes C_2)[t_1,\ldots,t_n]} = \overline{C_1 \otimes C_2}(u_1 \otimes u_2)}\ \otimes_I$$

$$\frac{\overline{C[t_1,\ldots,t_k]} = \overline{C}u \quad \overline{D[t_{k+1},\ldots,t_n]} = \overline{D}v \quad x \notin Var(v) \quad y \notin Var(v)}{\overline{(\text{let } C \text{ be } x \otimes y \text{ in } D)[t_1,\ldots,t_n]} = \overline{\text{let } C \text{ be } x \otimes y \text{ in } D}(u \otimes \lambda z.\text{let } z \text{ be } * \otimes * \text{ in } v)}\ \otimes_{E_1}$$

$$\frac{\overline{C[t_1,\ldots,t_k]} = \overline{C}u \quad \overline{D[t_{k+1},\ldots,t_n]} = \overline{D}v \quad x \in Var(v) \quad y \notin Var(v)}{\overline{(\text{let } C \text{ be } x \otimes y \text{ in } D)[t_1,\ldots,t_n]} = \overline{\text{let } C \text{ be } x \otimes y \text{ in } D}(u \otimes \lambda z.\text{let } z \text{ be } x \otimes * \text{ in } v)}\ \otimes_{E_2}$$

$$\frac{\overline{C[t_1,\ldots,t_k]} = \overline{C}u \quad \overline{D[t_{k+1},\ldots,t_n]} = \overline{D}v \quad x \notin Var(v) \quad y \in Var(v)}{\overline{(\text{let } C \text{ be } x \otimes y \text{ in } D)[t_1,\ldots,t_n]} = \overline{\text{let } C \text{ be } x \otimes y \text{ in } D}(u \otimes \lambda z.\text{let } z \text{ be } * \otimes y \text{ in } v)}\ \otimes_{E_3}$$

$$\frac{\overline{C[t_1,\ldots,t_k]} = \overline{C}u \quad \overline{D[t_{k+1},\ldots,t_n]} = \overline{D}v \quad x \in Var(v) \quad y \in Var(v)}{\overline{(\text{let } C \text{ be } x \otimes y \text{ in } D)[t_1,\ldots,t_n]} = \overline{\text{let } C \text{ be } x \otimes y \text{ in } D}(u \otimes \lambda z.\text{let } z \text{ be } x \otimes y \text{ in } v)}\ \otimes_{E_4}$$

Figure 37: Rules for filling HRS context symbols

Example 17. The symbol for the context $F(\lambda x^{\rho}.G\Box^{\sigma}\Box^{\tau})$, given a signature Σ including $F : (\rho \multimap 0) \multimap 0$ and $G : \sigma \multimap \tau \multimap 0$, is constructed as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \overline{G} : I \multimap \sigma \multimap \tau \multimap 0} \quad \dfrac{}{\vdash \overline{\Box} : \sigma \multimap \sigma}}{\vdash \overline{G\Box} : (I \otimes \sigma) \multimap \tau \multimap 0} \quad \dfrac{}{\vdash \overline{\Box} : \tau \multimap \tau}}{\vdash \overline{G\Box\Box} : (I \otimes \sigma \otimes \tau) \multimap 0}}{\vdash \overline{\lambda x.G\Box\Box} : (\rho \multimap (I \otimes \sigma \otimes \tau)) \multimap \rho \multimap 0} \quad \dfrac{}{\vdash \overline{F} : I \multimap (\rho \multimap 0) \multimap 0}}{\vdash \overline{F(\lambda x.G\Box\Box)} : (I \otimes (\rho \multimap (I \otimes \sigma \otimes \tau))) \multimap 0}$$

The term $\overline{F(\lambda x.G\Box\Box)}(* \otimes \lambda x.(* \otimes Ax \otimes B))$ then represents the filling of this context, $F(\lambda x.G\Box\Box)[Ax, B]$.

Note here that the variable used within the constant is essentially arbitrary, and has no bearing on the binding of a term within its argument. For example, the previous may just as well be written $\overline{F(\lambda x.G\Box\Box)}(* \otimes \lambda y.(* \otimes Ay \otimes B))$. We do not concern ourselves with variable 'shadowing', e.g. $\lambda x^{\rho}.\lambda x^{\sigma}.\Box^{\tau}$, which would be typed as $\overline{\lambda x^{\rho}.\lambda x^{\sigma}.\Box^{\tau}} : (\rho \multimap \sigma \multimap \tau) \multimap \tau$ and so allow the capture of a variable by the outer abstraction when the hole is filled, even though this would not be possible without α-renaming, since the occurrence of x in $\lambda x.\lambda x.x$ is bound not to the outer abstraction but the inner one. This is not practically a problem, as any valid context may still be translated into a valid symbol, even if the reverse is not necessarily true.

One rule in need of comment is $\multimap_{I_2}$, which closes over holes with an abstraction. This allows us to introduce an abstraction whose variable does not occur in the context, but rather may fill one of its holes. The purpose of the side condition $C \notin Ter(\Sigma)$ is so that we do not end up with illegal symbols like $\overline{G(\lambda x^{0}.A)} : (I \otimes (0 \multimap I)) \multimap 0$, where $\vdash G(\lambda x^{0}.A) : 0$ is not a valid context (or term) because the linear variable x does not occur within its binding abstraction. Consider also that such a context could not be filled: the linear type $0 \multimap I$ is uninhabited, and so $f : 0 \multimap I \vdash \overline{(G(\lambda x^{0}.A))}(* \otimes f) : 0$ cannot be fulfilled for any legal term f (barring a constant). The side condition therefore means that C must include a hole, as it is not a complete term, so that it can be filled and permit the bound variable to occur.

We tie this closure to the abstraction itself rather than e.g. allowing a hole to

occur with a non-empty type context:

$$\frac{}{x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash \Box^\tau : (\sigma_1 \otimes \ldots \otimes \sigma_n) \multimap \tau}\ \Box'$$

So that we cannot derive multiple alternative typings for a context $\lambda x^!\Box^{\sigma\multimap\tau}\Box^\sigma$, which with this alternative ruleset would have several possible typings, such as,

$$\dfrac{\dfrac{\dfrac{}{x : \rho \vdash \overline{\Box^{\sigma\multimap\tau}} : (\rho \multimap \sigma \multimap \tau) \multimap (\sigma \multimap \tau)}\ \Box' \quad \dfrac{}{\vdash \overline{\Box^\sigma} : \sigma \multimap \sigma}\ \Box'}{x : \rho \vdash \overline{\Box^{\sigma\multimap\tau}\Box^\sigma} : ((\rho \multimap \sigma \multimap \tau) \otimes \sigma) \multimap \tau}\ \multimap_E}{\vdash \overline{\lambda x^!\Box^{\sigma\multimap\tau}\Box^\sigma} : ((\rho \multimap \sigma \multimap \tau) \otimes \sigma) \multimap (\rho \multimap \tau)}\ \multimap_{I_1}$$

and,

$$\dfrac{\dfrac{\dfrac{}{\vdash \overline{\Box^{\sigma\multimap\tau}} : (\sigma \multimap \tau) \multimap (\sigma \multimap \tau)}\ \Box' \quad \dfrac{}{x : \rho \vdash \overline{\Box^\sigma} : (\rho \multimap \sigma) \multimap \sigma}\ \Box'}{x : \rho \vdash \overline{\Box^{\sigma\multimap\tau}\Box^\sigma} : ((\sigma \multimap \tau) \otimes (\rho \multimap \sigma)) \multimap \tau}\ \multimap_E}{\vdash \overline{\lambda x^!\Box^{\sigma\multimap\tau}\Box^\sigma} : ((\sigma \multimap \tau) \otimes (\rho \multimap \sigma)) \multimap (\rho \multimap \tau)}\ \multimap_{I_1}$$

whereas with the rule $\multimap_{I_2}$ as described above, the only possible typing for this term is the following:

$$\dfrac{\dfrac{\dfrac{}{\vdash \overline{\Box^{\sigma\multimap\tau}} : (\sigma \multimap \tau) \multimap (\sigma \multimap \tau)}\ \Box \quad \dfrac{}{\vdash \overline{\Box^\sigma} : \sigma \multimap \sigma}\ \Box}{\vdash \overline{\Box^{\sigma\multimap\tau}\Box^\sigma} : ((\sigma \multimap \tau) \otimes \sigma) \multimap \tau}\ \multimap_E}{\vdash \overline{\lambda x^!\Box^{\sigma\multimap\tau}\Box^\sigma} : (\rho \multimap ((\sigma \multimap \tau) \otimes \sigma)) \multimap (\rho \multimap \tau)}\ \multimap_{I_2}$$

The distinction here is that the structure of a context does not dictate which variables may occur in which holes, except to the extent that a variable may only occur within a hole over which it is bound.

Note that whereas in the first-order case we had $\overline{C_1} \equiv \overline{C_2}$ iff $C_1 \equiv C_2$, this is not strong enough a property in the case of the λ-calculus. In this case we equate two symbols if they are equal, i.e. in the same βη-equivalence class: $\overline{C_1} \equiv \overline{C_2}$ iff $C_1 = C_2$. For example, we have that $\overline{(\lambda x.x)\Box} \equiv \overline{\Box}$.

Theorem 10. *For each symbol* $\Gamma \vdash \overline{C} : \rho \multimap \tau$ *there exists a function* $\Gamma \vdash t : \rho \multimap \tau$ *which, if substituted for* $\overline{C}$ *in* $\overline{C[u_1, \ldots, u_n]}$, *would reduce to* $C[u_1, \ldots, u_n]$.

Proof. Case by case:

1. $\vdash \overline{\square} \mapsto \lambda z.z : \tau \multimap \tau$
2. $F : \tau \in \Sigma \implies \vdash \overline{F} \mapsto \lambda z.(\text{let } z \text{ be } * \text{ in } F) : I \multimap \tau$
3. $x : \sigma \vdash \overline{x} \mapsto \lambda z.(\text{let } z \text{ be } * \text{ in } x) : I \multimap \sigma$
4. $\Gamma, x : \sigma \vdash \overline{C} : \rho \multimap \tau \implies \Gamma \vdash \overline{\lambda x^{\sigma} C} \mapsto \lambda z.\lambda x.\overline{C}(z*) : (I \multimap \rho) \multimap (\sigma \multimap \tau)$
5. $\Gamma \vdash \overline{C} : \rho \multimap \tau \implies \Gamma \vdash \overline{\lambda x^{\sigma} C} \mapsto \lambda z.\lambda x.\overline{C}(zx) : (\sigma \multimap \rho) \multimap (\sigma \multimap \tau)$
6. $\Gamma_1 \vdash \overline{C_1} : \rho_1 \multimap (\sigma \multimap \tau) \wedge \Gamma_2 \vdash \overline{C_2} : \rho_2 \multimap \sigma \implies$
   $\Gamma_1, \Gamma_2 \vdash \overline{C_1 C_2} \mapsto \lambda z.(\text{let } z \text{ be } z_1 \otimes z_2 \text{ in } \overline{C_1} z_1 (\overline{C_2} z_2)) : (\rho_1 \otimes \rho_2) \multimap \tau$
7. $\vdash \overline{*} \mapsto \lambda z.(\text{let } z \text{ be } * \text{ in } *) : I \multimap I$
8. $\Gamma \vdash \overline{C} : \rho_1 \multimap I \wedge \Delta \vdash \overline{D} : \rho_2 \multimap \tau \implies$
   $\Gamma, \Delta \vdash \overline{\text{let } C \text{ be } * \text{ in } D} \mapsto \lambda z.(\text{let } z \text{ be } z_1 \otimes z_2 \text{ in let } \overline{C} z_1 \text{ be } * \text{ in } \overline{D} z_2) :$
   $(\rho_1 \otimes \rho_2) \multimap \tau$
9. $\Gamma \vdash \overline{C} : \rho_1 \multimap \sigma \wedge \Delta \vdash \overline{D} : \rho_2 \multimap \tau \implies$
   $\Gamma, \Delta \vdash \overline{C \otimes D} \mapsto \lambda z.(\text{let } z \text{ be } z_1 \otimes z_2 \text{ in } \overline{C} z_1 \otimes \overline{D} z_2) : (\rho_1 \otimes \rho_2) \multimap (\sigma \otimes \tau)$
10. $\Gamma \vdash \overline{C} : \rho_1 \multimap (\sigma_1 \otimes \sigma_2) \wedge \Delta, x : \sigma_1, y : \sigma_2 \vdash \overline{D} : \rho_2 \multimap \tau \implies$
    $\Gamma, \Delta \vdash \overline{\text{let } C \text{ be } x \otimes y \text{ in } D} \mapsto$
    $\lambda z.(\text{let } z \text{ be } z_1 \otimes z_2 \text{ in let } \overline{C} z_1 \text{ be } x \otimes y \text{ in } \overline{D}(z_2(* \otimes *))) :$
    $(\rho_1 \otimes ((I \otimes I) \multimap \rho_2)) \multimap \tau$
11. $\Gamma \vdash \overline{C} : \rho_1 \multimap (\sigma_1 \otimes \sigma_2) \wedge \Delta, y : \sigma_2 \vdash \overline{D} : \rho_2 \multimap \tau \implies$
    $\Gamma, \Delta \vdash \overline{\text{let } C \text{ be } x \otimes y \text{ in } D} \mapsto$
    $\lambda z.(\text{let } z \text{ be } z_1 \otimes z_2 \text{ in let } \overline{C} z_1 \text{ be } x \otimes y \text{ in } \overline{D}(z_2(x \otimes *))) :$
    $(\rho_1 \otimes ((\sigma_1 \otimes I) \multimap \rho_2)) \multimap \tau$
12. $\Gamma \vdash \overline{C} : \rho_1 \multimap (\sigma_1 \otimes \sigma_2) \wedge \Delta, x : \sigma_1 \vdash \overline{D} : \rho_2 \multimap \tau \implies$
    $\Gamma, \Delta \vdash \overline{\text{let } C \text{ be } x \otimes y \text{ in } D} \mapsto$
    $\lambda z.(\text{let } z \text{ be } z_1 \otimes z_2 \text{ in let } \overline{C} z_1 \text{ be } x \otimes y \text{ in } \overline{D}(z_2(* \otimes y))) :$
    $(\rho_1 \otimes ((I \otimes \sigma_2) \multimap \rho_2)) \multimap \tau$

13. $\Gamma \vdash \overline{C} : \rho_1 \multimap (\sigma_1 \otimes \sigma_2) \wedge \Delta \vdash \overline{D} : \rho_2 \multimap \tau \implies$
$\Gamma, \Delta \vdash \overline{\text{let } C \text{ be } x \otimes y \text{ in } D} \mapsto$
$\lambda z.(\text{let } z \text{ be } z_1 \otimes z_2 \text{ in let } \overline{C} z_1 \text{ be } x \otimes y \text{ in } \overline{D}(z_2(x \otimes y))) :$
$(\rho_1 \otimes ((\sigma_1 \otimes \sigma_2) \multimap \rho_2)) \multimap \tau$

This demonstrates the correctness of the context embedding. □

5.3.3 Translating HRSs to HISs

Now that we have our signature morphism for embedding of λ-term contexts into an HRS signature, the process is much the same as that described for first-order TRSs (§5.3.1). In this section we assume that the source Linear HRS is match-sequential and contains no rigid variable occurrences, as Higher-order Interaction Systems do not permit patterns having rigid variable occurrences (cf. §5.4).

Given a match-sequential HRS $\mathcal{H} = (\Sigma, R)$, we may apply a signature morphism $f : \Sigma \to \Sigma \times \overline{\Sigma}$, where Σ is the set of match-sequential prefixes of R. This signature will be used as the signature of a Higher-order Interaction System, where symbols in the original signature Σ will be used as constructors, and those in the prefix signature $\overline{\Sigma}$ as destructors.

The 'lifting' operation $\lceil \cdot \rceil$ is a map $\mathit{Ter}(\Sigma) \to \mathit{Ter}(\Sigma \times \overline{\Sigma})$, defined as follows. Note that we only add the trivial context destructor $\overline{\Box}$ over other constants. We must also η-expand the constant to pass in the necessary variables for matching.

$$
\begin{aligned}
\lceil x \rceil &= x \\
\lceil F \rceil &= \lambda x_1 . \ldots \lambda x_n . \overline{\Box}(F x_1 \ldots x_n) \\
\lceil \lambda x.t \rceil &= \lambda x . \lceil t \rceil \\
\lceil t_1 \otimes t_2 \rceil &= \lceil t_1 \rceil \otimes \lceil t_2 \rceil \\
\lceil \text{let } t \text{ be } x \otimes y \text{ in } u \rceil &= \text{let } \lceil t \rceil \text{ be } x \otimes y \text{ in } \lceil u \rceil \\
\lceil * \rceil &= * \\
\lceil \text{let } t \text{ be } * \text{ in } u \rceil &= \text{let } \lceil t \rceil \text{ be } * \text{ in } \lceil u \rceil
\end{aligned}
$$

For each match-sequential prefix symbol $\overline{C} \in \overline{\Sigma}$ and each constant $F \in \Sigma$ we generate an HIS rewrite rule. In the following we assume that the sequential index

of C is its first hole. There are three possible cases, depending on the status of the context $C' = C[F\Box \dots \Box, \Box, \dots, \Box]$ as a prefix of R.

1. Match success: if C' is a redex, i.e. $C[F\xi_1 \dots \xi_k, \xi_{k+1}, \dots, \xi_n] \to r$, the term is rewritten to the right-hand side, though lifted to enable further reduction.

$$\lambda\xi_1. \dots \lambda\xi_n.\overline{C}(F\xi_1 \dots \xi_k)\xi_{k+1} \dots \xi_n \to \lambda\xi_1. \dots \lambda\xi_n.\lceil r \rceil$$

2. No match yet: if C' is a strict prefix of R, the context consumes the symbol F, and will go on to inspect another subterm.

$$\lambda\xi_1. \dots \lambda\xi_n.\overline{C}(F\xi_1 \dots \xi_k)\xi_{k+1} \dots \xi_n \to \lambda\xi_1. \dots \lambda\xi_n.\overline{C[F\Box \dots \Box, \Box, \dots, \Box]}\xi_1 \dots \xi_n$$

3. Match failure: otherwise, C' is neither a prefix of R nor a redex; the term is replaced with the 'literal' filling of the context, as matching has failed.

$$\lambda\xi_1. \dots \lambda\xi_n.\overline{C}(F\xi_1 \dots \xi_k)\xi_{k+1} \dots \xi_n \to \lambda\xi_1. \dots \lambda\xi_n.C[F\xi_1 \dots \xi_k, \xi_{k+1}, \dots, \xi_n]$$

Example 18. The following HRS, adapted from Example 16,

$$\begin{aligned} \lambda\xi.^0 FA\xi &\to r_1 \\ \lambda\xi.^0 F(GE\xi)B &\to r_2 \\ \lambda\xi.^0 F(G\xi E)C &\to r_3 \end{aligned}$$

Translates into the HIS:

$$\begin{aligned} \lambda\xi.\lambda\zeta.\overline{\Box}(F\xi\zeta) &\to \lambda\xi.\lambda\zeta.\overline{F\Box\Box}(* \otimes \xi \otimes \zeta) \\ \lambda\xi.\overline{F\Box\Box}(* \otimes A \otimes \xi) &\to \lceil r_1 \rceil \\ \lambda\xi.\lambda\zeta.\lambda\varsigma.\overline{F\Box\Box}(* \otimes G\xi\zeta \otimes \varsigma) &\to \lambda\xi.\lambda\zeta.\lambda\varsigma.\overline{F(G\Box\Box)\Box}(* \otimes * \otimes \xi \otimes \zeta \otimes \varsigma) \\ \lambda\xi.\lambda\zeta.\overline{F(G\Box\Box)\Box}(* \otimes * \otimes \xi \otimes \zeta \otimes B) &\to \lambda\xi.\lambda\zeta.\overline{F(G\Box\Box)B}(* \otimes * \otimes \xi \otimes \zeta \otimes *) \\ \lambda\zeta.\overline{F(G(\Box\Box)B}(* \otimes * \otimes E \otimes \zeta \otimes *) &\to \lceil r_2 \rceil \\ \lambda\xi.\lambda\zeta.\overline{F(G(\Box\Box)\Box}(* \otimes * \otimes \xi \otimes \zeta \otimes C) &\to \lambda\xi.\lambda\zeta.\overline{F(G\Box\Box)C}(* \otimes * \otimes \xi \otimes \zeta \otimes *) \\ \lambda\xi.\overline{F(G(\Box\Box)C}(* \otimes * \otimes \xi \otimes E \otimes *) &\to \lceil r_3 \rceil \end{aligned}$$

Plus various rules of the form e.g. $\lambda\xi.\lambda\zeta.\overline{F(G\Box\Box)\Box}(* \otimes * \otimes \xi \otimes \zeta \otimes E) \rightarrow \lambda\xi.\lambda\zeta.F(G\xi\zeta)E$, which act as match failure conditions.

Theorem 11. *Index reduction of the resulting HIS simulates index reduction of the original HRS.*

Proof. By the same token as Theorem 8, owing to the correctness of the context embedding established in Theorem 10. □

5.4 Unfolding sℓ-structures

The effective sharing graph rewriting strategy as already described can be generalised to match-sequential Linear HRSs without first translating them into a Higher-order Interaction System, simply by treating a contiguous network of symbol links, without interspersed multiplexers, as if it were a discrete Interaction Net agent in itself. We diverge here from Interaction Nets by forgetting the distinction of principal and auxiliary ports, and working instead with arbitrary link graphs.

5.4.1 Optimal index reduction

The subgraph corresponding to a match-sequential prefix of a purely-linear-pattern HRS — which must by definition comprise contiguous symbol links — will have a distinguished port corresponding to that context's sequential index. An optimal reduction strategy may then permit a multiplexer to propagate through this distinguished port, as is the case for the principal ports of agents in an Interaction Net (Guerrini 1996), because it is the subgraph that would need to be inspected, according to sequentiality, in order to determine whether the term is a matching redex. Thus, the propagation of a multiplexer through this port serves to further the matching process, until the port connects to a symbol link and it can be inspected to determine a match. We say that a port that is permitted for π-interaction is *open*, and one that is not is *closed*.

Thus each of a form's ports is tagged as open or closed; a multiplexer may freely interact with an open port, but cannot with a closed port. Open ports correspond to the principal ports of forms in Interaction Nets, but instead of belonging to

a single agent they belong to a network of contiguous links serving as a term context. When matching, we open a port when inspecting it, i.e. if it corresponds to a sequential index. This allows us to either inspect the form on the other end of the arrow at that port, or to allow multiplexers to pass through the form so as to 'unshare' or 'unfold' that portion of the sharing graph. Again, this is as though the whole subgraph, corresponding to the prefix of a pattern, were a single discrete agent, and the open port its principal port.

Theorem 12. *Index reduction on sharing graphs, wherein multiplexers propagate only through links' sequential indices, is optimal for non-rigid match-sequential HRSs.*

Proof. By Theorem 11, a non-rigid match-sequential HRS may be encoded into an HIS, and index reduction of such an HIS simulates index reduction of the original HRS. By Theorem 7, an HIS is optimal since it may be translated into an Interaction Net. However, we may omit the encoding step and equivalently apply the reduction strategy to the non-rigid match-sequential HRS itself. □

One possible optimisation here is that if there are sequential indices which need not be in either specific order — e.g. in the singleton ruleset $F(A, B) \to C$, either hole of $F(\Box, \Box)$ may be the first sequential index — then we may set both to 'open' for the purpose of parallelism. This is as though there were two principal ports for the link $\overline{F(\Box, \Box)}$, because the alternative reductions $\overline{F(\Box, \Box)}(A, B) \to \overline{F(A, \Box)}(B) \to C$, and $\overline{F(\Box, \Box)}(A, B) \to \overline{F(\Box, B)}(A) \to C$, are confluent. We could opt to open only one or the other, as in the earlier translation from match-sequential Linear HRSs to Interaction Nets in which an arbitrary sequential index is chosen as the principal port, but it is equivalent to compute the two ports' interactions concurrently.

5.4.2 Rigid bound variables

We have until now assumed that the left-hand side of any rule contains no rigid bound variable occurrences. This is because Interaction Nets as they are generally defined cannot have agents with rules for interacting through their principal port with *themselves*. For instance, take the HRS pattern $G(\lambda x.x)$; recall Example 7

and Figure 15. This pattern is orthogonal and match sequential, yet has thus far been forbidden due to the rigid occurrence of the bound variable x.

This rule, then, requires that G interact through its principal port (the second type atom occurrence from the left) with one of its auxiliary ports (the first). This is simply not possible with Interaction Nets, as an interaction rule is defined as being between two distinct agents. Nevertheless, this behaviour has no actual effect on the confluence of the (sℓ-structure) term graph rewriting system, as any interaction a link has with itself is equivalent to an interaction it might have with another unique link identifying its own port.

Theorem 13. *A link interacting with itself through two determinate ports is behaviourally equivalent to two links interacting through respective principal ports.*

Proof. Instead of a link that would have to interact with itself, suppose we instead used a pair of agents with a unique label ι, serving as a form of identity. The subsequent interaction between two such agents would be equivalent to the interaction of a link, if it were permitted, with itself. In any other case the agents would fall back as if the link were interacting with any other. □

Example 19. Consider the rewrite rule $F(C(\lambda x^0_!x)) \to A$, which may be translated to the following HIS, with a unique identifier ι for each $\overline{F(C(\lambda x^0_!\Box))}_\iota : 0 \multimap 0$ and $\overline{x}_\iota : 1 \multimap 0$ instantiated from the right-hand side of the second rule.

$$\begin{aligned}
\lambda\zeta^0_!\overline{\Box}(F\zeta) &\to \lambda\xi^0_!\overline{F\Box}\zeta \\
\lambda\xi^{0\multimap 0}_!\overline{F\Box}(C(\lambda x^0_!\xi x)) &\to \lambda\xi^0_!\overline{F(C(\lambda x^0_!\Box))}_\iota(\xi(\overline{x}_\iota *)) \\
\overline{F(C(\lambda x^0_!\Box))}_\iota(\overline{x}_\iota *) &\to \overline{\Box}A
\end{aligned}$$

Theorem 14. *Index reduction is optimal for match-sequential HRS even with rigid bound variables.*

Proof. Straightforward by Theorems 12 and 13. □

Thus we have an optimal reduction strategy for all match-sequential HRSs. This result is significantly more general than the optimal reduction strategies previously discovered for second-order Interaction Systems (Asperti & Laneve 1994) and match-sequential constructor TRSs (Fernández & Mackie 1996).

5.4.3 Sequentiality 'through binders'

A side effect of admitting bound variables is that our graph-based definition of sequentiality becomes more distinctive compared to the term-based approach: we can take advantage of sequentiality 'through binders', by permitting sequential term inspection by traversal not only from a constant to an argument, but also through the binding edge of an abstraction to the (linear) occurrence of its bound variable. A more radical alternative for broadening the possibilities for principal ports is introduced later as a generalisation of higher-order patterns (§6.2).

Example 20. Recall 'Gustave's TRS' from Example 3. The following is a second-order variant in a Linear HRS, in which the symbol $F : (0 \multimap (0 \otimes 0 \otimes 0)) \multimap 0$ binds a variable to occur in its body.

$$\lambda\xi^0.F(\lambda x^0.(Ax \otimes B \otimes \xi)) \to r_1$$
$$\lambda\xi^0.F(\lambda x^0.(\xi \otimes Ax \otimes B)) \to r_2$$
$$\lambda\xi^0.F(\lambda x^0.(B \otimes \xi \otimes Ax)) \to r_3$$

The first-order counterpart is non-sequential, since it is not known which of the arguments of F must be inspected in order to determine which rule matches. Yet in this second-order case there is another possible way in which the term may be traversed in order to determine the position of Ax: through the vertex corresponding to the port of F which binds x.

Here we enumerate the steps for matching a redex for rule 2 above. We take advantage of $\beta\eta$-equivalence in order to inspect whichever subterm is necessary at each step.

1. $\Box$
2. $F(\lambda x.\Box)$
3. $F(\lambda x.(\lambda y.\Box)(Ax))$
4. $F(\lambda x.(\lambda y.(\Box \otimes y \otimes \Box))(Ax)) = F(\lambda x.(\Box \otimes Ax \otimes \Box))$
5. $F(\lambda x.(\Box \otimes Ax \otimes B))$

Note how at step 3 we make use of the substitution calculus to inspect the environs of the bound variable x, although we do not yet know where it occurs in the term. It is not until step 4 that we discover this information.

This more general notion of sequentiality is another obstacle in translating match-sequential HRSs to HISs: this sequentiality cannot necessarily be linearised as necessary for Higher-order Interaction Systems, even if it may into Interaction Nets, because it would require terms on the left-hand side that are not Miller's (1991) higher-order patterns. We will however later explore a generalisation of higher-order patterns so as to include such terms (see §6.2).

Chapter 6

Extensions

There are a number of extensions and generalisations we can make to the algorithm as previously described. These range from altering the substitution calculus used for rewriting (§6.1, §6.3) to generalising what is permitted on the left-hand sides of rules (§6.2). These extensions have been grouped into a separate chapter either because they cannot be translated into Interaction Nets, or because they introduce other complexities we have been able to avoid by restricting ourselves to the multiplicative fragment of Intuitionistic Linear Logic. Nevertheless, all extensions are effectively computable using the technique previously described, perhaps with some minor alterations.

6.1 Substructural type systems

6.1.1 Additive types

Additive types for the linear λ-calculus (Benton & Wadler 1996) are described in Figure 38. A motivating example is an *if–then–else* function symbol. Without additive types, such a symbol (on some type τ) may be typed as $\mathrm{If} : \mathbb{B} \multimap\ !\tau \multimap\ !\tau \multimap\ !\tau$, and its rewrite rules defined as follows:

$$\mathrm{If\ True}\ \xi\ \zeta \to \mathrm{discard}\ \zeta\ \mathrm{in}\ \xi$$
$$\mathrm{If\ False}\ \xi\ \zeta \to \mathrm{discard}\ \xi\ \mathrm{in}\ \zeta$$

$$\frac{\Gamma \vdash s : \sigma \quad \Gamma \vdash t : \tau}{\Gamma \vdash (s,t) : \sigma \,\&\, \tau} \qquad \frac{\Gamma \vdash t : \sigma \,\&\, \tau}{\Gamma \vdash \text{fst } t : \sigma} \qquad \frac{\Gamma \vdash t : \sigma \,\&\, \tau}{\Gamma \vdash \text{snd } t : \tau} \qquad \frac{}{\Gamma \vdash () : 1}$$

Figure 38: Additive types in ILL

The use of weakening requires both ξ and ζ to be intuitionistic, yet in each case exactly one of them is used. Additive types allow us to state this explicitly: that *one* of the two will be used. Thus we have $\text{If} : \mathbb{B} \multimap (\tau \,\&\, \tau) \multimap \tau$, with the following rules:

$$\text{If True } \xi \to \text{fst } \xi$$
$$\text{If False } \xi \to \text{snd } \xi$$

The fundamental problem is that although the linear variable x in the term $\lambda x.(s,t)$ is only *used* once, in the sense of linear logic, it does still occur more than once syntactically, as it is in the basis of both s and t. Thus, in the corresponding sharing graph, the variable must be shared with a fan, and the expressions 'fst' and 'snd' would then use an eraser to discard the unused instance. This is illustrated in Figure 39. Notably, although this would then require fans, unlike in the case of contraction, no promotion (and so a raised sharing level) is necessary, as it is only actually *used* once. Additive types are also unique in their fanning and erasure of an entire type context, as opposed to individual types as in the rules for contraction and weakening.

The reason we chose not to include additive types in our Linear HRSs, aside from our limiting the calculus to that in Benton et al. (1992), is that they introduce extra complexity into the system with little concrete benefit. The use of fans and erasers for contraction and weakening of modals on the one hand, and purely syntactic duplication and erasure of free variables in additive terms on the other, although convenient implementation-wise, can serve to obscure the precise function of each of these forms of multiplexer, which in the multiplicative fragment of the linear λ-calculus are single purpose.

An alternative is to translate addition by closing over each term and then passing in the required variables when the choice is made. This complicates the

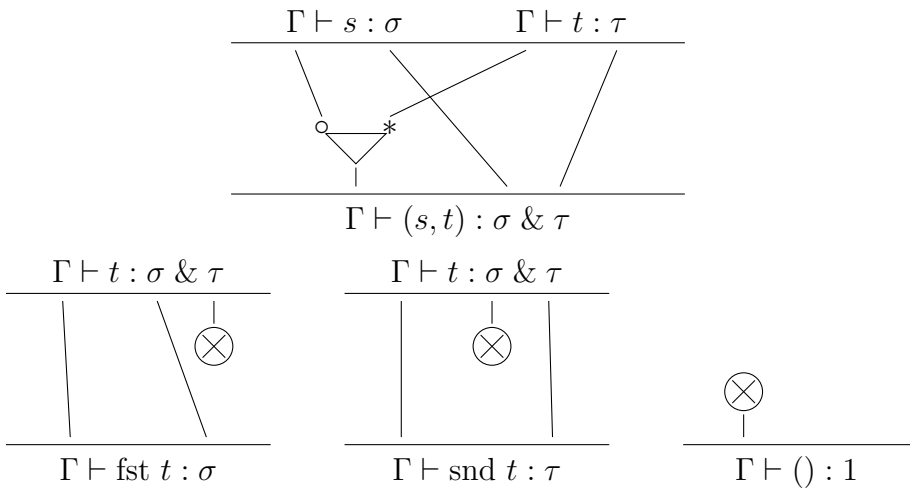


Figure 39: Translations of additive types

types of symbols in the HRS signature, and requires these terms to be handled appropriately at run-time, but have the benefit that they do not require the presence of multiplexers. This corresponds to one of the optimisations proposed by Asperti & Guerrini (1998) for the *append* function in BOHM, although their optimisation required detecting such a case, whereas in our case it is explicit in the type system.

$$\frac{\vec{x} : \vec{\rho} \vdash (s, t) : \sigma \mathbin{\&} \tau}{\vec{x} : \vec{\rho} \vdash \vec{x} \otimes (\lambda\vec{x}.s, \lambda\vec{x}.t) : \vec{\rho} \otimes ((\vec{\rho} \multimap \sigma) \mathbin{\&} (\vec{\rho} \multimap \tau))}$$

6.1.2 Affine types

An affine type system is a substructural type system which, unlike the linear types otherwise used in this thesis, forbid contraction but admit weakening without explicit promotion. Thus, although contraction requires explicit annotation with the exponentiation type operator, weakening can occur anywhere to a variable of any type. This means, for example, that whereas in a linear system (without additive types) an *if–then–else* function symbol (on some type τ) may be typed as $\mathbb{B} \multimap !\tau \multimap !\tau \multimap !\tau$, in an affine system it would be given the more general type $\mathbb{B} \to_{\mathrm{a}} \tau \to_{\mathrm{a}} \tau \to_{\mathrm{a}} \tau$, where $\to_{\mathrm{a}}$ indicates *affine* implication. This is because no subterm is ever contracted in a rewrite step, only weakened, and so we need not promote its non-Boolean arguments to an intuitionistic type.

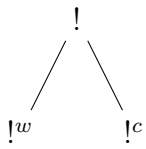


Figure 40: Partial ordering on Jacobs' modalities

Affine types are used in BOHM (Asperti & Guerrini 1998), a Lévy-optimal implementation of an enriched λ-calculus. There is no change needed to our translation in order to support affine types; so long as we still have the following substructural expressions then their translations will hold. However, note the lack of an exponentiation type operator in the weakening case.

$$\frac{\Gamma \vdash s : \sigma \quad \Delta \vdash t : \tau}{\Gamma, \Delta \vdash \text{discard } s \text{ in } t : \tau} \qquad \frac{\Gamma \vdash s : !\sigma \quad \Delta, x : !\sigma, y : !\sigma \vdash t : \tau}{\Gamma, \Delta \vdash \text{copy } s \text{ as } x, y \text{ in } t : \tau}$$

6.1.3 Multiple modalities

Jacobs (1994) splits the modality '!' of Girard (1987) into two distinct modalities, '$!^c$' for contraction, and '$!^w$' for weakening. Thus, a premise $!^c\phi$ may be used more than once, and $!^w\phi$ less than once. Their union is then the intuitionistic modality '!'. Bierman (1998) further generalises these dual modalities to arbitrary modalities, and introduces an accompanying term calculus. The separated linear λ-calculus, which has as its type system the substructural logic of Jacobs, has as its modalities the set $\{!^c, !^w, !\}$, with the partial ordering shown in Figure 40. It then permits the contraction or weakening only of terms whose types are of an appropriate modality, i.e. '$!^c$' or '!', and '$!^w$' or '!', respectively.

Bierman uses a term calculus in the style of DILL (Barber & Plotkin 1997), but the type system may be mapped to a term calculus in the style of Benton et al. (1992), as described in Figure 41. The 'order' expression is required to reduce an intuitionistic type $!\tau$ to an affine ($!^c\tau$) or relevance ($!^w\tau$) type. Bierman treats these 'permissible' conversions as implicit coercions, but we prefer to make them explicit in the syntax, which we feel is more in keeping with the spirit of ILL.

$$\frac{}{x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash t : \sigma \multimap \tau \quad \Delta \vdash u : \sigma}{\Gamma, \Delta \vdash tu : \tau} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x^{\sigma}.t : \sigma \multimap \tau}$$

$$\frac{\Delta_1 \vdash s_1 : \Box_1\sigma_1 \quad \ldots \quad \Delta_n \vdash s_n : \Box_n\sigma_n \quad x_1 : \Box_1\sigma_1, \ldots, x_n : \Box_n\sigma_n \vdash t : \tau \quad \Box_i \geq \Box'}{\Delta_1, \ldots, \Delta_n \vdash \text{promote}^{\Box'} \text{ for } s_1, \ldots, s_n \text{ in } x_1, \ldots, x_n \text{ in } t : \Box'\tau}$$

$$\frac{\Gamma \vdash t : \Box\tau}{\Gamma \vdash \text{derelict } t : \tau} \qquad \frac{\Gamma \vdash t : \Box\tau \quad \Box \geq \Box'}{\Gamma \vdash \text{order}^{\Box'}\ t : \Box'\tau}$$

Figure 41: Term calculus with arbitrary modalities

6.2 Generalised patterns

Another opportunity for generalisation, specifically in the context of Linear HRSs, is that the notion of higher-order patterns can be loosened in order to allow a free variable's linear arguments to be effectively 'rigidified'. That is, as a linear bound variable must occur exactly once, we may loosen the requirement that arguments to a free variable be distinct bound variables, and instead permit an argument to be any term containing a rigid linear variable occurrence. This is similar to having 'sequentiality through binders' (§5.4.3), but rather than permitting us to follow the binding edge merely so as to determine a match sequentially, we may do so in order to establish a match that would not otherwise be possible at all, as we do not cycle back to the variable's binding link. One example of this occurring is in the translation from an HRS with 'sequentiality through binders' to a Higher-order Interaction System.

Example 21. Recall the second-order variant of 'Gustave's TRS' from Example 20 (§5.4.3):

$$\lambda\xi^{0}.F(\lambda x^{0}.(Ax \otimes B \otimes \xi)) \to r_1$$
$$\lambda\xi^{0}.F(\lambda x^{0}.(\xi \otimes Ax \otimes B)) \to r_2$$
$$\lambda\xi^{0}.F(\lambda x^{0}.(B \otimes \xi \otimes Ax)) \to r_3$$

If we were to try to encode this system into an HIS by the translation described

earlier (§5.3.3), the result would be as follows (cleaned up a little for brevity):

$$\begin{aligned}
\lambda\xi.^{0\multimap(0\otimes 0\otimes 0)}\overline{\Box}(F(\lambda x.^{0}\xi x)) &\to \lambda\xi.^{0\multimap(0\otimes 0\otimes 0)}\overline{F(\lambda x.\Box x)}(\lambda x.^{0}\xi x)\\
\lambda\xi.^{0\multimap(0\otimes 0\otimes 0)}\overline{F\Box}(\lambda x.^{0}\xi(Ax)) &\to \lambda\xi.^{0\multimap(0\otimes 0\otimes 0)}\overline{F(\lambda x.\Box(Ax))}(\lambda x.^{0}\xi x)\\
\lambda\xi.^{0}\lambda\zeta.^{0}\overline{F(\lambda x.\Box(Ax))}(\lambda x.^{0}(x\otimes\xi\otimes\zeta)) &\to \lambda\xi.^{0}\lambda\zeta.^{0}\overline{F(\lambda x.(Ax\otimes\Box_1\otimes\Box_2))}(\xi\otimes\zeta)\\
\lambda\zeta.^{0}\overline{F(\lambda x.(Ax\otimes\Box_1\otimes\Box_2))}(B\otimes\zeta) &\to \lceil r_1\rceil\\
\lambda\xi.^{0}\lambda\zeta.^{0}\overline{F(\lambda x.\Box(Ax))}(\lambda x.^{0}(\zeta\otimes x\otimes\xi)) &\to \lambda\xi.^{0}\lambda\zeta.^{0}\overline{F(\lambda x.(\Box_2\otimes Ax\otimes\Box_1))}(\xi\otimes\zeta)\\
\lambda\zeta.^{0}\overline{F(\lambda x.(\Box_2\otimes Ax\otimes\Box_1))}(B\otimes\zeta) &\to \lceil r_2\rceil\\
\lambda\xi.^{0}\lambda\zeta.^{0}\overline{F(\lambda x.\Box(Ax))}(\lambda x.^{0}(\xi\otimes\zeta\otimes x)) &\to \lambda\xi.^{0}\lambda\zeta.^{0}\overline{F(\lambda x.(\Box_1\otimes\Box_2\otimes Ax))}(\xi\otimes\zeta)\\
\lambda\zeta.^{0}\overline{F(\lambda x.(\Box_1\otimes\Box_2\otimes Ax))}(B\otimes\zeta) &\to \lceil r_3\rceil
\end{aligned}$$

The term $\lambda\xi.\overline{F\Box}(\lambda x.^{0}\xi(Ax))$, on the left-hand side of rule 2, is not however permitted as a higher-order pattern, as the parameter ξ is applied to the non-variable term Ax.

Definition 65. A *pattern component* is a $\beta\eta$-normal context containing no substructural expressions, whose holes comprise the arguments to free variables.

Definition 66. A *generalised pattern* is a pattern component whose holes are filled by terms, each of which is either (i) η-equivalent to a distinct bound variable, or (ii) itself a component containing in its body — not in a hole, and so in 'rigid position' (Nipkow 1991) — at least one non-contracting variable bound outside of that component.

The term $\lambda\xi.\overline{F\Box}(\lambda x.^{0}\xi(Ax))$ would then be a generalised pattern, since the term Ax, to which the parameter ξ is applied, is not a distinct bound variable yet contains the 'rigid' linear variable x. This latter constraint means that another term $\lambda\xi.\lambda\zeta.\overline{F\Box}(\lambda x.^{0}\xi(A(\zeta x)))$ is not, however, a generalised pattern, since x is an argument to ζ within another argument, and so does not occur 'rigid'.

Theorem 15. *A generalised pattern has a unique most general unifier.*

Proof. The one case that our patterns do not have in common with those of Miller (1991) is clause (ii) of Definition 66. In this case, as the subterm must contain at least one linear bound variable, which is unique, and the context of this occurrence

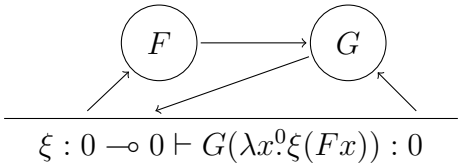


Figure 42: Generalised pattern $G(\lambda x^0_!\xi(Fx))$

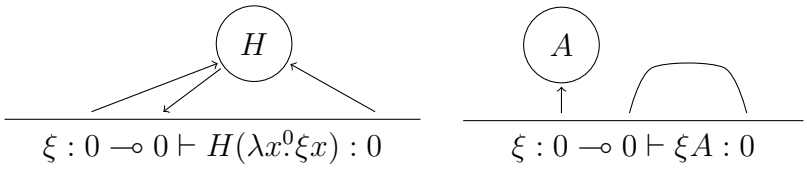


Figure 43: Example right-hand sides to a generalised pattern

is *rigid* in the sense of Nipkow (1991), only one match is possible: the unique rigid subterm containing that variable occurrence. □

Note that this does not hold for affine variables — i.e. those which cannot contract but may weaken — since $(\lambda x.t)u = t$ for any term u, so there is no unique most general unifier. Intuitively, if we were to walk the binding edge of such a variable then the link we reach on the other end could well be a proper subterm of the binding abstraction, but it could just as well be garbage.

Example 22. The term $G(\lambda x^0_!\xi(Fx))$ is a generalised pattern, corresponding to the Interaction Net in Figure 42. Although F is not a direct argument of G, because it is applied to x, which is bound by an abstraction that *is* a direct argument of G, $[\![F]\!]$ is still a neighbouring link of $[\![G]\!]$ in the corresponding link graph. This means the term $\lambda\xi^0_!G(\lambda x^0_!\xi(Fx))$ does in fact translate to a legal left-hand side to an Interaction Net rewrite rule. The right-hand side of such a rule could be, for instance, $\lambda\xi^0_!H(\lambda x^0_!\xi x)$ or $\lambda\xi^0_!\xi A$, the body of each of which is illustrated in Figure 43.

Note that, in this example, it is F that has a principal port at its head, and G at its tail — the arrow goes *from* F, *to* G — even though G is the outer symbol, which has before now always coïncided with the principal port being at the head of the corresponding link.

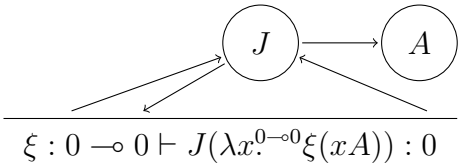


Figure 44: Generalised pattern $J(\lambda x^{0 \multimap 0}_{!} \xi(xA))$

Example 23. The term $J(\lambda x^{0 \multimap 0}_{!} \xi(xA))$ is likewise a generalised pattern, corresponding again to the Interaction Net in Figure 44. Here the principal arrow goes from J to A, as one might otherwise expect.

6.2.1 Limitations

A fundamental problem with generalised patterns is that the mechanism of higher-order unification used in higher-order rewriting requires more sophisticated matching than can be effectively achieved using sharing graphs. Suppose for example we have the term $G(\lambda x^{0}_{!} \xi(Cx\zeta))$. This may look very similar to the generalised pattern $G(\lambda x^{0}_{!} \xi(Fx))$ discussed earlier, but the addition of ζ actually means that it cannot be implemented as an Interaction Net. This is because the substitute for ζ cannot contain any variable bound within the substitute for ξ. For example, this can be unified with $G(\lambda x^{0}_{!} G(\lambda y^{0}_{!} B(CxA)y))$, by the substitution $\theta = \{\xi \mapsto \lambda z^{0}_{!} G(\lambda y^{0}_{!} Bzy), \zeta \mapsto A\}$, but not with $G(\lambda x^{0}_{!} G(\lambda y^{0}_{!} B(Cxy)A))$, because y is not free in $\theta\zeta$. These complications mean that only certain non-pattern terms can be implemented as Interaction Nets.

On additive types

In the presence of additive types, a variable may occur *syntactically* in multiple subexpressions: given some term $\lambda x^{0}_{!}(s, t)$, the linear variable x will occur in both s and t, even though it is only *used* once. This syntactic duplication would undermine the presumption that a variable of linear type must be a direct arrow to the appropriate form, as opposed to there potentially being multiplexers between the two agents that are attempting to interact.

The solution for rigid variables and sequentiality 'through binders' would be simply to forbid rigid bound variables within additive terms on the left-hand side

of rules. There is no straightforward workaround to this problem in the case of generalised patterns, as the effect it has on matching is not limited to the pattern in which it occurs.

On affine types

An affine variable may be weakened, leading not to term rewriting but to term narrowing. For example, in an affine HRS the generalised pattern $G(\lambda x.^0 \xi(x\zeta))$ would match $G(\lambda x.^0 \text{discard } x \text{ in } A)$, because x is not required to occur in ξ. The parameter ζ may then be assigned any value during the rewrite step, so a rewrite may effectively pull terms out of nowhere.

In terms of sharing graphs, this means the portion of the graph the corresponding arrow connects to may be garbage. Traversing this arrow during matching might then lead to one inspecting garbage links, disconnected from the rest of the graph. A solution to this may be to use *multiple modalities* (§6.1.3), so that we may distinguish between linear and affine types, as well as intuitionistic. This way $G(\lambda x.^0 \xi(Fx))$ could only match a term $G(\lambda x.^0 C[Fx])$, and its affine counterpart $G(\lambda x.^{!^w 0} \xi(Fx))$ would be forbidden because x, being affine, may lead to garbage.

6.3 Further extensions

Finally, we describe methods of translating polymorphism and fixed points from an augmented substitution calculus also into Lamping–Gonthier sharing graphs. With these two additions in place, it should be possible to perform Lévy-optimal reduction of higher-order term rewriting systems with what is essentially a purely-functional programming language, à la 'ISWIM' (Landin 1966), as their substitution calculus — though minus pattern matching, which can of course be handled by rewrite rules atop the substitution calculus.

6.3.1 Polymorphism

If polymorphism of the calculus is limited to that of the Hindley–Milner type system (Damas & Milner 1982), i.e. with polymorphic behaviour limited to let-expressions, then any term may be preprocessed to remove polymorphism through

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{\Gamma \vdash t : \sigma \to \tau \quad \Gamma \vdash u : \sigma}{\Gamma, \Delta \vdash tu : \tau} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x^{\sigma}.t : \sigma \to \tau}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \Lambda\alpha.t : \forall\alpha.\tau} \qquad \frac{\Gamma \vdash t : \forall\alpha.\tau}{\Gamma \vdash t[\sigma] : \tau[\sigma/\alpha]}$$

Figure 45: (Intuitionistic) System F

type erasure. There are however two problems with this. The first problem is that such preprocessing requires that polymorphic subterms be unshared, which rather undermines the subsequent optimal sharing. The second problem is that this cannot be done for higher-rank polymorphism, as present in types like $(\forall\alpha.(\alpha \multimap \alpha)) \multimap I$. This is not equivalent to $\forall\alpha.((\alpha \multimap \alpha) \multimap I)$: the latter will take a term of any type $\tau \multimap \tau$, whereas the former will only take a term of type $\forall\alpha.(\alpha \multimap \alpha)$, the sole occupant of which is the identity function.

Higher-rank polymorphism is supported by the fully polymorphic System F (Girard 1972), the rules for which are given in Figure 45. The first row of rules are the same as in the simply-typed λ-calculus; the second row are unique to System F, being type abstraction and type application (specialisation) respectively. To use the earlier example of higher-rank polymorphism, the following may be derived in System F (assuming we have $* : I$):

$$\dfrac{\dfrac{\dfrac{}{x : \forall\alpha.(\alpha \to \alpha) \vdash x : \forall\alpha.(\alpha \to \alpha)}}{x : \forall\alpha.(\alpha \to \alpha) \vdash x[I] : I \to I} \quad * : I}{x : \forall\alpha.(\alpha \to \alpha) \vdash x[I]* : I}}{\vdash \lambda x^{\forall\alpha.(\alpha\to\alpha)}x[I]* : (\forall\alpha.(\alpha \to \alpha)) \to I}$$

Clearly, for our purposes, we require a *linear* System F, built upon the ILL term calculus instead of the intuitionistic simply-typed λ-calculus assumed in 'vanilla' System F. The key to translating the system into sharing graphs is in the type substitution $\tau[\sigma/\alpha]$. For this, we multiplex the single arrow corresponding to each occurrence in τ of type variable α, into however many arrows are required by type σ. This is illustrated in Figure 46.

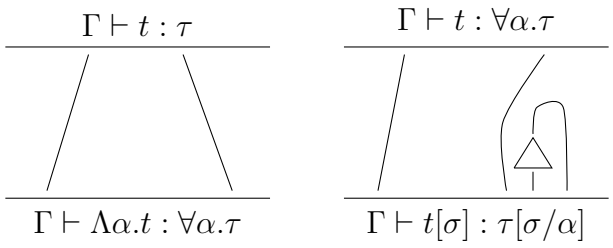


Figure 46: Translations of polymorphic types

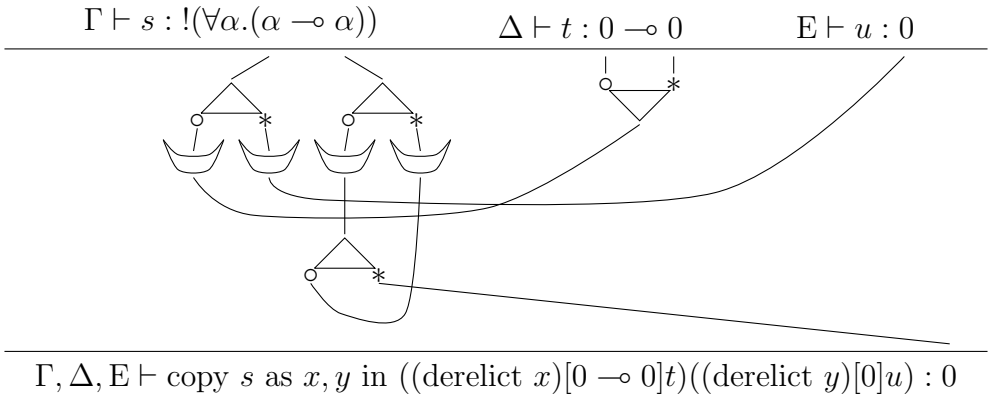


Figure 47: Example term graph with polymorphism

Example 24. The following term is translated to the graph in Figure 47.

$$\frac{\Gamma \vdash s : !(\forall\alpha.(\alpha \multimap \alpha)) \quad \Delta \vdash t : 0 \multimap 0 \quad \mathrm{E} \vdash u : 0}{\Gamma, \Delta, \mathrm{E} \vdash \text{copy } s \text{ as } x, y \text{ in } ((\text{derelict } x)[0 \multimap 0]t)((\text{derelict } y)[0]u) : 0}$$

6.3.2 Fixed points

Recursion may be implemented in Linear HRSs at the term rewriting level by such a rewrite rule as, where $\text{Fix} : !(!0 \multimap 0) \multimap 0$:

$$\text{Fix } \xi \to \text{copy } \xi \text{ as } \xi_1, \xi_2 \text{ in } (\text{derelict } \xi_1)(\text{promote } \xi_2 \text{ for } \xi_2' \text{ in Fix } \xi_2')$$

However, we may alternatively introduce a primitive recursion operator into the substitution calculus by an additional equivalence rule, $\mu x.t = t[\mu x.t/x]$. Such an operator is implemented as an optimisation in BOHM by way of the *Chroboczek*

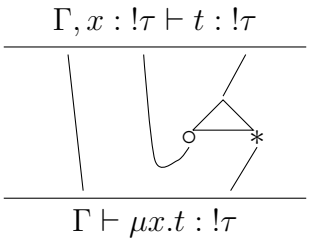


Figure 48: Simplified Chroboczek encoding

encoding (Asperti & Guerrini 1998), which translates the μ-operator as a self-referential binary Guerrini multiplexer with auxiliary port offsets 1 and -1, and a set of brackets. With our algorithm, due to the requisite promotion and dereliction being made explicit by the type system, this encoding is reduced to a greatly simplified self-referential Lamping fan, as illustrated in Figure 48.

Chapter 7

Conclusion

As explained in the Introduction (§1), Van Oostrom (1996) identified the following potential areas of research:

> The next question is whether it is possible to implement an evaluator for orthogonal HRSs in which redexes in the same family are actually represented by the same structure, and which only ever contracts redexes which will contribute to the final result of the computation.
>
> For the second, a generalisation to the higher-order case of the theory of *sequentiality* and *neededness* as founded in Huet & Lévy (1991) for orthogonal TRSs, is required.

We have introduced an optimal evaluator for orthogonal higher-order rewrite systems, which is effective when the system is match sequential, though with the added proviso that the ruleset have purely-linear left-hand sides (in the substructural sense). This is a large class of HRSs, subsuming all term rewriting systems for which optimal evaluators have yet been discovered. Our evaluator builds upon much of the theory developed for the second-order case, but takes it further by generalising Lamping–Gonthier sharing from second-order Bourbaki graphs to higher-order abstract syntax. Treating these structures as a substitution calculus enables us to perform higher-order term rewriting modulo the unfolding of a Lamping–Gonthier sharing graph, and so achieve Lévy optimality. Although match sequentiality is necessary in order for our algorithm to find needed redexes

effectively, all we really require for optimality is that (only) needed redexes be reduced modulo π-equivalence. This is possible, as Van Oostrom (1996) observes, "due to the logical nature of substitution, not rules."

7.1 Untyped λ-calculus

Asperti & Guerrini's (1998) encoding from the untyped λ-calculus into an Interaction Net maps each abstraction and application in an untyped λ-term to an agent in the Interaction Net, just as would the straightforward Interaction System (Laneve 1993) for the untyped λ-calculus:

$$\frac{}{\Gamma, x \vdash x}\ \text{Var} \qquad \frac{\Gamma, x \vdash t}{\Gamma \vdash \text{Abs}(x.t)}\ \text{Abs} \qquad \frac{\Gamma \vdash t_1 \quad \Gamma \vdash t_2}{\Gamma \vdash \text{App}(t_1, t_2)}\ \text{App}$$

Naturally, we can define an equivalent Linear HRS for the untyped λ-calculus. Here we also distinguish the special case of an abstraction whose bound variable does not occur, as this is likewise distinguished when the above Interaction System is translated into an Interaction Net.

$$\frac{}{x : !0 \vdash \text{derelict } x : 0}\ \text{Var} \qquad \frac{\Gamma, x : !0 \vdash t : 0}{\Gamma \vdash \text{Abs } (\lambda x{:}^{!0} t) : 0}\ \text{Abs}$$

$$\frac{\Gamma_1, \Delta \vdash t_1 : 0 \quad \Gamma_2, \Delta \vdash t_2 : 0}{\Gamma_1, \Gamma_2, \Delta \vdash \text{copy } \Delta \text{ as } \Delta, \Delta' \text{ in App } t_1 \text{ (promote } \Gamma_2, \Delta' \text{ for } \Gamma_2, \Delta \text{ in } t_2) : 0}\ \text{App}$$

$$\frac{\Gamma \vdash t : 0}{\Gamma \vdash \text{Abs } (\lambda x{:}^{!0} \text{discard } x \text{ in } t) : 0}\ \text{Abs}'$$

These terms are translated into Interaction Nets as illustrated in Figure 49. Comparing these to the initial encoding of the untyped λ-calculus (Asperti & Guerrini 1998, §3.3), the two are identical, allowing for our differences in graphical notation: instead of arrows leading to or from boxes labelled $[M]$ or $[N]$, the arrows lead to or from premises at the top of each diagram; and instead of a context arrow leading from the top of the diagram, the context arrow leads to the 'result type' 0 at the bottom. This equivalence demonstrates how our translation from HRSs to sharing graphs is a proper generalisation of the existing translation for the untyped

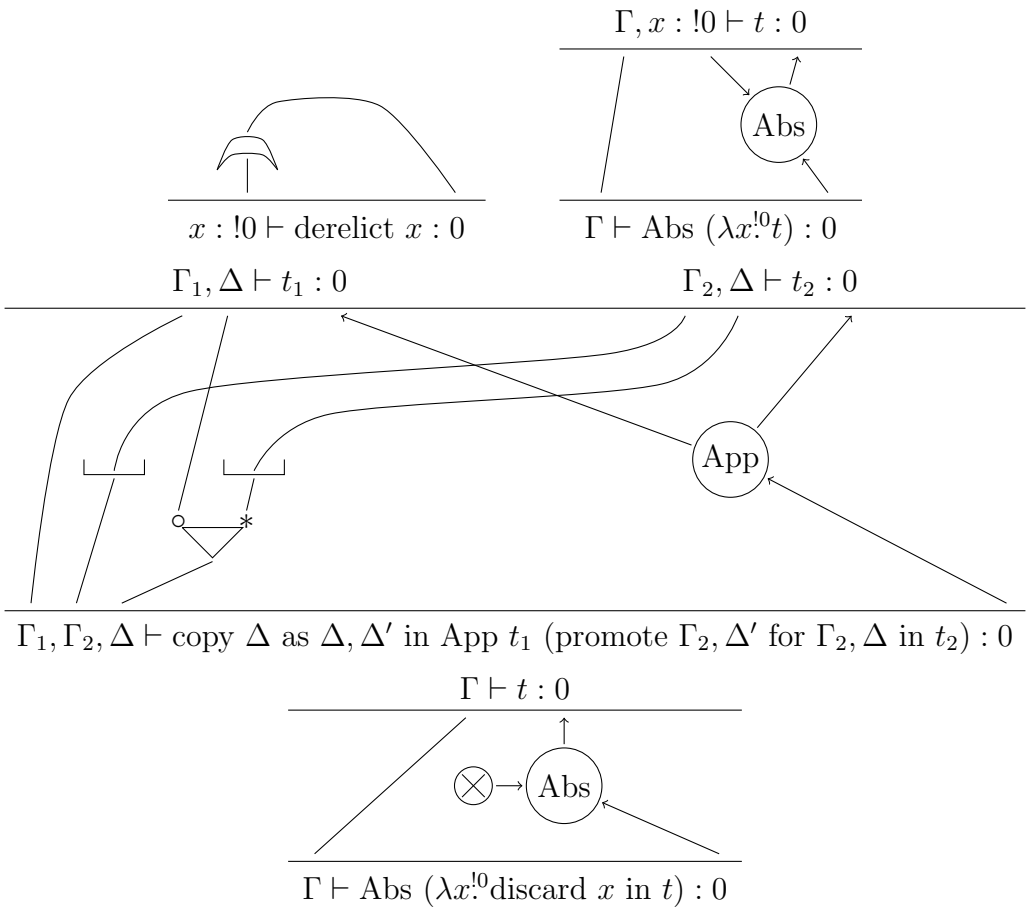


Figure 49: Translation of an HRS for the untyped λ-calculus

λ-calculus, and for Interaction Systems in general.

Although the results of these two translations, for ISs and HRSs, are identical in their result, ours is driven by induction on the underlying substitution calculus which, in contrast to the encoding of Asperti, is more resemblant of Gonthier et al.'s (1992) encoding of the untyped λ-calculus into Interaction Nets comprising only multiplexers. However, our translation makes the need for these multiplexers explicit through its use of the linear λ-calculus, instead of adding them implicitly in each expression. It could also be said that our translation unifies these two earlier approaches, by translating a typed λ-calculus à la Gonthier et al. (1992) which then serves as a substitution calculus for an 'object-level' untyped λ-calculus, resulting in an Interaction Net equivalent to that of Asperti & Guerrini (1998).

Alternative embeddings of the untyped λ-calculus into an HRS are also possible, as are alternative translations of linear λ-calculus (as a substitution calculus) into Interaction Nets, but we do not explore them here; cf. Asperti & Laneve (1995).

7.2 Future work

There are a number of possible avenues for future research:

- More rigorous a treatment of $s\ell$-structures as a substitution calculus. It has already been noted by Van Raamsdonk (1996) that the 'graphical' nature of such structures is something of a departure from the term-orientated substitution calculi of Higher-order Rewrite Systems (Nipkow 1991) or Combinatory Reduction Systems (Klop et al. 1993). We have focused our efforts on the algorithm for translating higher-order rewriting systems with Intuitionistic Linear Logic as a substitution calculus, to equivalent systems having instead $s\ell$-structures as a substitution calculus. However, we expect that certain concepts we have made use of in our treatment of these systems, such as contraction sets (§4.3.2) for establishing π-equivalence classes and normal forms, would benefit from more thorough formalisation.

- The substructural higher-order term rewriting systems, for which we have used our substitution calculus translation as a medium, are themselves an interesting topic, and one which to our knowledge has had little attention. The possible interactions between substructurally-typed variables, and the properties of a rewriting system featuring those variables, is especially interesting. Take, for instance, the effect of linear variables on sequentiality in our second-order variant of 'Gustave's TRS' (§5.4.3). Another example is that in a non-fully-extended orthogonal HRS whose bound variables are non-weakening, a non-normalised term must have an external redex, yet may still require variable occurrence checks (§5.1.2).

- Finally, an obvious practical project would be an efficient implementation of our technique. We have written a 'proof of concept', in the programming language Haskell, but it is not intended to be an efficient implementation,

rather following the theory 'by the book'. An optimised implementation more in the line of BOHM (Asperti & Guerrini 1998) would allow for the runtime behaviour of our algorithm to be properly compared to earlier Lévy-optimal implementations, as well as to other term rewriting software and functional language runtimes. The theoretical developments described here should enable an implementation to outperform BOHM even in the second-order case, let alone the higher.

Appendix A

Equivalence proofs

This appendix comprises figures for proofs of equivalence for our translations of purely-linear (§3.3.3) and modal (§4.3.3) terms from Intuitionistic Linear Logic to Lamping–Gonthier sharing graphs.

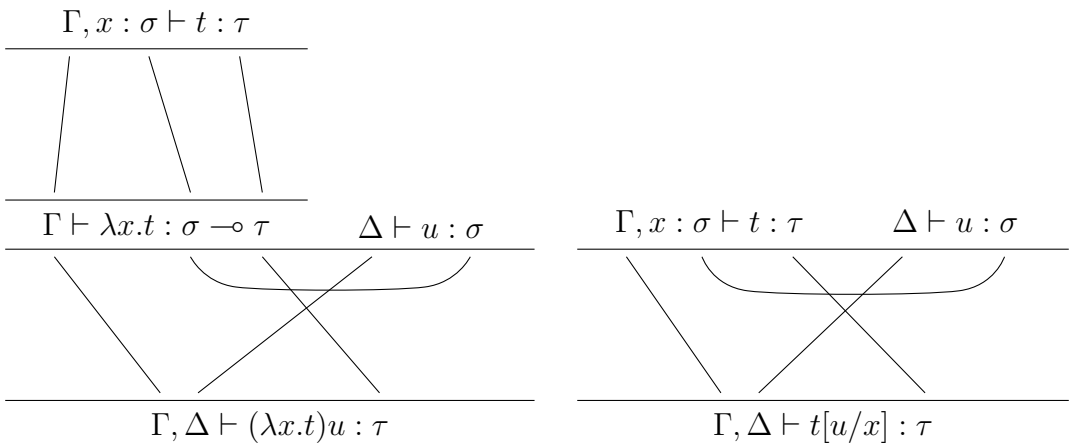


Figure 50: $(\lambda x.t)u = t[u/x]$

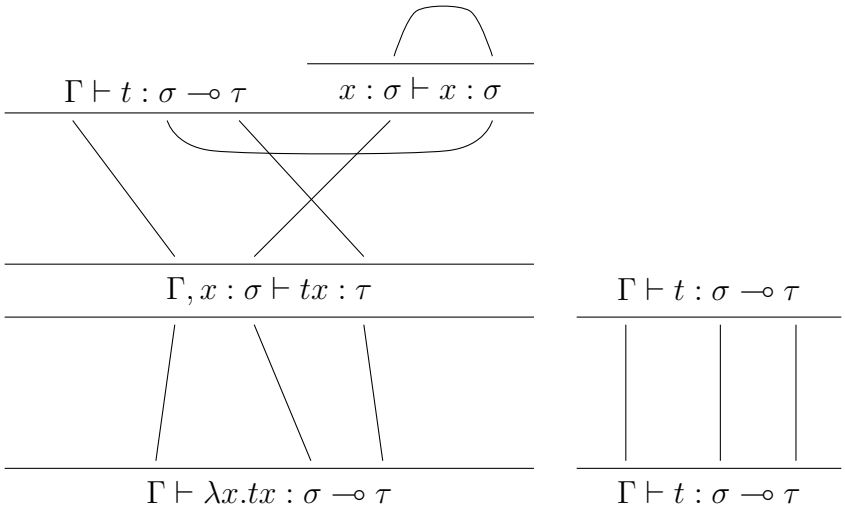


Figure 51: $\lambda x.tx = t$

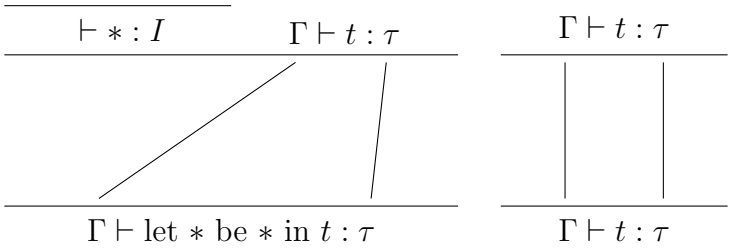


Figure 52: let $*$ be $*$ in $t = t$

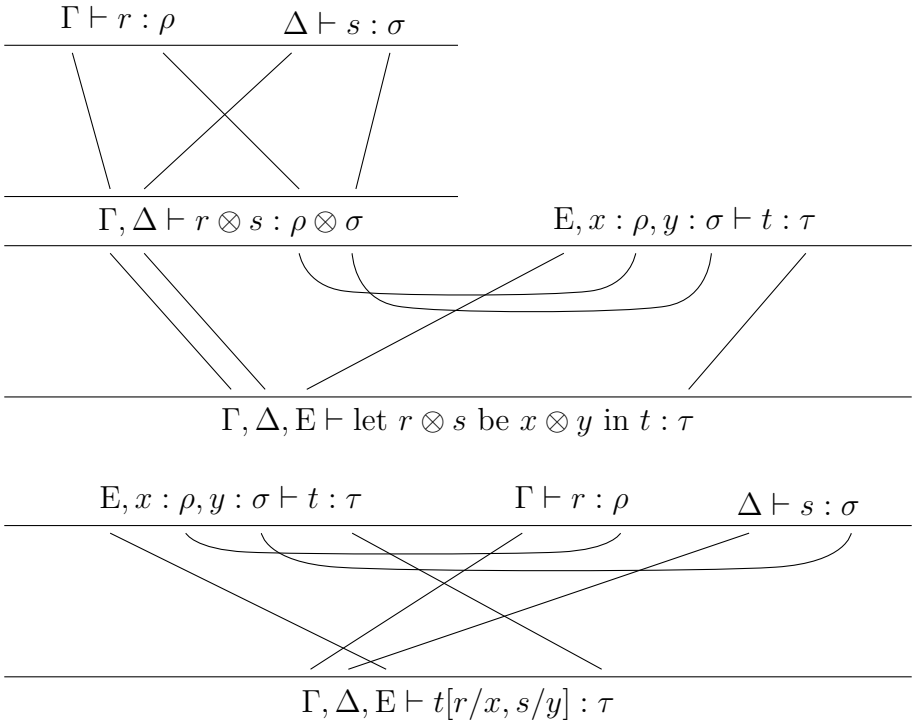


Figure 53: let $r \otimes s$ be $x \otimes y$ in $t = t[r/x, s/y]$

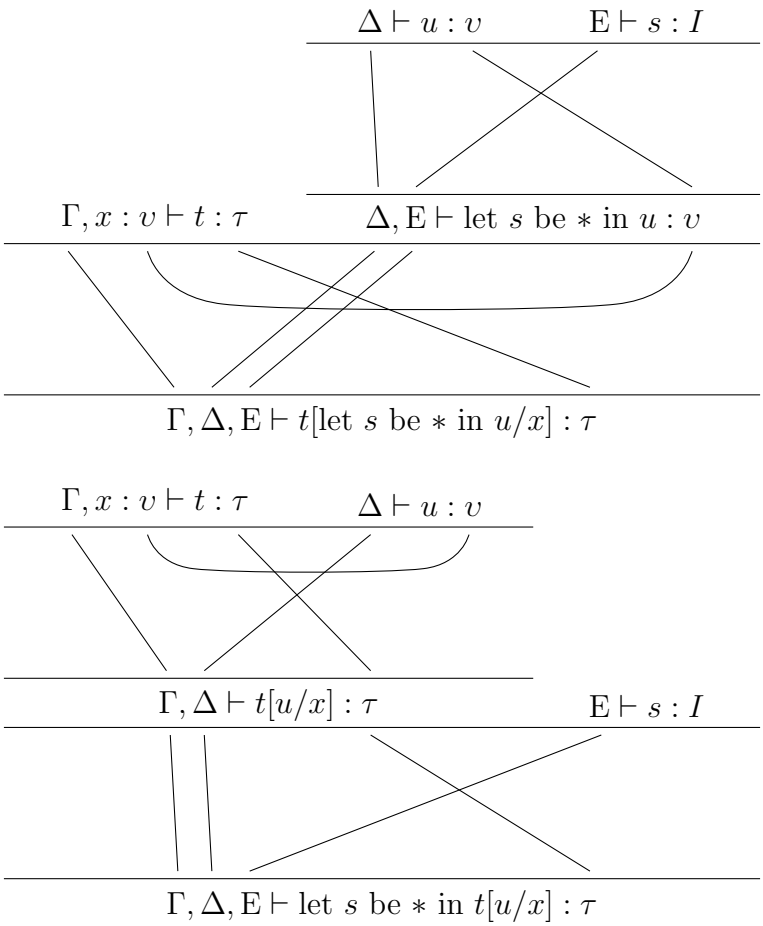


Figure 54: $t[\text{let } s \text{ be } * \text{ in } u/x] = \text{let } s \text{ be } * \text{ in } t[u/x]$

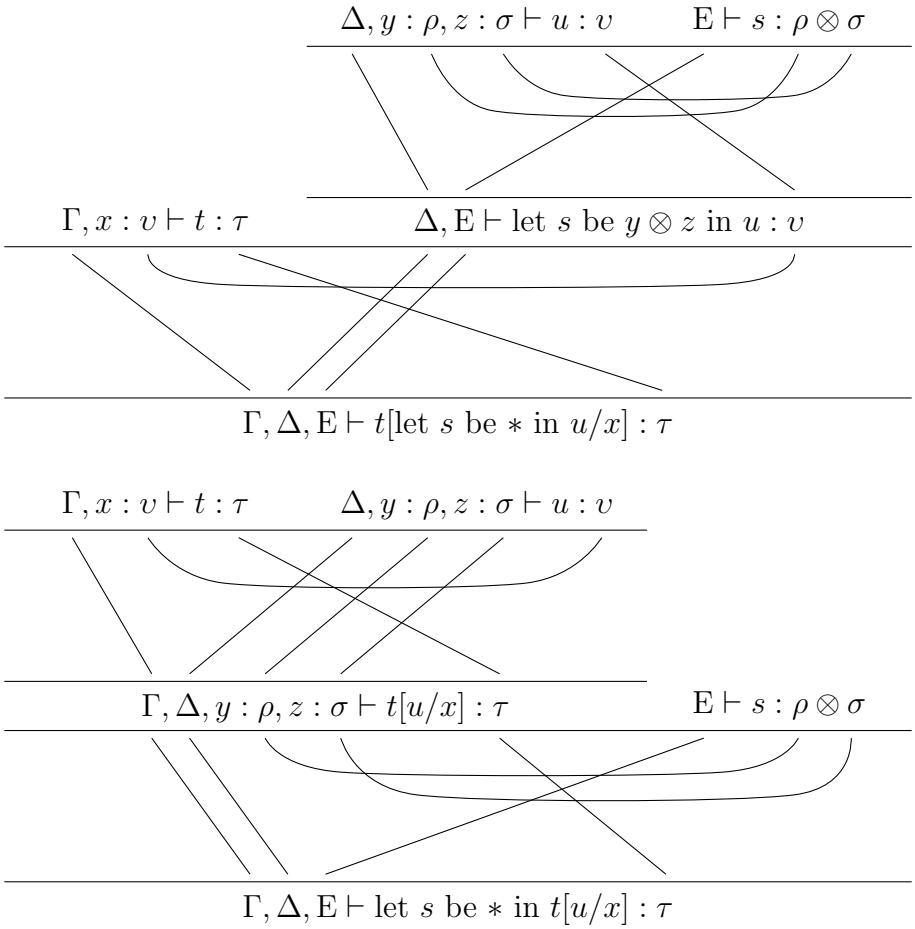


Figure 55: $t[\text{let } s \text{ be } y \otimes z \text{ in } u/x] = \text{let } s \text{ be } y \otimes z \text{ in } t[u/x]$

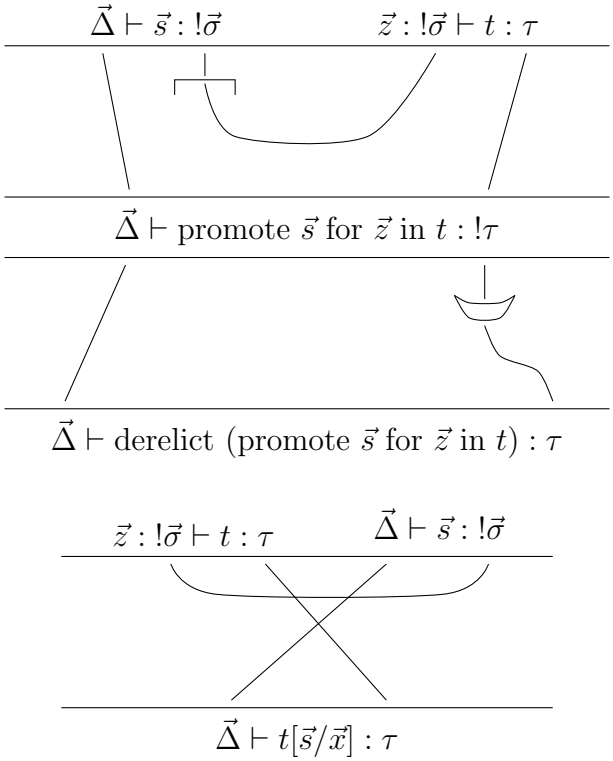


Figure 56: derelict (promote $\vec{s}$ for $\vec{z}$ in t) $= t[\vec{s}/\vec{z}]$

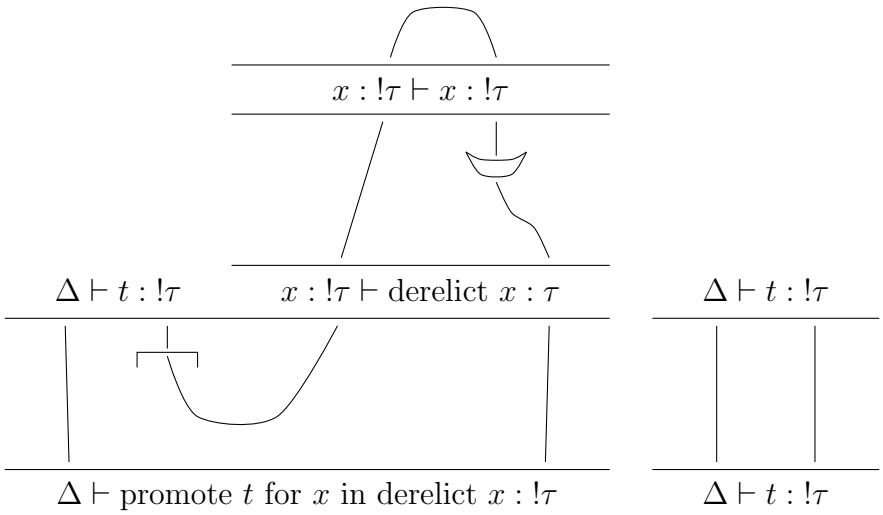


Figure 57: promote t for x in derelict $x = t$

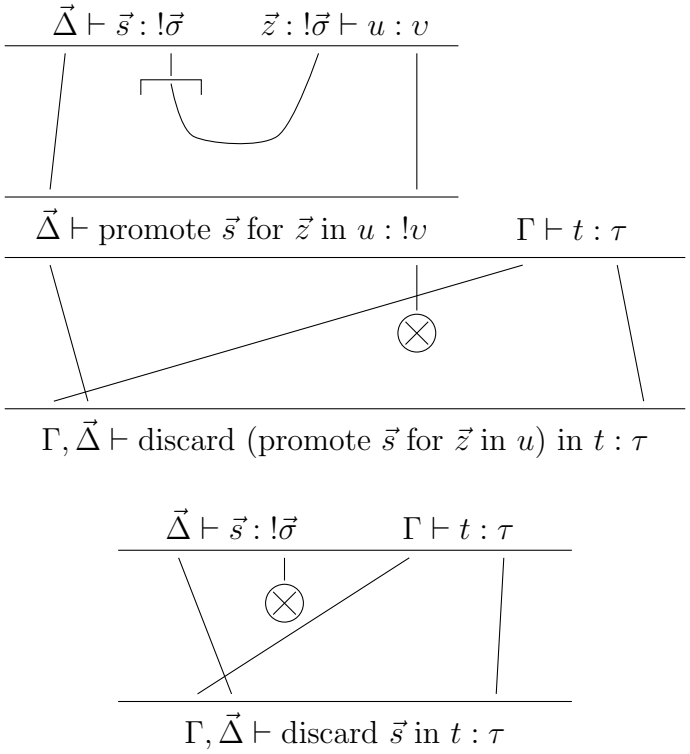


Figure 58: discard (promote $\vec{s}$ for $\vec{z}$ in u) in t = discard $\vec{s}$ in t

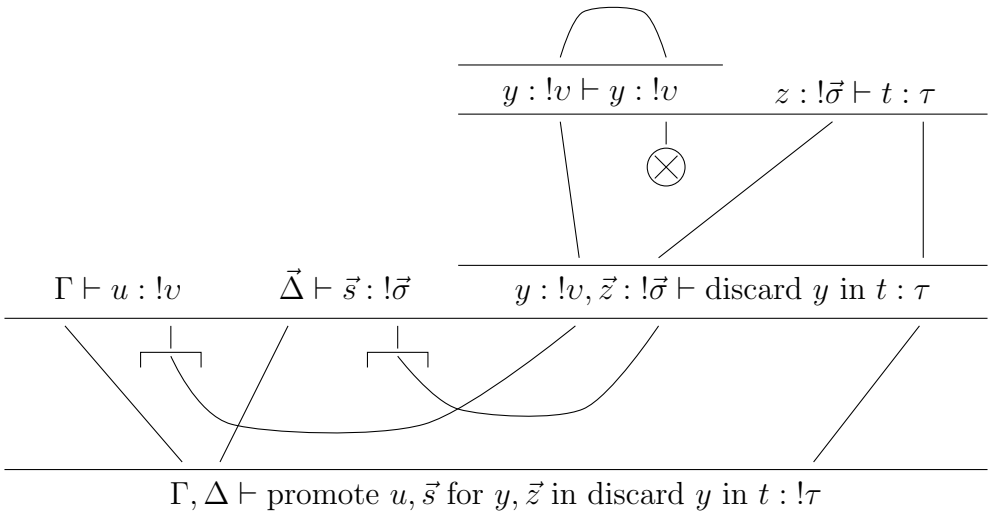


Figure 59: promote $u, \vec{s}$ for $y, \vec{z}$ in discard y in t

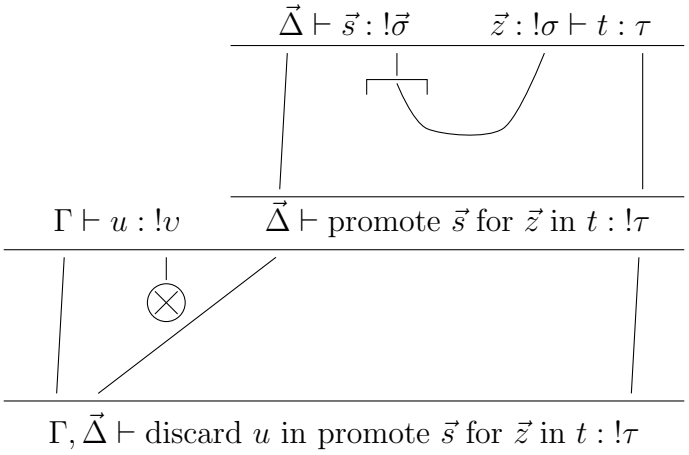
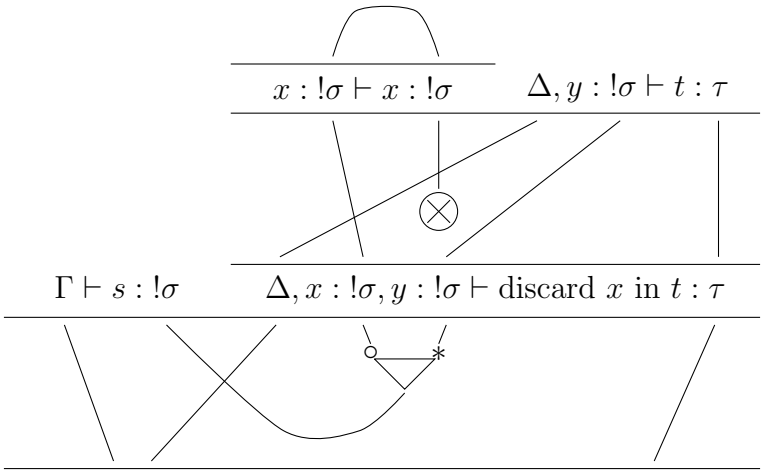


Figure 60: discard u in promote $\vec{s}$ for $\vec{z}$ in t



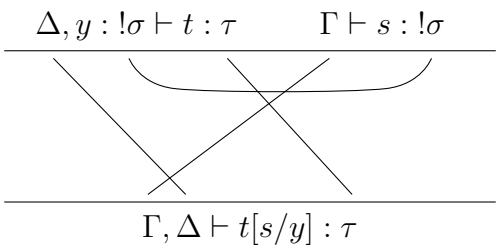


Figure 61: copy s as x, y in discard x in $t = t[s/y]$

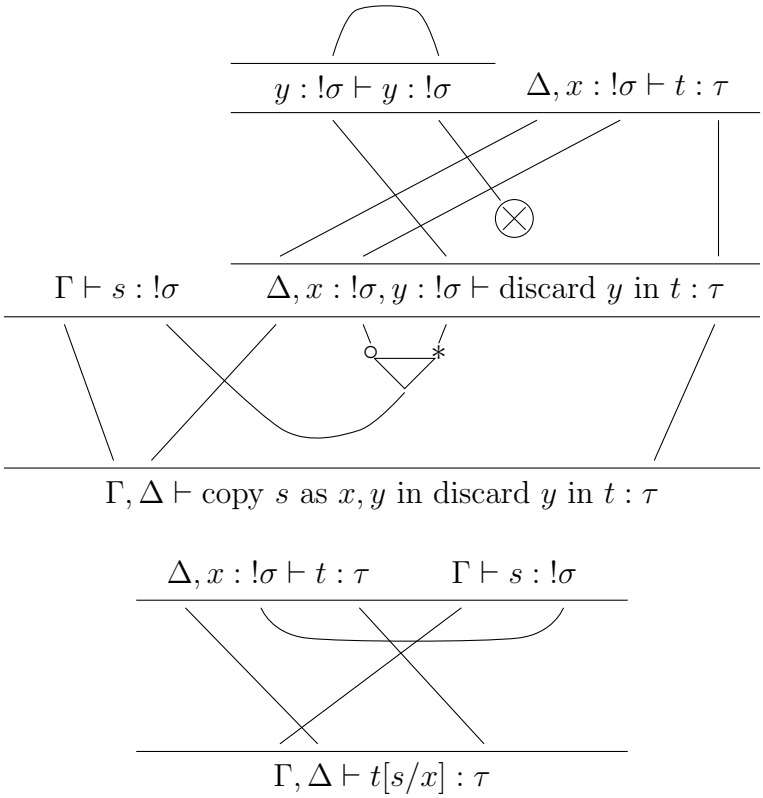


Figure 62: copy s as x, y in discard y in $t = t[s/x]$

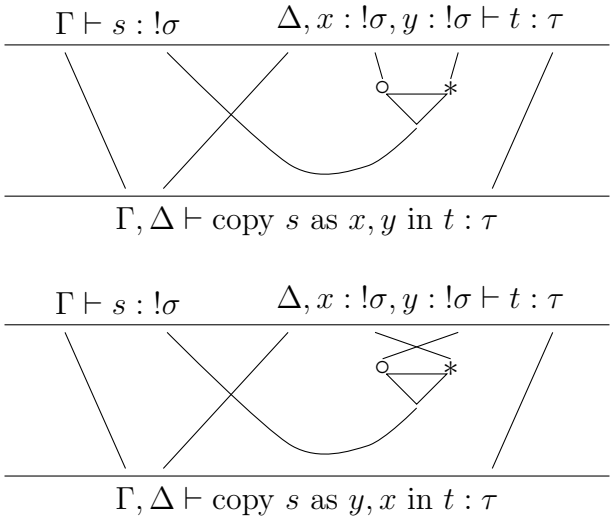


Figure 63: copy s as x, y in t = copy s as y, x in t

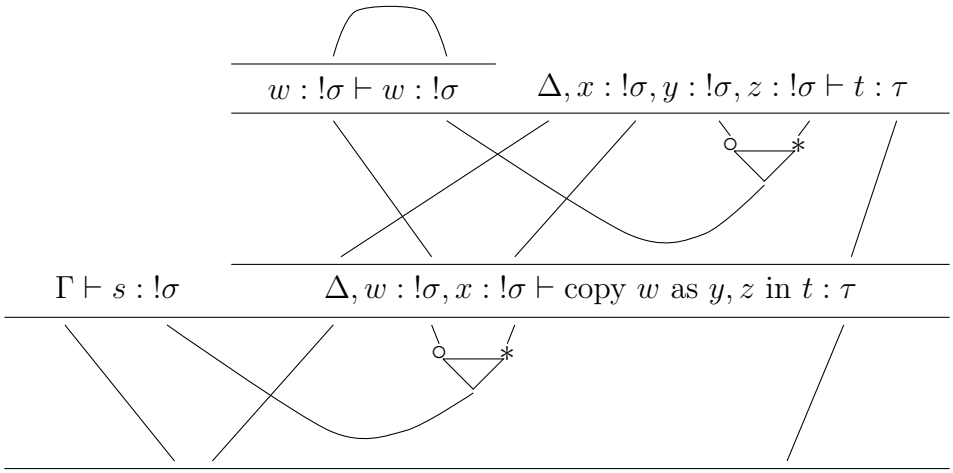


Figure 64: copy s as w, x in copy w as y, z in t

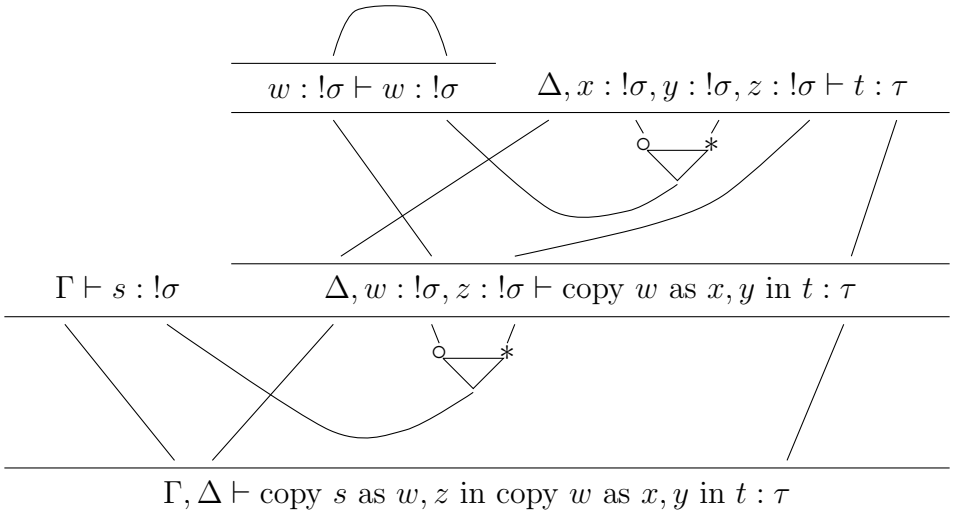


Figure 65: copy s as w, z in copy w as x, y in t

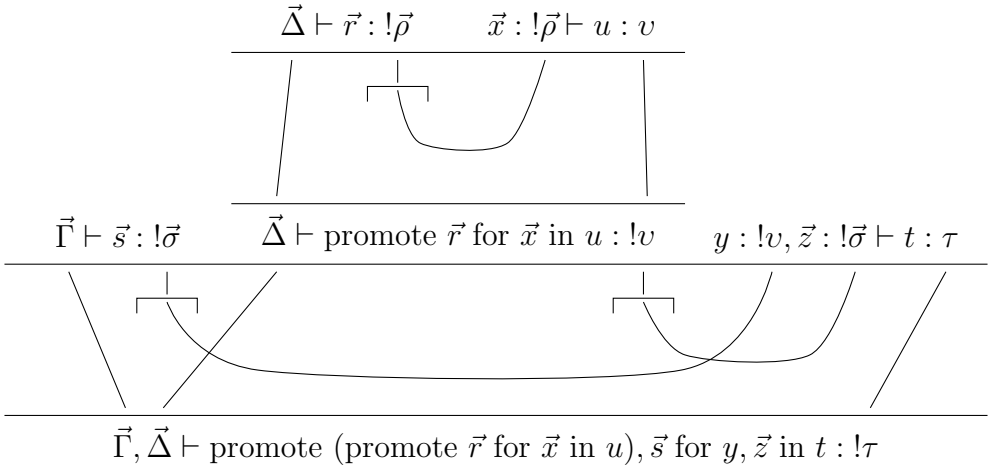


Figure 66: promote (promote $\vec{r}$ for $\vec{x}$ in u), $\vec{s}$ for $y, \vec{z}$ in t

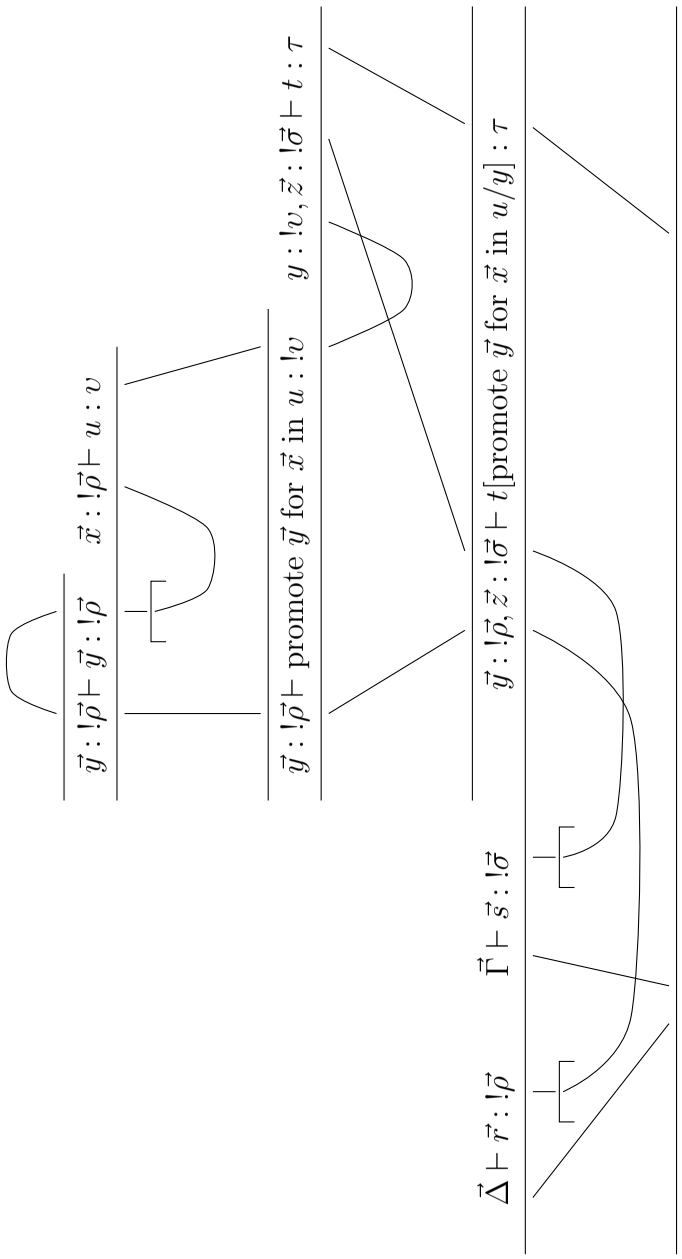


Figure 67: promote $\vec{r}, \vec{s}$ for $\vec{y}, \vec{z}$ in $t[\text{promote } \vec{y} \text{ for } \vec{x} \text{ in } u/y]$

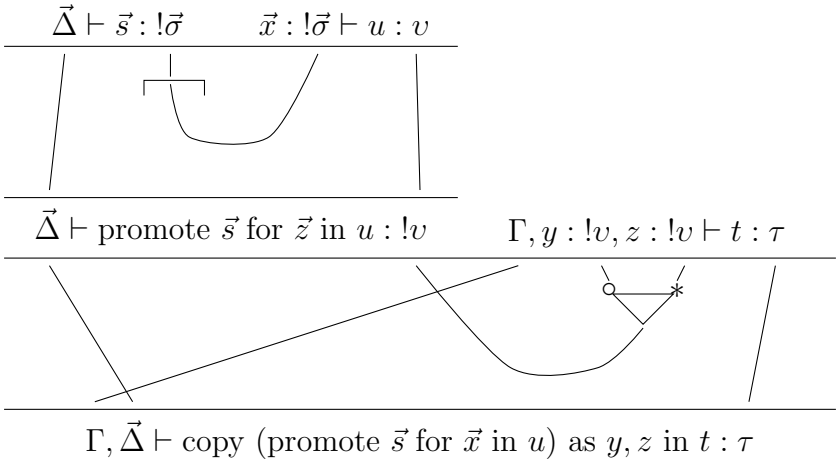


Figure 68: copy (promote $\vec{s}$ for $\vec{x}$ in u) as y, z in t

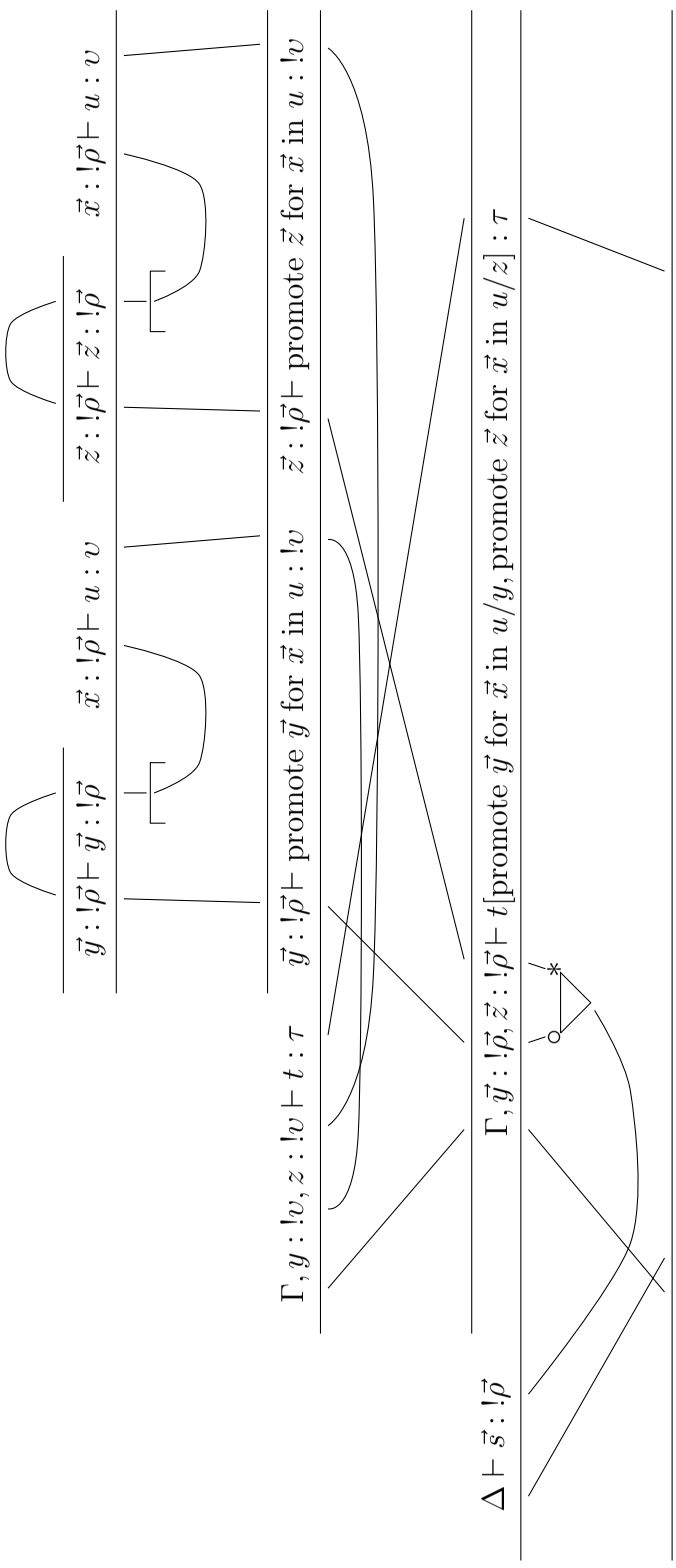


Figure 69: copy $\vec{s}$ as $\vec{y}, \vec{z}$ in $t[$promote $\vec{y}$ for $\vec{x}$ in $u/y,$ promote $\vec{z}$ for $\vec{x}$ in $u/z]$

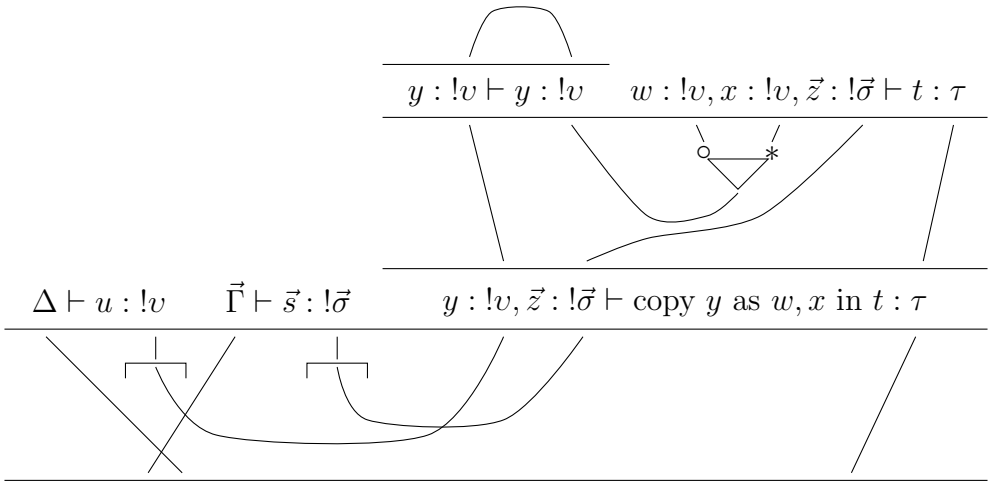


Figure 70: promote $u, \vec{s}$ for $y, \vec{z}$ in copy y as w, x in t

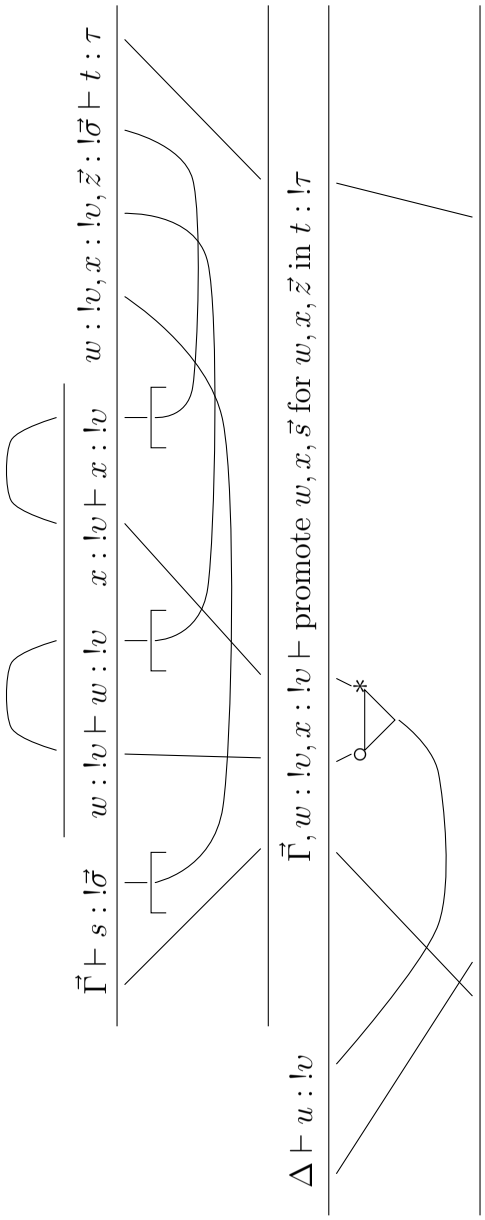


Figure 71: copy u as w, x in promote $w, x, \vec{s}$ for $w, x, \vec{z}$ in t

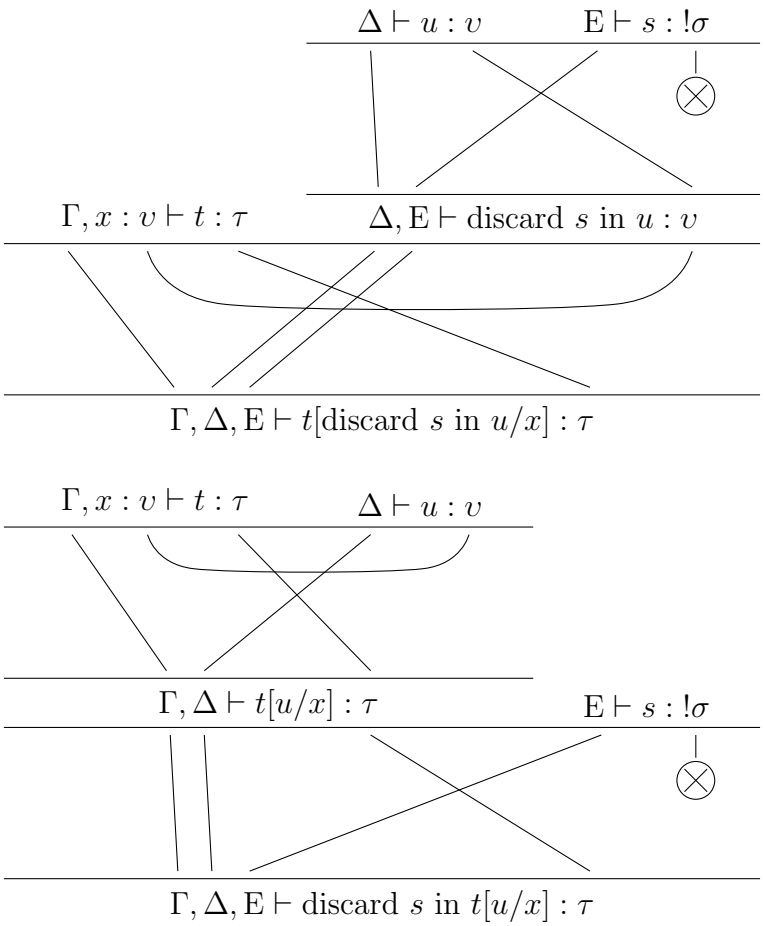


Figure 72: $t[\text{discard } s \text{ in } u/x] = \text{discard } s \text{ in } t[u/x]$

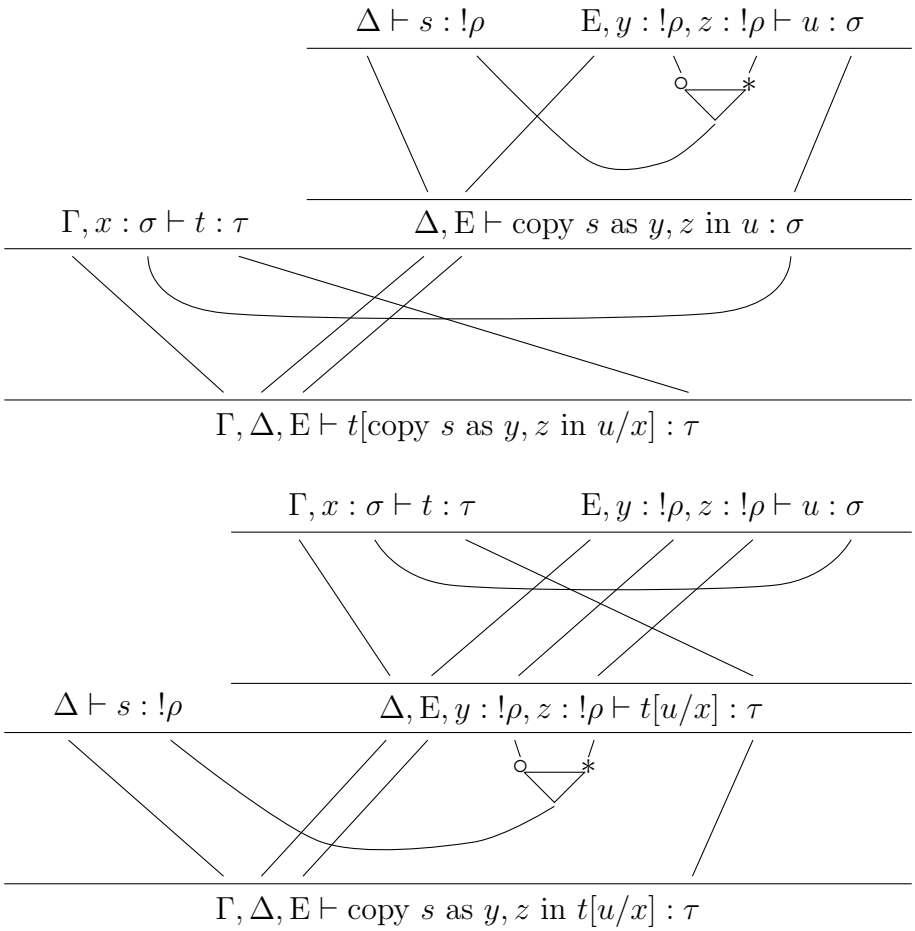


Figure 73: $t[\text{copy } s \text{ as } y, z \text{ in } u/x] = \text{copy } s \text{ as } y, z \text{ in } t[u/x]$

Appendix B

Krivine machines

As described in the introduction, Lévy optimality is but one cost model for the λ-calculus (Lawall & Mairson 1996), and indeed optimal reduction but one strategy for achieving efficient term sharing with strong β-reduction, i.e. reduction under abstraction (Fernández, Mackie & Sinot 2005). Our early work on this subject (Smith 2014) was based not on Lévy optimality and $s\ell$-structures, but instead on alternative notions of efficiency, and rather built on an abstract machine in the likeness of the Krivine machine (Krivine 2007). However, it was set aside when the alternative research otherwise described in this thesis seemed to bear more fruit. Nevertheless, it has since been cited in later research (Accattoli, Barenbaum & Mazza 2015). This appendix details our strong Krivine machine and its benefits over other abstract machines of its ilk.

The Krivine machine (Krivine 2007) comprises a term, an environment, and a stack. The latter two of these are lists of term–environment pairs, making environments a recursive data type. The rules for the original Krivine machine are given in Figure 74; through execution of these rules the machine is capable of evaluating a term's weak head normal form, at which point it will terminate. Note that we count De Bruijn indices from $\underline{0}$, as we feel it is more 'natural' to define the natural numbers to include 0.

The Krivine machine was extended by Abadi et al. (1991) to use explicit substitutions from the λσ-calculus as environments instead of lists as in the original.

$$
\begin{aligned}
(t_1 t_2, E, S) &\to (t_1, E, (t_2, E) \cdot S) \\
(\lambda t_1, E_1, (t_2, E_2) \cdot S) &\to (t_1, (t_2, E_2) \cdot E_1, S) \\
(\underline{0}, (t, E_2) \cdot E_1, S) &\to (t, E_2, S) \\
(\underline{n{+}1}, (t, E_2) \cdot E_1, S) &\to (\underline{n}, E_1, S)
\end{aligned}
$$

Figure 74: Krivine machine

These σ-substitutions are defined as follows:

$$
\begin{aligned}
\mathrm{id} &= \{n \mapsto n\} && \text{Identity} \\
\uparrow &= \{n \mapsto n{+}1\} && \text{Shift} \\
t \cdot \sigma &= \{0 \mapsto t, n{+}1 \mapsto \sigma(n)\} && \text{Cons} \\
\rho\,;\sigma &= \{n \mapsto \rho(n)[\sigma]\} && \text{Compose}
\end{aligned}
$$

Note that we use the syntax $\rho\,;\sigma$ for composition, whereas Abadi et al. use $\rho \circ \sigma$, as we feel it is more clear that substitution composition is left-to-right, rather than right-to-left as with function composition. The rules for the λσ-calculus Krivine machine are given in Figure 75.

This machine is quite similar to the λσ-calculus Krivine machine, but differentiates itself in three main respects. The first difference is that we use the $\lambda\sigma_{\Uparrow}$-calculus of Hardin & Lévy (1989), which introduces an additional substitution operator, $\Uparrow(\sigma)$.

$$
\Uparrow(\sigma) = \{0 \mapsto 0, n{+}1 \mapsto \sigma(n)[\uparrow]\} \qquad \text{Lift}
$$

The reason for our using this alternative explicit substitution calculus is that, when moving a substitution under an abstraction, the λσ-calculus lifts it by the rule $(\lambda t)[\sigma] \to \lambda(t[\underline{0} \cdot (\sigma\,;\uparrow)])$. This De Bruijn index $\underline{0}$ is created afresh, so if we wish to label variables with some kind of provenance information — in our case these were Lévy labels (Lévy 1980) — then they will be lost in the substitution $\underline{0}^{\ell}[\underline{0} \cdot (\sigma\,;\uparrow)] = \underline{0}$. We do not have this problem with the $\lambda\sigma_{\Uparrow}$-calculus substitution $\underline{0}^{\ell}[\Uparrow(\sigma)] = \underline{0}^{\ell}$.

$$
\begin{aligned}
(\underline{n}, \uparrow, S) &\to (\underline{n{+}1}, \mathrm{id}, S) \\
(\underline{0}, u[\rho] \cdot \sigma, S) &\to (u, \rho, S) \\
(\underline{n{+}1}, u \cdot \sigma, S) &\to (\underline{n}, \sigma, S) \\
(\underline{n}, \rho\,; \sigma, S) &\to (\underline{n}[\rho], \sigma, S) \\
(t_1 t_2, \sigma, S) &\to (t_1, \sigma, t_2[\sigma] \cdot S) \\
(\lambda t, \sigma, u[\rho] \cdot S) &\to (t, u[\rho] \cdot \sigma, S) \\
(\underline{n}[\mathrm{id}], \sigma, S) &\to (\underline{n}, \sigma, S) \\
(\underline{n}[\uparrow], \sigma, S) &\to (\underline{n{+}1}, \sigma, S) \\
(\underline{0}[u \cdot \rho], \sigma, S) &\to (u, \sigma, S) \\
(\underline{n{+}1}[u \cdot \rho], \sigma, S) &\to (\underline{n}[\rho], \sigma, S) \\
(\underline{n}[\pi\,; \rho], \sigma, S) &\to (\underline{n}[\pi], \rho\,; \sigma, S) \\
(t[\rho], \sigma, S) &\to (t, \rho\,; \sigma, S)
\end{aligned}
$$

Figure 75: Krivine machine for the λσ-calculus

A second somewhat minor difference is that we never construct a closure $t[\rho]$ in the term component of the Krivine machine, e.g. $(t[\rho], \sigma, S)$. Instead, the term component is always a fully-evaluated λ-term, as in the original Krivine machine. Besides being conceptually cleaner, this is preferable in the aforementioned case of using labelled substitutive terms, as the original term need not then be labelled even if substitutive closures are.

A third, far more distinctive, difference in our Krivine machine is that, just as environment lists were replaced with substitutions, we likewise replace the stack with a term context. This may be implemented as a zipper (Huet 1997), a data type representing a context which may be defined inductively like so:

$$\frac{}{\Box \in \mathit{Ter}_{\Box}(\Sigma)} \qquad \frac{t \in \mathit{Ter}(\Sigma) \quad \sigma \in \mathit{Sub}(\Sigma) \quad C \in \mathit{Ter}_{\Box}(\Sigma)}{(t[\sigma])\Box \in \mathit{Ter}_{\Box}(\Sigma)}$$

$$\frac{t \in \mathit{Ter}(\Sigma) \quad \sigma \in \mathit{Sub}(\Sigma) \quad C \in \mathit{Ter}_{\Box}(\Sigma)}{\Box(t[\sigma]) \in \mathit{Ter}_{\Box}(\Sigma)} \qquad \frac{C \in \mathit{Ter}_{\Box}(\Sigma)}{\lambda\Box\,; C \in \mathit{Ter}_{\Box}(\Sigma)}$$

Much as an environment list $t[\sigma] \cdot \mathrm{nil}$ maps to an substitution $t[\sigma] \cdot \mathrm{id}$, a stack list $t[\sigma] \cdot \mathrm{nil}$ maps to a context $\Box(t[\sigma])\,; \Box$.

$$
\begin{array}{rll}
(tu, \sigma, C) & \to (t, \sigma, \Box(u[\sigma])\,; C) & \text{Left} \\
(\lambda t, \sigma, \Box(u[\rho])\,; C) & \to (t, (u[\rho] \cdot \sigma)\,; \mathrm{id}, C) & \text{Beta} \\
\\
(\underline{n}, (\pi\,; \rho)\,; \sigma, C) & \to (\underline{n}, \pi\,; (\rho\,; \sigma), C) & \text{Associate} \\
(\underline{0}, (u[\pi] \cdot \rho)\,; \sigma, C) & \to (u, \pi\,; \sigma, C) & \text{Head} \\
(\underline{n{+}1}, (u[\pi] \cdot \rho)\,; \sigma, C) & \to (\underline{n}, \rho\,; \sigma, C) & \text{Tail} \\
(\underline{0}, \Uparrow(\rho)\,; \sigma, C) & \to (\underline{0}, \sigma, C) & \text{Naught} \\
(\underline{n{+}1}, \Uparrow(\rho)\,; \sigma, C) & \to (\underline{n}, \rho\,; \uparrow\,; \sigma, C) & \text{Lift} \\
(\underline{n}, \uparrow\,; \sigma, C) & \to (\underline{n{+}1}, \sigma, C) & \text{Shift} \\
(\underline{n}, \mathrm{id}\,; \sigma, C) & \to (\underline{n}, \sigma, C) & \text{Id} \\
\\
(\lambda t, \sigma, C) & \to (t, \Uparrow(\sigma)\,; \mathrm{id}, \lambda\Box\,; C) & \text{Lambda} \\
\\
(tu, \sigma, C) & \to (u, \sigma, (t[\sigma])\Box\,; C) & \text{Right} \\
(t, \sigma, \Box(u[\rho])\,; C) & \to (u, \rho, (t[\sigma])\Box\,; C) & \text{LeftRight} \\
(u, \rho, (t[\sigma])\Box\,; C) & \to (t, \sigma, \Box(u[\rho])\,; C) & \text{RightLeft}
\end{array}
$$

Figure 76: Krivine machine for the $\lambda\sigma_{\Uparrow}$-calculus with contexts

The rules for our abstract machine are given in Figure 76. In order to keep the definition simple, we assume that σ-substitutions are always of the pattern $(\sigma_1\,; (\sigma_2\,; \ldots\,; (\sigma_n\,; \mathrm{id})))$ where $n \geq 0$. The initial substitution, id, satisfies this pattern, and the rules of the machine maintain it — see for example 'Beta' and 'Lambda'. Our using contexts in place of a stack has four major effects on the machine's execution:

1. A machine configuration (t, σ, C) represents precisely the term $C[t[\sigma]]$; there is no need for any 'interpretation' of the machine's output.

2. The λσ-calculus Krivine machine is able to perform strong head reduction, as opposed to the weak head reduction supported by the original Krivine machine, but it cannot do so without some external help. Namely, if it halts with some configuration $(\lambda t, \sigma, \mathrm{nil})$ then it must be restarted with the configuration $(t, \underline{0} \cdot (\uparrow\,; \sigma), \mathrm{nil})$. In contrast, our machine will continue running until

the strong head normal form is reached, as it handles the above configuration with the 'Lambda' rule:

$$(\lambda t, \sigma, C) \to (t, \Uparrow(\sigma), \lambda\Box\,; C) \qquad \text{Lambda}$$

3. Our abstract machine is not only capable of performing strong head reduction, it may also be used to perform any *standard reduction* (Barendregt 1984), i.e. where β-redexes are reduced outside-in. We do this by introducing three additional rules. The first complements 'Left':

$$(tu, \sigma, C) \to (u, \sigma, (t[\sigma])\Box\,; C) \qquad \text{Right}$$

The remaining two allow us to swap from the left-hand side of an application to the right, and vice versa. We cannot rewind out of an application or abstraction, as that would result in a conflation of term and closure we have thus far avoided, but we can still swap from one side of an application to another.

$$(t, \sigma, \Box(u[\rho])\,; C) \to (u, \rho, (t[\sigma])\Box\,; C) \qquad \text{LeftRight}$$
$$(u, \rho, (t[\sigma])\Box\,; C) \to (t, \sigma, \Box(u[\rho])\,; C) \qquad \text{RightLeft}$$

The addition of these rules does however lead to nondeterminism unless we impose some reduction strategy, i.e. a preference on which rule to apply in a given circumstance. For head reduction we can of course simply ignore these rules, but we will now suggest one example where a more nuanced choice is indeed preferable, even if all we want is a head normal form.

4. The original Krivine machine has a space leak whereby a term like $(\lambda\underline{00})(\lambda\underline{00})$ will cause the stack and environments to grow unboundedly, even though the head reduction of this term should require only constant space (Wand 2003). The λσ-calculus Krivine machine inherits this behaviour. Wand proposes that this be remedied by a special rule when the right-hand side is a variable:

$$(t\underline{n}, \sigma, S) \to (t, \sigma, \sigma(\underline{n}) \cdot S)$$

Note, however, that this operation $\sigma(\underline{n})$ requires that we be able to apply the substitution σ to a given De Bruijn index using a mechanism auxiliary to the Krivine machine's own substitutive mechanisms. We might then just as well perform all substitutions through this auxiliary mechanism.

In contrast, our abstract machine is capable of switching over to perform the substitution of a variable on the right-hand side of an application, before switching back to head reduction. Namely, we favour 'Right' over 'Left' iff the right-hand side of the application is a variable, and favour 'RightLeft' over all other rules (including 'Right') iff the current term is not a variable. These two rules may then be thought of as 'pushing' and 'popping' a right-hand variable lookup, respectively.

We have, incidentally, also developed an implementation of this abstract machine in the dependently-typed programming language Agda, which verifies the correctness of its manipulation of De Bruijn indices when β-reducing an input λ-term.

Bibliography

Abadi, M., Cardelli, L., Curien, P.-L. & Lévy, J.-J. (1991), 'Explicit substitutions', *Journal of Functional Programming* **1**, 375–416.

Abramsky, S. (1993), 'Computational interpretations of linear logic', *Theoretical Computer Science* **111**, 3–57.

Accattoli, B. (2011), Jumping around the box: Graphical and operational studies on λ-calculus and Linear Logic, PhD thesis, Sapienza Università di Roma.

Accattoli, B., Barenbaum, P. & Mazza, D. (2015), A strong distillery, *in* 'Proceedings of the 13th Asian Symposium on Programming Languages and Systems', Springer, pp. 231–250.

Asperti, A. & Guerrini, S. (1998), *The Optimal Implementation of Functional Programming Languages*, Vol. 45 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press.

Asperti, A. & Laneve, C. (1994), 'Interaction Systems, I: The theory of optimal reductions', *Mathematical Structures in Computer Science* **4**, 457–504.

Asperti, A. & Laneve, C. (1995), Comparing λ-calculus translations in sharing graphs, *in* 'Typed Lambda Calculi and Applications', Springer, pp. 1–15.

Balabonski, T. (2013), 'Weak optimality, and the meaning of sharing', *ACM SIGPLAN Notices* **48**, 263–274.

Banach, R. (1995), 'Sequent reconstruction in LLM – a sweepline proof', *Annals of Pure and Applied Logic* **73**, 277–295.

Barber, A. & Plotkin, G. (1997), Dual intuitionistic linear logic.

Barendregt, H. P. (1984), *The Lambda Calculus, its Syntax and Semantics*, Vol. 40 of *Studies in Logic and the Foundations of Mathematics*, North-Holland.

Benton, N., Bierman, G., de Paiva, V. & Hyland, M. (1992), Term assignment for intuitionistic linear logic, Technical report, University of Cambridge.

Benton, N. & Wadler, P. (1996), Linear logic, monads and the lambda calculus, *in* 'Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science', IEEE, pp. 420–431.

Bierman, G. (1998), Multiple modalities, Technical Report 455, University of Cambridge.

Bourbaki, N. (1954), *Éléments de mathématiques: Théories des ensembles*, Hermann.

Church, A. (1940), 'A formulation of the simple theory of types', *The Journal of Symbolic Logic* **5**, 56–68.

Damas, L. & Milner, R. (1982), Principal type-schemes for functional programs, *in* 'Proceedings of the 9th ACM SIGPLAN–SIGACT symposium on Principles of Programming Languages', ACM, pp. 207–212.

De Bruijn, N. G. (1972), 'Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser theorem', *Indagationes Mathematicae* **75**, 381–392.

Fernández, M. & Mackie, I. (1996), From term rewriting to generalised interaction nets, *in* 'Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs', Springer, pp. 319–333.

Fernández, M. & Mackie, I. (1998), Coinductive techniques for operational equivalence of interaction nets, *in* 'Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science', IEEE, pp. 321–332.

Fernández, M., Mackie, I. & Pinto, J. S. (2007), 'A higher-order calculus for graph transformation', *Electronic Notes in Theoretical Computer Science* **72**, 45–58.

Fernández, M., Mackie, I. & Sinot, F.-R. (2005), 'Closed reduction: explicit substitutions without α-conversion', *Mathematical Structures in Computer Science* **15**, 343–381.

Girard, J.-Y. (1972), Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur, PhD thesis, Université Paris VII.

Girard, J.-Y. (1987), 'Linear logic', *Theoretical Computer Science* **50**, 1–102.

Girard, J.-Y. (1989), 'Geometry of Interaction, I: Interpretation of System F', *Studies in Logic and the Foundations of Mathematics* **127**, 221–260.

Gonthier, G., Abadi, M. & Lévy, J.-J. (1992), The geometry of optimal lambda reduction, *in* 'Proceedings of the 19th ACM SIGPLAN–SIGACT symposium on Principles of Programming Languages', ACM, pp. 15–26.

Guerrini, S. (1996), Theoretical and Practical Issues of Optimal Implementations of Functional Languages, PhD thesis, Università di Pisa.

Hardin, T. & Lévy, J.-J. (1989), A confluent calculus of substitutions, *in* 'France–Japan Artificial Intelligence and Computer Science Symposium', Vol. 106.

Huet, G. (1997), 'Functional pearl: The zipper', *Journal of Functional Programming* **7**, 549–554.

Huet, G. & Lévy, J.-J. (1991), Computations in orthogonal rewriting systems, I and II, *in* 'Computational Logic: Essays in honor of Alan Robinson', MIT Press, pp. 396–443.

Jacobs, B. (1994), 'Semantics of weakening and contraction', *Annals of Pure and Applied Logic* **69**, 73–106.

Johansson, I. (1936), 'Der minimalkalkül, ein reduzierter intuitionischer formalismus', *Compositio Mathematica* **4**, 119–136.

Kahl, W. (1998), 'Relational treatment of term graphs with bound variables', *Logic Journal of IGPL* **6**, 259–303.

Kahrs, S. (1994), Compilation of Combinatory Reduction Systems, *in* 'Higher-Order Algebra, Logic, and Term Rewriting', Springer, pp. 169–188.

Kahrs, S. (1996), The variable containment problem, *in* 'Higher-Order Algebra, Logic, and Term Rewriting', Springer, pp. 109–123.

Kahrs, S. & Smith, C. (2014), Normal forms and infinity, *in* 'Workshop of Infinitary Rewriting'.

Kahrs, S. & Smith, C. (2016), Non-ω-overlapping TRSs are UN, *in* 'Proceedings of the 1st International Conference on Formal Structures for Computation and Deduction'.

Kathail, V. (1990), Optimal Interpreters for Lambda-Calculus Based Functional Languages, PhD thesis, Massachusetts Institute of Technology.

Klop, J. W. (1980), Combinatory Reduction Systems, PhD thesis, Centrum Wiskunde & Informatica.

Klop, J. W., Van Oostrom, V. & Van Raamsdonk, F. (1993), 'Combinatory Reduction Systems: Introduction and survey', *Theoretical Computer Science* **121**, 279–308.

Krivine, J.-L. (2007), 'A call-by-name lambda-calculus machine', *Higher-Order and Symbolic Computation* **20**, 199–207.

Lafont, Y. (1990), Interaction Nets, *in* 'Proceedings of the 17th ACM SIGPLAN–SIGACT symposium on Principles of Programming Languages', ACM, pp. 95–108.

Lamping, J. (1990), An algorithm for optimal lambda calculus reduction, *in* 'Proceedings of the 17th ACM SIGPLAN–SIGACT symposium on Principles of Programming Languages', ACM, pp. 16–30.

Landin, P. J. (1966), 'The next 700 programming languages', *Communications of the ACM* **9**, 157–166.

Laneve, C. (1993), Optimality and Concurrency in Interaction Systems, PhD thesis, Università di Pisa.

Lawall, J. L. & Mairson, H. G. (1996), 'Optimality and inefficiency: What isn't a cost model of the lambda calculus?', *ACM SIGPLAN Notices* **31**, 92–101.

Lévy, J.-J. (1980), 'Optimal reductions in the lambda-calculus', *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* pp. 159–191.

Loader, R. (2003), 'Higher order β-matching is undecidable', *Logic Journal of the IGPL* **11**, 51–68.

Miller, D. (1991), 'A logic programming language with lambda-abstraction, function variables, and simple unification', *Journal of Logic and Computation* **1**, 497–536.

Nipkow, T. (1991), Higher-order critical pairs, *in* 'Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science', IEEE, pp. 342–349.

Okasaki, C. (1998), 'Even higher-order functions for parsing', *Journal of Functional Programming* **8**, 195–199.

Perlis, A. (1982), 'Epigrams on programming', *ACM SIGPLAN Notices* **17**, 7–13.

Pfenning, F. & Elliot, C. (1988), 'Higher-order abstract syntax', *ACM SIGPLAN Notices* **23**, 199–208.

Plump, D. (1993), Evaluation of Functional Expressions by Hypergraph Rewriting, PhD thesis, Universität Bremen.

Sannella, D. & Tarlecki, A. (2012), *Foundations of algebraic specification and formal software development*, Springer.

Seely, R. A. G. (1989), Linear logic, *-autonomous categories, and cofree coalgebras, *in* 'Proceedings of the AMS Conference on Categories in Computer Science and Logic', AMS.

Smith, C. (2014), Abstract machines for higher-order term sharing, *in* 'Preproceedings of the 26th Symposium on the Implementation and Application of Functional Languages'.

Stirling, C. (2009), 'Decidability of higher-order matching', *Logical Methods in Computer Science* **5**.

Terese (2003), *Term Rewriting Systems*, Vol. 55 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press.

Thatte, S. (1987), 'A refinement of strong sequentiality for term rewriting with constructors', *Information and Computation* **72**, 46–65.

Van Oostrom, V. (1994), Confluence for Abstract and Higher-Order Rewriting, PhD thesis, Vrije Universiteit Amsterdam.

Van Oostrom, V. (1996), Higher-order families, *in* 'Proceedings of the 7th International Conference on Rewriting Techniques and Applications', Springer, pp. 392–407.

Van Oostrom, V. & Van Raamsdonk, F. (1994), Comparing Combinatory Reduction Systems and Higher-order Rewrite Systems, *in* 'Proceedings of the 1st International Workshop on Higher-order Algebra, Logic, and Term Rewriting', Springer, pp. 276–304.

Van Raamsdonk, F. (1996), Confluence and Normalisation for Higher-Order Rewriting, PhD thesis, Vrije Universiteit Amsterdam.

Wadler, P. (1991), There's no substitute for linear logic.

Wadler, P. (1993), A syntax for linear logic, *in* 'Mathematical foundations of programming semantics', pp. 513–529.

Wadsworth, C. P. (1971), Semantics and Pragmatics of the Lambda-Calculus, PhD thesis, University of Oxford.

Walker, D. (2005), Substructural type systems, *in* B. C. Pierce, ed., 'Advanced Topics in Types and Programming Languages', MIT Press, pp. 3–43.

Wand, M. (2003), On the correctness of the Krivine machine.

Wolfram, D. A. (1990), The Clausal Theory of Types, PhD thesis, University of Cambridge.