Λ-Symsym: An Interactive Tool for Playing with Involutions and Types

Furio Honsell
University of Udine, Italy

Marina Lenisa
University of Udine, Italy

Ivan Scagnetto
University of Udine, Italy

Abstract

We present the web portal **Λ-symsym**, available at `http://158.110.146.197:31780/automata/`, for experimenting with *game semantics* of $\lambda^!$-calculus, and its *normalizing elementary sub-calculus*, the λ^{EAL}-calculus. The $\lambda^!$-calculus is a generalization of the λ-calculus obtained by introducing a modal operator !, giving rise to a *pattern* β-reduction. Its sub-calculus corresponds to an applicatively closed class of terms normalizing in an elementary number of steps, in which all *elementary functions* can be encoded. The game model which we consider is the *Geometry of Interaction* model $\mathcal{I}$ introduced by Abramsky to study reversible computations, consisting of *partial involutions* over a very simple language of moves.

Given a $\lambda^!$- or a λ^{EAL}-term, M, **Λ-symsym** provides:

- an abstraction algorithm $\mathcal{A}^!$, for compiling M into a term, $\mathcal{A}^!(M)$, of the *linear combinatory logic* $\mathbf{CL}^!$, or the normalizing combinatory logic $\mathbf{CL}^{EAL}$;
- an interpretation algorithm $[\![\ ]\!]^{\mathcal{I}}$ yielding a specification of the partial involution $[\![\mathcal{A}^!(M)]\!]^{\mathcal{I}}$ in the model $\mathcal{I}$;
- an algorithm, $\mathcal{I}2\mathcal{T}$, for synthesizing from $[\![\mathcal{A}^!(M)]\!]^{\mathcal{I}}$ a type, $\mathcal{I}2\mathcal{T}([\![\mathcal{A}^!(M)]\!]^{\mathcal{I}})$, in a *multimodal, intersection type assignment discipline*, $\vdash_!$.
- an algorithm, $\mathcal{T}2\mathcal{I}$, for synthesizing a specification of a partial involution from a type in $\vdash_!$, which is an inverse to the former.

We conjecture that $\vdash_! M : \mathcal{I}2\mathcal{T}([\![\mathcal{A}^!(M)]\!]^{\mathcal{I}})$. **Λ-symsym** permits to investigate experimentally the fine structure of $\mathcal{I}$, and hence the game semantics of the $\lambda^!$- and λ^{EAL}-calculi. For instance, we can easily verify that the model $\mathcal{I}$ is a $\lambda^!$-*algebra* in the case of *strictly linear* λ-terms, by checking all the necessary equations, and find counterexamples in the general case.

We make this tool available for readers interested to *play* with games (-semantics). The paper builds on earlier work by the authors, the type system being an improvement.

2012 ACM Subject Classification Theory of computation → Linear logic

Keywords and phrases game semantics, lambda calculus, involutions, linear logic, implicit computational complexity

Digital Object Identifier 10.4230/LIPIcs.TYPES.2020.7

Supplementary Material *Software (Tool)*: `http://158.110.146.197:31780/automata/`

Acknowledgements The authors would like to thank the anonymous referees for their helpful comments.

1 Introduction

We present the web portal **Λ-symsym**, available at `http://158.110.146.197:31780/automata/`, for experimenting, in the spirit of [24], with the *game semantics* of the $\lambda^!$-calculus, and its *normalizing elementary affine sub-calculus*, the λ^{EAL}-calculus. The $\lambda^!$-calculus is a generalization of the λ-calculus obtained by introducing a co-monadic modal operator !, which gives rise to a *pattern* β-reduction. Its sub-calculus corresponds to an applicatively closed class of terms normalizing in an elementary number of steps. All *elementary functions*

can be encoded in the λ^{EAL}-calculus, *cf* [20, 6]. The game model that we consider consists of *partial involutions*, *i.e. partial symmetric functions*. These were introduced by Abramsky in [3] to provide a *Geometry of Interaction* model, $\mathcal{I}$, of reversible computation for *linear combinatory logic*, see also [4, 5].

In a series of papers by the authors, [12, 10, 11, 18], we have explored and extended Abramsky's model $\mathcal{I}$. In particular, we have introduced:

- a generalized co-monadic λ-calculus, the $\lambda^!$-calculus, which subsumes also the *elementary affine* and *linear affine* calculi [26, 7, 6, 15, 16];
- *abstraction algorithms* for compiling the above calculi in corresponding fragments of an extended linear combinatory logic, $\mathbf{CL}^!$ or the normalizing $\mathbf{CL}^{EAL}$;
- a semantics of these combinatory logics in terms of partial involutions, and hence a semantics of the $\lambda^!$-calculus via the abstraction algorithm;
- a *multi-modal intersection type systems* $\vdash_!$ for the $\lambda^!$-calculus;
- an algorithm $\mathcal{T}2\mathcal{I}$ for obtaining a specification of a partial involution, given a type in the language of $\vdash_!$, and an algorithm $\mathcal{I}2\mathcal{T}$ for synthesizing a type given a specification of a partial involution, which are "morally" *mutual* inverses.

Moreover, we have shown that *linear application* between involutions, *i.e.* the game analogue of Girard's "Execution Formula", amounts to a kind of *resolution* between types in $\vdash_!$. In particular, in the case of strictly linear terms M, N, we have that $\mathcal{I}2\mathcal{T}([\![MN]\!]^{\mathcal{I}}) = \mathcal{I}2\mathcal{T}([\![M]\!]^{\mathcal{I}} \cdot [\![N]\!]^{\mathcal{I}}) = \mathcal{R}es(\mathcal{I}2\mathcal{T}([\![M]\!]^{\mathcal{I}}), \mathcal{I}2\mathcal{T}([\![N]\!]^{\mathcal{I}}))$, [12]. In this perspective, types deriving from $\mathcal{I}2\mathcal{T}$ can be viewed as a kind of *principal types* for terms in the $\lambda^!$-calculus.

In this paper, we outline the implementation of the algorithms above and present the on-line tool **Λ-symsym**[1] for experimenting with them in a user-friendly interactive setting, thus allowing for the investigation of the fine structure and the peculiarities of the symmetric game semantics of $\lambda^!$-calculus. The sheer combinatorial complexity of applying linear application by hand, in effect, makes it essential to use a machine for checking equalities and finding counterexamples.

Λ-symsym easily provides evidence for the following facts.

- The model $\mathcal{I}$ of partial involutions does not satisfy any form of the ξ-rule, if the terms involved are not *strictly linear*. Namely, we can check all the necessary equations for $\mathcal{I}$ to be a $\lambda^!$-*algebra* in the *strictly linear* case and provide counterexamples otherwise.
- Partial involutions keep track of the history of the execution, such as the erasures of arguments or the comonad !.

In particular, given a normalizing $\lambda^!$-term, or a λ^{EAL}-term, M, the web tool provides a web application for compiling M into a term of *linear combinatory logic*, $\mathcal{A}^!(M)$ or $\mathcal{A}^{EAL}(M)$, respectively. Another web application provides the interpretation of $\mathcal{A}^!(M)$ as a specification, $[\![\mathcal{A}^!(M)]\!]^{\mathcal{I}}$, of a *partial involution* over a very simple language of moves, in the model $\mathcal{I}$ of [3]. Yet another application permits to synthesize from $[\![\mathcal{A}^!(M)]\!]^{\mathcal{I}}$ a type, $\mathcal{I}2\mathcal{T}([\![\mathcal{A}^!(M)]\!]^{\mathcal{I}})$, in a *multimodal, intersection type discipline*. The type assignment system is given in the form of a general type system $\vdash_!$, for which we conjecture that $\vdash_! M : \mathcal{I}2\mathcal{T}([\![\mathcal{A}^!(M)]\!]^{\mathcal{I}})$. In this paper we do not discuss the other web applications made available by the tool **Λ-symsym** for dealing with the other $\lambda^!$-calculi of [18], and non normalizing terms.

We wrote this paper to encourage interested readers to enjoy utilizing **Λ-symsym**. We made it available for people to play with games (-semantics)!

---

[1] We combine Λ to the magic word of the formula *Open Sesame*, افتح يا سمسم (*iftah ya symsym*), in the Arab collection of novels *1001 Nights*, ألف ليلة وليلة (*kitab alf laylah wa-laylah*), which reminds that our semantics are symmetric partial functions.

Synopsis. In Section 2, we introduce in a self-contained format the formal systems used in the web tool, namely the $\lambda^!$-calculus, and the λ^{EAL}-calculus, the linear combinatory logics $\mathbf{CL}^!$ and $\mathbf{CL}^{EAL}$, !-intersection-types, and the $\vdash_!$ type discipline. In Subsection 2.1 we give the corresponding abstraction algorithms. In Section 3, we present the game theoretic semantics of the linear combinatory logic $\mathbf{CL}^!$, and hence of its normalizing sublogic $\mathbf{CL}^{EAL}$, in terms of partial involutions. We introduce also the crucial concept of specification of an involution. In Subsection 3.1, we present the two algorithms $\mathcal{I}2\mathcal{T}$ and $\mathcal{T}2\mathcal{I}$, for mapping specifications of partial involutions into types and back. In Section 4, we illustrate the web tool **Λ-symsym** and present its web interface. In Subsection 4.2 we discuss remarkable example sessions. Finally, in Section 5, we draw some conclusions and outline future directions. In the Appendix A, we give the Erlang code corresponding to the implementation of linear application of involutions. The complete code of our web tool is available in [13].

2 The $\lambda^!$-calculus, Linear Combinatory Logic, !-intersection-types

In this section we recall the formal systems used in the web tool. Most of these appear already in earlier papers by the authors, see [12, 10, 11, 18], but for the definition of the type assignment system $\vdash_!$.

We start with the $\lambda^!$-calculus and its sub-calculus.

▶ **Definition 1** ($\lambda^!$-calculus, λ^{EAL}-calculus).

- *The language* $\mathbf{\Lambda}^!$ *of the* $\lambda^!$-calculus *is inductively defined from variables* $x, y, z, \ldots$ *and constants* $c, \ldots$, *and it is closed under the following formation rules:*

$$\frac{M \in \mathbf{\Lambda}^! \quad N \in \mathbf{\Lambda}^!}{MN \in \mathbf{\Lambda}^!}\ (app) \qquad \frac{M \in \mathbf{\Lambda}^!}{!M \in \mathbf{\Lambda}^!}\ (!)$$

$$\frac{M \in \mathbf{\Lambda}^! \quad \mathcal{O}^!_{\leq 1}(x, M)}{\lambda x.M \in \mathbf{\Lambda}^!}\ (\lambda) \qquad \frac{M \in \mathbf{\Lambda}^! \quad \mathcal{O}_{\geq 1}(x, M)}{\lambda !x.M \in \mathbf{\Lambda}^!}\ (\lambda !)$$

  where $FV(M)$ *denotes the set of free variables of* M, *and* $\mathcal{O}^{\#}_{?}(x, M)$ $(\mathcal{O}^{\#}_{?}(M))$, *for* $\#$ *denoting either* ! *or blank, and* ? *denoting* ≤ 1 *or* ≥ 1 *or no constraint, means that the variable* x *(the free variables of* M*) appears at most once* (≤ 1) *or at least once* (≥ 1) *in* M, *respectively, and, when* $\#$ *is not blank, that it cannot appear in the scope of* !.
  We denote by $\equiv$ *syntactical identity on* λ*-terms.*

- *The language* $\mathbf{\Lambda}^{EAL}$ *of the* λ^{EAL}-calculus *is the restriction of* $\mathbf{\Lambda}^!$ *obtained by considering rules (app), (*λ*), and the following versions of the rules (*$\lambda !$*) and (!):*

$$\frac{M \in \mathbf{\Lambda}^{EAL} \quad \mathcal{O}_{\geq 1}(x, M) \quad x \text{ is in the scope of a single } !}{\lambda !x.M \in \mathbf{\Lambda}^{EAL}} \qquad \frac{M \in \mathbf{\Lambda}^{EAL} \quad \mathcal{O}^!(M)}{!M \in \mathbf{\Lambda}^{EAL}}$$

- *The reduction rules of the* $\lambda^!$*-calculus include the restrictions of the standard* (β)*-rule and* (ξ)*-rule to linear abstractions, the* pattern-β-reduction *rule, which define the behaviour of the ! pattern abstraction operator, the corresponding (str!) structural rule, and the* $(\xi !)$ *rule, namely:*

$$(\beta)\ (\lambda x.M)N \to_! M[N/x] \qquad (\beta !)\ (\lambda !x.M)!N \to_! M[N/x]$$

$$(\xi)\ \frac{M \to_! N \quad \lambda x.M \in \mathbf{\Lambda}^!}{\lambda x.M \to_! \lambda x.N} \qquad (\xi !)\ \frac{M \to_! N \quad \lambda !x.M \in \mathbf{\Lambda}^!}{\lambda !x.M \to_! \lambda !x.N} \qquad (str!)\ \frac{M \to_! N}{!M \to_! !N}$$

$$(appl_L)\ \frac{M_1 \to_! M_1'}{M_1 M_2 \to_! M_1' M_2} \qquad (appl_R)\ \frac{M_2 \to_! M_2'}{M_1 M_2 \to_! M_1 M_2'}$$

  We denote by $\to_!^*$ *the reflexive and transitive closure of* $\to_!$.
  The reduction rules of the λ^{EAL}*-calculus are the restrictions of the rules of the* $\lambda^!$*-calculus to* $\mathbf{\Lambda}^{EAL}$*-terms. We denote by* $\to_{EAL}$ *the corresponding reduction relation.*

We define strong normal forms as the irreducible terms which do not contain *stuck redexes*, *i.e.* subterms of the shape $(!M)N$ or $(\lambda!x.M)P$, where P is irreducible and not a !-term.

▶ **Definition 2** (Strong Normal Forms). *The set* $SN^!$ *of* strong normal forms *is inductively defined as follows:*

$$\frac{M_i \in SN^! \quad \forall i \in \{1,\dots,n\}}{xM_1 \dots M_n \in SN^!} \qquad \frac{M \in SN^!}{!M \in SN^!}$$

$$\frac{M \in SN^! \quad \lambda x.M \in \Lambda^!}{\lambda x.M \in SN^!} \qquad \frac{M \in SN^! \quad \lambda!x.M \in \Lambda^!}{\lambda!x.M \in SN^!}$$

Of course many terms containing stuck redexes are typable in the typing system of Definition 6 below, and receive the appropriate involution semantics, *e.g.* $\lambda x.(\lambda!y.y)x$ behaves as $\lambda!x.x$, but these intricacies are not worthwhile considering for the purposes of this paper.

The involution semantics of terms in $\Lambda^!$ and in Λ^{EAL} will be given via a compilation in linear combinatory logic.

▶ **Definition 3** (Linear Combinatory Logics).

- *The language of the extended linear combinatory logic* $\mathbf{CL}^!$ *consists of variables* $x, y, \dots$, *distinguished constants (combinators)* $\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{I}, \boldsymbol{K}, \mathbf{W}, \mathbf{D}, \delta, \mathbf{F}$, *and it is closed under application and* !*-promotion, i.e.:*

$$\frac{M \in \mathbf{CL}^! \quad N \in \mathbf{CL}^!}{MN \in \mathbf{CL}^!} \qquad \frac{M \in \mathbf{CL}^!}{!M \in \mathbf{CL}^!}$$

  In $\mathbf{CL}^!$*-terms, we associate* $\cdot$ *to the left and we assume ! to have order of precedence greater than* $\cdot$*. Combinators reduce according to the following* pattern *reduction rules:*

$$\boldsymbol{B}MNP \to_! M(NP) \qquad \boldsymbol{I}M \to_! M \qquad \boldsymbol{C}MNP \to_! (MP)N \qquad \boldsymbol{K}MN \to_! M$$
$$\boldsymbol{W}M!N \to_! M!N!N \qquad \delta!M \to_! !!M \qquad \boldsymbol{D}!M \to_! M \qquad \boldsymbol{F}!M!N \to_! !(MN)$$
$$\frac{M \to_! M'}{C[M] \to_! C[M']} \quad \textit{for any context } C[\,],$$

  where one-hole contexts are defined as usual.
- *The language of* elementary affine combinatory logic $\mathbf{CL}^{EAL}$ *includes only the combinators* $\boldsymbol{B}, \boldsymbol{I}, \boldsymbol{C}, \boldsymbol{K}, \boldsymbol{W}, \boldsymbol{F}$, *and is closed under the rules of application and !-promotion. The reduction relation restricted to* $\mathbf{CL}^{EAL}$ *is denoted by* $\to_{EAL}$.

We use the notation ***Comb*** *to denote any of the combinators in the above definition.*

We recall the following proposition because it is not so immediate:

▶ **Proposition 4** (Normalization [18]). *The* λ^{EAL}*-calculus and* $\boldsymbol{CL}^{EAL}$ *linear combinatory logic are strongly normalizing.*

The type discipline introduced in this section is a refinement of the ones introduced in [10, 12, 11] and originated in the process of relating *partial involutions* and *principal types*. It amounts essentially to a *multimodal intersection type discipline*, which generalizes [8].

▶ **Definition 5** (Type Language). *The language of types,* $Type^!$*, is a two sorted language given by the following grammars, where* ω *is a constant, and* α, β, ... *and* i, j, ... *are type and index variables respectively:*

$(Type^! \ni)\ \sigma, \tau \ ::= \ \omega \mid \alpha \mid \dots \mid \sigma \multimap \tau \mid \widehat{\sigma}$

$(\widehat{Type^!} \ni)\ \widehat{\sigma}, \widehat{\tau} \ ::= \ !_u\sigma \mid \widehat{\sigma} \wedge \widehat{\tau}$

$u, v ::= \epsilon \mid i \mid j \mid \dots \mid \langle u, v \rangle.$

The syntactic category $\widehat{\sigma}$*-types isolates types whose main constructor is* ! *or* $\wedge$*. Moreover, we consider the equivalence relation on types* $\sim$ *induced by* $\omega \sim \sigma$*, for any type* σ *which contains only the constant* ω *and no type variables.*

▶ **Definition 6** (Type System)**.**
The type system $\vdash_!$ *derives judgements* $\Gamma \vdash_! M : \tau$*, where the environment* Γ *is a set of pairs of the following shapes:* $x : \sigma$*,* $!x : \widehat{\sigma}$*; each variable occurs at most once in* Γ *either as a linear variable or a* $!$*-variable.* $dom(\Gamma)$ *denotes the set of variables which appear in* Γ*.*

Assumption rule:

$$\frac{}{x : \alpha \vdash_! x : \alpha} \ (Var)$$

Introduction and elimination rules:

$$\frac{\Gamma \vdash_! \ M : \sigma \quad i \text{ fresh}}{\widehat{!_i}(\Gamma) \vdash_! \ !M :!_i \sigma} \ (!Intro) \qquad \frac{\Gamma \vdash_! \ !M :!_i \sigma \quad !(\Gamma)}{\widehat{!_{[\epsilon/i]}}(\Gamma) \vdash_! \ M : \sigma[\epsilon/i]} \ (!\text{-}Elim)$$

$$\frac{\Gamma_1 \vdash_! \ !M : \widehat{\sigma} \quad \Gamma_2 \vdash_! \ !M : \widehat{\tau} \quad !(\Gamma_1) \quad !(\Gamma_2)}{\Gamma_1 \wedge \Gamma_2 \vdash_! \ !M : \widehat{\sigma} \wedge \widehat{\tau}} \ (\wedge\text{- } Intro)$$

$$\frac{\Gamma \vdash_! \ M : \widehat{\sigma} \wedge \widehat{\tau}}{\Gamma \vdash_! \ M : \widehat{\sigma}} \ (\wedge\text{-}E\text{-}left) \qquad \frac{\Gamma \vdash_! \ M : \widehat{\sigma} \wedge \widehat{\tau}}{\Gamma \vdash_! \ M : \widehat{\tau}} \ (\wedge\text{-}E\text{-}right)$$

$$\frac{\Gamma_1 \vdash_! \ M : \sigma \multimap \tau \quad \Gamma_2 \vdash_! \ N : \sigma \quad \bigcap_!(\Gamma_1, \Gamma_2)}{\Gamma_1 \wedge \Gamma_2 \vdash_! MN : \tau} \ (MP)$$

$$\frac{\Gamma, x : \sigma \vdash_! \ M : \tau}{\Gamma \vdash_! \ \lambda x.M : \sigma \multimap \tau} \ (\lambda) \qquad \frac{\Gamma, !x : \widehat{\sigma} \vdash_! M : \tau}{\Gamma \vdash_! \lambda !x.M : \widehat{\sigma} \multimap \tau} \ (\lambda!)$$

$$\frac{\Gamma \vdash_! \ M : \sigma}{U(\Gamma) \vdash_! \ M : U(\sigma)} \ (Inst)$$

Structural-rules:

$$\frac{\Gamma \vdash_! \ M : \sigma \quad x \notin dom(\Gamma)}{\Gamma, x : \tau \vdash_! \ M : \sigma} \ (weak) \qquad \frac{\sigma \sim \omega}{\Gamma \vdash_! M : \sigma} \ (\omega)$$

where

- $!(\Gamma)$ *means that all variables in* Γ *are banged (*Γ *is possibly empty);*
- *by a slight abuse of notation*

$$\widehat{!_u}(\Gamma, x : \tau) = \widehat{!_u}(\Gamma), !x : \widehat{!_u}(\tau) \quad \text{and} \quad \begin{cases} \widehat{!_u}(\tau) = !_u \tau & \text{if } \tau \notin \widehat{Type^!} \\ \widehat{!_u}(!_v \tau) = !_{\langle u,v \rangle} \tau \\ \widehat{!_u}(\widehat{\tau} \wedge \widehat{\sigma}) = \widehat{!_u}(\widehat{\tau}) \wedge \widehat{!_u}(\widehat{\sigma}); \end{cases}$$

- $\widehat{!_{[\epsilon/i]}}(\Gamma, !x : \tau) = \widehat{!_{[\epsilon/i]}}(\Gamma), !x : \tau[\epsilon/i]$*;*
- $\bigcap_!(\Gamma_1, \Gamma_2)$ *means that each variable in the intersection of* $dom(\Gamma_1)$ *and* $dom(\Gamma_2)$ *is* $!$*-prefixed both in* Γ_1 *and* Γ_2*;*
- *for* Γ_1, Γ_2 *such that* $\bigcap_!(\Gamma_1, \Gamma_2)$*, the environment* $\Gamma_1 \wedge \Gamma_2$ *is defined by:*
  - *if* $x : \sigma \in \Gamma_1$ *or* $x : \sigma \in \Gamma_2$*, then* $x : \sigma \in \Gamma_1 \wedge \Gamma_2$
  - *if* $!x : \sigma \in \Gamma_1$ *and* $x \notin dom(\Gamma_2)$*, or* $!x : \sigma \in \Gamma_2$ *and* $x \notin dom(\Gamma_1)$*, then* $!x : \sigma \in \Gamma_1 \wedge \Gamma_2$
  - *if* $!x : \sigma_1 \in \Gamma_1$ *and* $!x : \sigma_2 \in \Gamma_2$*, then* $!x : \sigma_1 \wedge \sigma_2 \in \Gamma_1 \wedge \Gamma_2$*;*
- U *is a substitution, mapping type/index variables to types/indexes, respectively.*

We will denote by $Type^{!-}$ *and* $\vdash_!^-$ *the set of types without* ω *and the type system without the* ω*-rule, respectively.*
The type system $\vdash_{EAL}$ *is obtained by giving up the rule (!-Elim) and restricting the rules of* $\vdash_!$ *to* λ*-terms in* $\mathbf{\Lambda}^{EAL}$*.*

Some comments and remarks on the above definitions are in order:

- The type connective $\wedge$ is neither commutative nor idempotent nor associative.
- The notion of *type equivalence* $\sim$ defined above for the system of Definition 6 is a sharp restriction of the usual type equivalence on intersection types, *cf.* [8]. The traditional rule does not hold in the involution model of Section 3.
- Not all terms of $\mathbf{\Lambda}^!$ can be typed in $\vdash_!^-$, *i.e.* without the ω-rule, *e.g.*
  $\vdash_! (\lambda!x.x!x)(\lambda!x.x!x) : \omega$ $\qquad$ $\vdash_! \lambda!x!y.(\lambda!z.z)(xy) : (\alpha \multimap \omega) \to (\alpha \multimap \omega)$.
  The former is not normalizing, while in the latter pattern matching fails and hence it does not have a strong normal form. Notice however that the type which can be assigned to it records the fact that the two variables can be applied. The meaningfulness of such types is the reason for weakening the traditional definition of type equivalence $\sim$ as in the system of Definition 6. The issue is rather intricate and will be taken up in Section 4.2.2.
- Rule (Inst) could be made redundant by allowing arbitrary types in rule (Var) and arbitrary indices in rule (!Intro).
- One could consider also second or higher order (possibly recursive) type systems. Their typing strength is subsumed by the $\wedge$-rules.

A relevant class of types is that of *binary types*, which, as we will see in Section 3.1, arise naturally from specifications of partial involutions:

▶ **Definition 7.**
- *A type $\tau \in Type^!$ is* binary *if each type variable appears at most twice in τ.*
- *A judgement $\Gamma \vdash_! M : \tau$ is* binary *if each type variable occurs at most twice in it.*

A slight modification of Theorem 30 [11] allows to prove that:

▶ **Proposition 8.** *If $\Gamma \vdash_! M : \tau$ is derivable, then there exist a binary judgement $\Gamma' \vdash_! M : \tau'$ and a substitution U such that $U(\Gamma') = \Gamma$ and $U(\tau') = \tau$.*

2.1 Abstraction Algorithm

The abstraction algorithm for mapping terms of the $\lambda^!$-calculus into the linear combinatory algebra $\mathbf{CL}^!$ is a refinement of the standard abstraction algorithm, but this needs to be carefully spelled out so as to factor out as an abstraction algorithm also from λ^{EAL} to $\mathbf{CL}^{EAL}$.

▶ **Definition 9** (Abstraction Algorithms $\mathcal{A}^!$, $\mathcal{A}^{EAL}$)**.**
- *Let $\mathbf{\Lambda}^!_{\mathbf{CL}}$ denote the set of terms of the $\lambda^!$-calculus obtained by taking combinators as the constants of the calculus. We define $\mathcal{A}^! : \mathbf{\Lambda}^!_{\mathbf{CL}} \to \mathbf{CL}^!$ as in Figure 1, where $\#$ denotes the modality $!$ or no modality.*
- *Let $\mathbf{\Lambda}^{EAL}_{\mathbf{CL}}$ be the set of λ^{EAL}-terms obtained by taking $\mathbf{CL}^{EAL}$-combinators as constants. We define $\mathcal{A}^{EAL} : \mathbf{\Lambda}^{EAL}_{\mathbf{CL}} \to \mathbf{CL}^{EAL}$ as the restriction of $\mathcal{A}^!$ to terms in $\mathbf{\Lambda}^{EAL}_{\mathbf{CL}}$.*
- *Vice versa, we denote by $(\ )_\lambda : \mathbf{CL}^! \to \mathbf{\Lambda}^!$ the natural mapping of a $\mathbf{CL}$-term into a λ-term obtained by replacing, in place of each combinator, the corresponding $\mathbf{\Lambda}^!$-term as follows:*

$$(\mathbf{B})_\lambda = \lambda xyz.x(yz) \qquad (\mathbf{C})_\lambda = \lambda xyz.(xz)y \qquad (\mathbf{I})_\lambda = \lambda x.x \qquad (\mathbf{K})_\lambda = \lambda xy.x$$
$$(\mathbf{W})_\lambda = \lambda x!y.x!y!y \qquad (\mathbf{D})_\lambda = \lambda!x.x \qquad (\delta)_\lambda = \lambda!x.!!x \qquad (\mathbf{F})_\lambda = \lambda!x!y.!(xy).$$

The following theorem justifies the above definition. For ease of readability we will often use the notation $\lambda^*\#x_1 \dots \#x_n.M$, for $M \in \mathbf{CL}^!$, to denote $\mathcal{A}^!(\lambda\#x_1 \dots \#x_n.M)$, where $\#$ stands for either the modality $!$ or no modality (see [18] for more details).

$$\mathcal{A}^!(M) = \begin{cases} x & \text{if } M \equiv x \\ \mathbf{Comb} & \text{if } M \equiv \mathbf{Comb} \\ \mathbf{I} & \text{if } M \equiv \lambda x.x \\ \mathbf{F}(!I) & \text{if } M \equiv \lambda!x.!x \\ \mathbf{D} & \text{if } M \equiv \lambda!x.x \\ \mathbf{K}c & \text{if } M \equiv \lambda x.c \\ \mathbf{K}y & \text{if } M \equiv \lambda x.y \\ \mathbf{C}(\mathcal{A}^!(\lambda x.M_1))\mathcal{A}^!(M_2) & \text{if } M \equiv \lambda x.M_1M_2 \text{ and } x \in FV(M_1) \\ \mathbf{B}\mathcal{A}^!(M_1)\mathcal{A}^!(\lambda x.M_2) & \text{if } M \equiv \lambda x.M_1M_2 \text{ and } x \in FV(M_2) \\ \mathbf{K}(\mathcal{A}^!(M_1)\mathcal{A}^!(M_2)) & \text{if } M \equiv \lambda x.M_1M_2 \text{ and } x \notin FV(M_1M_2) \\ \mathcal{A}^!(\lambda\#x.\mathcal{A}^!(\lambda\#y.M_1)) & \text{if } M \equiv \lambda\#x.\lambda\#y.M_1 \\ \mathbf{W}(\mathbf{B}(\mathbf{C}\mathcal{A}^!(\lambda!x.M_1))(\mathcal{A}^!(\lambda!x.M_2))) & \text{if } M \equiv \lambda!x.M_1M_2 \text{ and } x \in FV(M_1),\, x \in FV(M_2) \\ \mathbf{B}\mathcal{A}^!(M_1)\mathcal{A}^!(\lambda!x.M_2) & \text{if } M \equiv \lambda!x.M_1M_2 \text{ and } x \notin FV(M_1), x \in FV(M_2) \\ \mathbf{C}\mathcal{A}^!(\lambda!x.M_1)\mathcal{A}^!(M_2) & \text{if } M \equiv \lambda!x.M_1M_2 \text{ and } x \in FV(M_1), x \notin FV(M_2) \\ \mathbf{F!}(\lambda x.\mathcal{A}^!(M_1)) & \text{if } M \equiv \lambda!x.!M_1 \text{ and } x \text{ occurs once in } M_1 \text{ and } \mathcal{O}^!(M_1) \\ \mathbf{W}\mathcal{A}^!(\lambda!y.\lambda!x.!M_1[y/x^1]) & \text{if } M \equiv \lambda!x.!M_1[x^1],\ x \text{ occurs in } M_1 \text{ more than once} \\ & \text{and } x^1 \text{ is the leftmost occurrence of } x \text{ in } M_1 \text{ and } \mathcal{O}^!(M_1) \\ \mathbf{B}(\mathbf{F}(!\mathcal{A}^!(\lambda!x.M_1)))\delta & \text{if } M \equiv \lambda!x.!M_1 \text{ and } \neg\mathcal{O}^!(M_1) \\ \mathcal{A}^!(M_1)\mathcal{A}^!(M_2) & \text{if } M = M_1M_2 \\ !\mathcal{A}^!(M_1) & \text{if } M = !M_1 \\ FAIL & \text{otherwise} \end{cases}$$

Figure 1 The Abstraction Algorithm $\mathcal{A}^! : \mathbf{\Lambda}^!_{\mathbf{CL}} \to \mathbf{CL}^!$.

▶ **Theorem 10** (Soundness of the Abstraction Algorithms $\mathcal{A}^!$ and $\mathcal{A}^{EAL}$).

- *Let* $M \in \mathbf{CL}^!$, *let* $\#M_1, \ldots, \#M_n \in \mathbf{\Lambda}^!$, $n \geq 0$, *then* $(\lambda^*\#x_1 \ldots \#x_n.M)\mathcal{A}^!(\#M_1) \ldots \mathcal{A}^!(\#M_n) \to^*_! M[\mathcal{A}^!(M_1)/x_1, \ldots, \mathcal{A}^!(M_n)/x_n]$.
- *Let* $M \in \mathbf{CL}^{EAL}$, *let* $\#M_1, \ldots, \#M_n \in \mathbf{\Lambda}^{EAL}$, $n \geq 0$, *then* $\mathcal{A}^{EAL}(M) \in \mathbf{\Lambda}^{EAL}$ *and moreover:* $(\lambda^*\#x_1 \ldots \#x_n.M)\mathcal{A}^{EAL}(\#M_1) \ldots \mathcal{A}^{EAL}(\#M_n) \to^*_! M[\mathcal{A}^{EAL}(M_1)/x_1, \ldots, \mathcal{A}^{EAL}(M_n)/x_n]$.

The abstraction algorithm in Fig. 1 is implemented in **Λ-symsym**, see Section 4.

3 Game Semantics for Combinatory Logics and λ-calculi

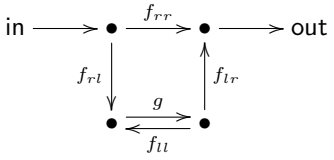
In this section, we present the *game theoretic* semantics of the linear combinatory logic $\mathbf{CL}^!$, and hence of its sublogic $\mathbf{CL}^{EAL}$, in terms of *partial involutions*. Then, via the abstraction algorithm, we obtain a game semantics for terms in $\Lambda^!$ and Λ^{EAL}. Finally, we present two algorithms, $\mathcal{I}2\mathcal{T}$ and $\mathcal{T}2\mathcal{I}$, which relate schematic representations of partial involutions and types.

We start by introducing the model of partial involutions, $\mathcal{I}$, originally defined by Abramsky in [3] in order to study reversible computations, and studied in [12, 11, 18]. The notion of application on this model amounts to the categorical trace on a subcategory of the category of relations, introduced in [23] (see also [2]).

▶ **Definition 11** (The Model of Partial Involutions, $\mathcal{I}$).

- T_Σ, *the language of* moves, *is defined by the signature* $\Sigma_0 = \{\epsilon\}$, $\Sigma_1 = \{l, r\}$, $\Sigma_2 = \{\langle\ ,\ \rangle\}$ *(where* Σ_i *is the set of constructors of arity* i*); terms* $r(t)$ *are* output words, *while terms* $l(t)$ *are* input words *(often denoted simply by* rt *and* lt*);*

- $\mathcal{I}$ *is the set of* partial involutions *over* T_Σ*, i.e. the set of all partial functions* $f : T_\Sigma \rightharpoonup T_\Sigma$ *such that* $f(t) = t' \Leftrightarrow f(t') = t$*;*
- *the operation of* replication *is defined by* $!f = \{(\langle t, t_1\rangle, \langle t, t_2\rangle) \mid t \in T_\Sigma \ \wedge (t_1, t_2) \in f\}$*;*
- *the notion of* linear application *is defined by* $f \cdot g = f_{rr} \cup (f_{rl}; g; (f_{ll}; g)^*; f_{lr})$*, where* $f_{ij} = \{(t_1, t_2) \mid (i(t_1), j(t_2)) \in f\}$*, for* $i, j \in \{r, l\}$ *(see diagram below), where ";" denotes postfix composition of relations.*



▶ **Proposition 12** ([3]). *$\mathcal{I}$ is closed under !-replication and linear application.*

Next we give the notion of *schematic specification* of the graph of a partial involution, which was introduced in [18] and recall some of its main properties. Schematic specifications induce a substructure of $\mathcal{I}$ which is still closed under application and replication, and it is sufficient to interpret all combinators, see [18] for full details.

▶ **Definition 13** (Schematic Specification). *Let $T_\Sigma[Var]$ denote the set of terms generated by the signature $\Sigma \setminus \{\epsilon\}$ starting from a set of variables Var, where we distinguish between* term *variables $x, y, \ldots$ and* index *variables $i, j, \ldots$, i.e.:*
$(T_\Sigma[Var] \ni)\ t \ ::= \ x \mid y \mid \ \ldots \ \mid lt \mid rt \mid \langle u, t\rangle \quad$ *where* $\quad u \ ::= \ \epsilon \mid i \mid j \mid \ \ldots \ \mid lu \mid ru \mid \langle u, v\rangle$ *.*
A specification *P of a partial involution is a (possibly infinite)* irreflexive *set of pairs $\{t_1 \leftrightarrow t'_1, t_2 \leftrightarrow t'_2, \ldots\}$, where:*

(i) *$t_i, t'_i \in T_\Sigma[Var]$;*
(ii) *a single pair $t_i \leftrightarrow t'_i$ represents both (t_i, t'_i) and (t'_i, t_i);*
(iii) *in any pair $t_i \leftrightarrow t'_i$ exactly one term variable appears, once in t_i and once in t'_i;*
(iv) *for any pairs $t_i \leftrightarrow t'_i$, $t_j \leftrightarrow t'_j$, t_i is not an instance of t_j.*

Schematic specifications generate graphs of partial involutions as follows:

▶ **Definition 14** ($\mathcal{G}$). *Given a schematic specification P, the set $\mathcal{G}(P)$ is the symmetric closure of the set $\{(U(t_1), U(t_2)) \mid (t_1 \leftrightarrow t_2) \in P \ \wedge \ U : T_\Sigma[Var] \to T_\Sigma\}$.*

The following lemma follows immediately from the above definitions:

▶ **Lemma 15.** *If P is a specification, then $\mathcal{G}(P)$ is the graph of a partial involution on T_Σ.*

Not all partial involutions are generated by specifications, *e.g.* the relation $\{r\epsilon \leftrightarrow l\epsilon\}$ is trivially not induced by any specification.

A notion of composition on specifications, which corresponds to composition on partial involutions can be defined as follows:

▶ **Definition 16** (Composition of Specifications). *Let P, P' be specifications. We define*

$$P; P' = \{(t, t') \mid \exists (t_1, t_2) \in P,\ (t'_1, t'_2) \in P',\ U.\ (U = \mathcal{U}(t_2, t'_1) \wedge t = U(t_1) \wedge t' = U(t'_2))\}\ ,$$

where $\mathcal{U}(t_2, t'_1)$ is the most general unifier of t_2 and t'_1.

This definition essentially amounts to composition in Clause Algebras, as in [19].

The above definitions permits to define a notion of application between specifications, $P \cdot P'$, which mimics the one between partial involutions in Definition 11.

$$
\begin{array}{lcl}
[\![\mathbf{B}]\!]^{\mathcal{I}} & = & \{r^3x \leftrightarrow lrx \ , \ l^2x \leftrightarrow rlrx \ , \ rl^2x \leftrightarrow r^2lx\} \\
[\![\mathbf{I}]\!]^{\mathcal{I}} & = & \{lx \leftrightarrow rx\} \\
[\![\mathbf{C}]\!]^{\mathcal{I}} & = & \{l^2x \leftrightarrow r^2lx \ , \ lrlx \leftrightarrow rlx \ , \ lr^2x \leftrightarrow r^3x\} \\
[\![\mathbf{K}]\!]^{\mathcal{I}} & = & \{lx \leftrightarrow r^2x\} \\
[\![\mathbf{F}]\!]^{\mathcal{I}} & = & \{l\langle i, rx\rangle \leftrightarrow r^2\langle i, x\rangle \ , \ l\langle i, lx\rangle \leftrightarrow rl\langle i, x\rangle\} \\
[\![\mathbf{W}]\!]^{\mathcal{I}} & = & \{r^2x \leftrightarrow lr^2x \ , \ l^2\langle i, x\rangle \leftrightarrow rl\langle li, x\rangle \ , \ lrl\langle i, x\rangle \leftrightarrow rl\langle ri, x\rangle\} \\
[\![\delta]\!]^{\mathcal{I}} & = & \{l\langle\langle i, j\rangle, x\rangle \leftrightarrow \ r\langle i, \langle j, x\rangle\rangle\} \\
[\![\mathbf{D}]\!]^{\mathcal{I}} & = & \{l\langle \epsilon, x\rangle \leftrightarrow rx\} \\
[\![MN]\!]^{\mathcal{I}} & = & [\![M]\!]^{\mathcal{I}} \cdot [\![N]\!]^{\mathcal{I}} \\
[\![!M]\!]^{\mathcal{I}} & = & ![\![M]\!]^{\mathcal{I}}
\end{array}
$$

Figure 2 Game Semantics for $\mathbf{CL}^!$.

▶ **Lemma 17** ([18]). *Let P, P' be specifications. Then*
(i) $\mathcal{G}(P; P') = \mathcal{G}(P); \mathcal{G}(P')$*;*
(ii) $\mathcal{G}(P \cdot P') = \mathcal{G}(P) \cdot \mathcal{G}(P')$.

One can easily check that:

▶ **Lemma 18** ([18]). *The substructure of $\mathcal{I}$ induced by schematic specifications is closed under application and replication.*

Now we are in the position of giving the interpretation in $\mathcal{I}$ of $\mathbf{CL}^!$, by defining the schematic specifications of the graphs of the combinators.

▶ **Definition 19** (Game Semantics of $\mathbf{CL}^!$). *The combinatory logic $\mathbf{CL}^!$ is interpreted in $\mathcal{I}$ as in Figure 2, where we use the abbreviation l^nt and r^nt for the terms $\underbrace{l \dots l}_{n} t$, and $\underbrace{r \dots r}_{n} t$.*

In following proposition for (i) see [3], for (ii) see [11].

▶ **Proposition 20.**
(i) *$\mathcal{I}$ is a model of $\mathbf{CL}^!$, and hence of the combinatory sublogic $\mathbf{CL}^{\mathbf{EAL}}$.*
(ii) *$\mathcal{I}$ is a model of all the reduction rules of $\mathbf{\Lambda}^!$, but the ξ-rules.*

Using the semantics for $\mathbf{CL}^!$-combinators defined in Figure 2, and the abstraction algorithm described in Section 2.1, one can derive the partial involution corresponding to a given λ-term, by repeatedly applying our implementation of linear application on involutions (see Appendix A). However, not all interpretations of $\mathbf{CL}^!$-combinators are partial involutions with *finite* specifications, since graphs of finite specifications are not closed under linear application. *E.g.* the fixed point combinator $(\lambda^*!x!y.y!(x!y!y!x))!(\lambda^*!x!y.y!(x!y!y!x))$ has an infinite specification. This is taken care of in **Λ-symsym** by outputting the stream of pairs in the partial specification.

3.1 Correspondence Algorithms, $\mathcal{I}2\mathcal{T}$, $\mathcal{T}2\mathcal{I}$

Here we present the two algorithms, $\mathcal{I}2\mathcal{T}$ and $\mathcal{T}2\mathcal{I}$, which relate schematic representations of partial involutions and types in $Type^!$. The first algorithm, given a schematic specification of a partial involution, yields a binary type, the latter, given a binary type, provides a schematic specification of a partial involution. The two transformations are one inverse of the another over suitable domains of schematic specifications and types.

We begin by explaining how the algorithm $\mathcal{I}2\mathcal{T}$ works. First of all, we need the following *build_type_tree*( ) function, in order to build a *skeleton* type tree from a term $t \in T_\Sigma[Var]$. The function definition is syntax driven:

$$
\begin{aligned}
&\mathit{build\_type\_tree}(t)\\
&= \begin{cases}
\alpha_x & \text{if } t \equiv x\\
\mathit{build\_type\_tree}(t') \to \_ & \text{if } t \equiv lt'\\
\_ \to \mathit{build\_type\_tree}(t') & \text{if } t \equiv t'\\
!_u \mathit{build\_type\_tree}(t') & \text{if } \tau \equiv \langle u, t' \rangle\\
 & \text{and } u \text{ does not contain } l/r \text{ constructors}\\
\mathit{build\_type\_tree}(\langle u', t' \rangle) \wedge \_ & \text{if } t \equiv \langle u, t' \rangle \text{ and } l \text{ is the leftmost } l/r \text{ constructor}\\
 & \text{in } u \text{ and } u' \text{ is obtained from } u \text{ by erasing it}\\
\_ \wedge \mathit{build\_type\_tree}(\langle u', t' \rangle) & \text{if } t \equiv \langle u, t' \rangle \text{ and } r \text{ is the leftmost } l/r \text{ constructor}\\
 & \text{in } u \text{ and } u' \text{ is obtained from } u \text{ by erasing it}
\end{cases}
\end{aligned}
$$

where the underscore characters (_) above stand for "missing parts" of the resulting type tree. Such holes will be (hopefully) filled in by successive unifications, as explained below.

Hence, for each rule $t_i \leftrightarrow t_i'$ (for $i = 1, \ldots, n$) of a given specification, we compute the two (partial) types $\mathit{build\_type\_tree}(t_i)$ and $\mathit{build\_type\_tree}(t_i')$. Finally, we try to unify all such partial types, over all rules. If we succeed, we infer a type representing the original partial specification; otherwise, we fail. Notice that the possible final type may still contain "holes". *E.g.* partial specifications of erasing terms do not exhibit any rule for the variable which is erased.

Due to the peculiar syntax of types in $Type^!$, unification of types is not a plain unification *à la* Robinson. Indeed, we have subcases which may need to spawn $\wedge$-constructors in order to instantiate new instances of some types. For instance, a !-subtype like $!_u\sigma$ may need to spawn new instances of $!_u\sigma$, becoming $!_{u'}\sigma' \wedge !_{u''}\sigma'' \wedge \ldots$, in order to unify against another type. In our implementation, we implement such spawnings in order to ease the unification process as much as possible. At the end there can be some leftovers of those spawning activities; thus, we implement a sort of "garbage collector" which takes care of those dummy $\wedge$'s, before returning the final type-candidate to the user.

Another subtle issue comes from the fact that the order in which the pairs of the specification are processed may imply success or failure in the unification process (because the spawning of $\wedge$-constructors must trigger at the right moment). Hence, we restart the procedure until a type is found, permuting in all possible ways the pairs in the specification.

Finally, the type synthesis algorithm may fail, totally or partially. Total failure can be easily detected, since the algorithm returns Ω, meaning that it was not able to build even a partial skeleton. On the other hand, partial failure occurs when the algorithm does not succeed in recovering a complete type tree from the partial specification, i.e., there remain some "holes" to be filled in. Such holes are marked by placeholders of the form Ω_i, where i is some integer index. Such Ω_i's correspond to the underscore characters generated by the *build_type_tree*() function above which have not been instantiated by the type unification algorithm. We could safely replace them with a plain Ω, since the type ω is a *wild card*, but we do not make this choice in the web tool, see Section 4.2.2.

The other way round is dealt with by the algorithm $\mathcal{T}2\mathcal{I}$. Here the input can be any type and the output is a specification of a symmetric relation; if the input type is binary, then the result is the specification of a partial involution. The algorithm processes all type variables occurring in a given type by means of the function *var_type*$2PI$, which produces a list L of partial specification terms. If the type is binary the list has at most length 2. The

partial specification rules for that type variable are obtianed by combining in all possible ways the terms of L. Repeating this procedure for all type variables of a type, we recover the corresponding partial specification. We give only $var\_type2PI$:

$$
var\_type2PI(\alpha,\tau) = \begin{cases}
[x_\alpha] & \text{if } \tau \equiv \alpha \\
l(var\_type2PI(\alpha,\tau')) ++ & \\
\quad r(var\_type2PI(\alpha,\tau'')) & \text{if } \tau \equiv \tau' \to \tau'' \\
 & \text{and } \alpha \in fv(\tau'), \alpha \in fv(\tau'') \\
l(var\_type2PI(\alpha,\tau')) & \text{if } \tau \equiv \tau' \to \tau'' \\
 & \text{and } \alpha \in fv(\tau') \setminus fv(\tau'') \\
r(var\_type2PI(\alpha,\tau'')) & \text{if } \tau \equiv \tau' \to \tau'' \\
 & \text{and } \alpha \in fv(\tau'') \setminus fv(\tau') \\
add\_index(u, var\_type2PI(\alpha, \bigwedge_{i=1}^{n} !_{v_i}\tau_i)) & \text{if } \tau \equiv \bigwedge_{i=1}^{n} !_{<u,v_i>}\tau_i \\
compose\_index(l, var\_type2PI(\alpha,\tau')) ++ & \\
\quad compose\_index(r, var\_type2PI(\alpha,\tau'')) & \text{if } \tau \equiv \tau' \wedge \tau'' \\
 & \text{and } \alpha \in fv(\tau'), \alpha \in fv(\tau'') \\
 & \text{and ! indices in } \tau \text{ have different } \pi_1\text{'s} \\
compose\_index(l, var\_type2PI(\alpha,\tau')) & \text{if } \tau \equiv \tau' \wedge \tau'' \\
 & \text{and } \alpha \in fv(\tau') \setminus fv(\tau'') \\
 & \text{and ! indices in } \tau \text{ have different } \pi_1\text{'s} \\
compose\_index(r, var\_type2PI(\alpha,\tau'')) & \text{if } \tau \equiv \tau' \wedge \tau'' \\
 & \text{and } \alpha \in fv(\tau'') \setminus fv(\tau') \\
 & \text{and ! indices in } \tau \text{ have different } \pi_1\text{'s} \\
add\_index(u, var\_type2PI(\alpha,\tau')) & \text{if } \tau \equiv !_u\tau' \text{ and } \alpha \in fv(\tau') \\
[] & \text{otherwise}
\end{cases}
$$

where:

- $fv(\tau)$ is the set of free variables of type τ;
- in the expression $\tau \equiv \bigwedge_{i=1}^{n} \tau_i$ the right-hand side term denotes the atomic components of τ;
- $++$ is the list concatenation operator;
- the functions $l()$ and $r()$, when applied to a list (i.e., the return type of $var\_type2PI()$), distribute, respectively the l and r constructors at the head of all elements of the list, i.e., each $t \in L$ will be substituted by lt (resp. rt);
- $compose\_index(c, L)$ (where c is either l or r) is a function distributing the constructor c to all the pairs of the list L, i.e., each $\langle u, t\rangle \in L$ will be substituted by $\langle cu, t\rangle$;
- $add\_index(u, t)$ is a function distributing the index u to all elements of the list L, *i.e.* each $tinL$ will be substituted by $\langle u, t\rangle$.

Notice that in dealing with types whose outermost constructor is $\wedge$, we distinguish between the cases where all atomic components have a common prefix in their !'s, denoting that promotions have been carried after the $\wedge$ operations, and the cases where there is no such common prefix, where the $\wedge$-constructor was indeed the last to be used in producing the type.

It is easy to check that:

▶ **Lemma 21.**

1. *For any specification P of a partial involution, $\mathcal{I}2\mathcal{T}(P)$ is a binary type.*
2. *For any binary type τ, $\mathcal{T}2\mathcal{I}(\tau)$ is a specification of a partial involution.*

The two transformations, $\mathcal{I}2\mathcal{T}$ and $\mathcal{T}2\mathcal{I}$, are the inverse of one another in the sense:

▶ **Conjecture 22.**

1. *Let $M \in \Lambda^!$ be a closed term which reduces to a term in strong normal form, without the use of the ξ-rules, then: $\vdash M : \mathcal{I}2\mathcal{T}([\![M]\!]^{\mathcal{I}})$ and $\mathcal{T}2\mathcal{I}(\mathcal{I}2\mathcal{T}([\![M]\!]^{\mathcal{I}})) = [\![M]\!]^{\mathcal{I}}$.*
2. *Let τ be a binary type, then $\mathcal{I}2\mathcal{T}(\mathcal{T}2\mathcal{I}(\tau)) = \tau$.*

When a closed term $M \in \Lambda^!$ needs the ξ-rules to reduce to strong normal form its semantics as a partial involution does not necessarily yield an immediately meaningful type. We will discuss this issue further in Section 5.

Conjecture 22 applies also when we restrict to terms of Λ^{EAL}.

4 The Web Tool Λ-symsym

In the following, we will illustrate how to use the web tool, **Λ-symsym**, available at `http://158.110.146.197:31780/automata/`, in order to experiment with the algorithms provided in the previous section. We will provide excerpts from various sessions illustrating how to convert $\lambda^!$-terms to combinators, from the latter to partial involutions, and finally to types. Moreover, we will also show how to infer a partial involution from a type. The web tool is implemented in the language Erlang, [17]. We chose this language because of its rich pattern matching features, its flexible constructors such as the *set data type* and for the large number of libraries which allow for a rapid prototyping. The fine details of the implementation appear in [13]

4.1 Encoding λ-terms, partial involutions and types

First we give the syntax representation; the following tables contain the correspondences between the syntax we use with "pencil and paper", the actual input syntax and the internal representation used by our algorithms.

λ-terms		
On the paper	Input syntax	Internal representation
x	`X`	`{var, "X"}`
C	`C`	`{comb, C}`
MN	`M@N`	`{lapp, M, N}`
$\lambda x.M$	`l*X.M`	`{abs, {var, "X"}, M}`
$\lambda!x.M$	`l*!X.M`	`{abs_b, {var, "X"}, M}`
$!M$	`!M`	`{bang, M}`

Partial Involutions		
On the paper	Input syntax	Internal representation
ϵ	`e`	`e`
x	`X`	`{var, "X"}`
lM	`lM`	`{l, M}`
rM	`rM`	`{r, M}`
$\langle M, N\rangle$	`<M,N>`	`{p, M, N}`
$M \leftrightarrow N$	`M <-> N`	`[{M,N},{N,M}]`

Types		
On the paper	Input syntax	Internal representation
α	`A`	`{var, "A"}`
$\alpha \longrightarrow \beta$	`A -> B`	`{map, A, B}`
$\alpha \wedge \beta$	`A /\ B`	`{cap, A, B}`
$!_i\alpha$	`!I M`	`{bang, I, M}`

Figure 3 The interface of the web tool.

4.1.1 The web interface

Figure 3 depicts the web interface we can use to call the algorithms. It has 4 components:

1. a green drop-down menu allowing the user to choose the function to execute (e.g., `lambda2type()` in the general setting, as it appears in Figure 3); it provides a number of functions for the general case, $\Lambda^!$ and for the Λ^{EAL} case, namely:
   - **(i)** the function `lambda2type()` which given a term produces its combinatory logic translation, its semantics as a specification of a partial involution, and the type corresponding to the specification;
   - **(ii)** the function `show_partial_involution` which produces a specification of a partial involution, given a type;
   - **(iii)** the function `show_type` which yields a type given a partial involution;
   - **(iv)** more functions which have not been discussed in this paper for dealing with polynomial terms, and Hindley-Milner Types;
2. a text area where the user can write the argument to pass to the selected function;
3. another text area where the result will appear after clicking the button "Compute";
4. a red button ("Compute") to start the computation.

4.2 Sessions

4.2.1 Strictly Linear Combinators in $\mathcal{I}$ are a λ-algebra

One can see that the partial involutions corresponding to the two terms in input are equal. This is one of the many equations which need to be checked to show that the ξ rule holds.

```
Input:
l*X.l*Y.l*Z.X@(Y@Z)
Combinators abstraction:
((C((BB)((BB)I)))((C((BB)I))I))
Partial involution rules:
--- begin ---
l(r(_X7)) <-> r(r(r(_X7)))
l(l(_X7)) <-> r(l(r(_X7)))
r(l(l(_X7))) <-> r(r(l(_X7)))
--- end ---
Principal type:
((_X10 -> _X9) -> ((_X8 -> _X10) -> (_X8 -> _X9)))
```

```
Input:
B
Partial involution rules:
--- begin ---
r(r(r(_X1))) <-> l(r(_X1))
l(l(_X1)) <-> r(l(r(_X1)))
r(l(l(_X1))) <-> r(r(l(_X1)))
--- end ---
Principal type:
((_X4 -> _X3) -> ((_X2 -> _X4) -> (_X2 -> _X3)))
```

4.2.2 The ξ-rule fails for erasers

The following example shows that the ξ-rule fails in $\mathcal{I}$ when terms erase variables. The session on the right hand side shows that erased subterms might still leave an echo in the partial involution. This is the reason why we have not taken the standard ω-equivalence

as for intersection types. Please bear in mind that, when an inferred type features some `_Omegai` variables (for `i=1,2,3...`), these variables stand for placeholders for any type. They are "holes" in the type tree structure which arise since partial involutions do not provide unnecessary information.

```
Input:
l*X.l*Y.((l*Z.I)@(X@Y))
Combinators abstraction:
((B(B(KI)))((C((BB)I))I))
Partial involution rules:
--- begin ---
r(r(l(_X4))) <-> r(r(r(_X4)))
l(l(_X4)) <-> r(l(_X4))
--- end ---
Principal type:
((_X5 -> _Omega3) -> (_X5 -> (_X6 -> _X6)))
```

```
Input:
l*X.l*Y.l*Z.Z
Combinators abstraction:
(K(KI))
Partial involution rules:
--- begin ---
r(r(l(_X2))) <-> r(r(r(_X2)))
--- end ---
Principal type:
(_Omega1 -> (_Omega2 -> (_X3 -> _X3)))
```

4.2.3 The ξ-rule fails when modalities are erased

As was the case in the above example, also in this case there remains an echo in the partial involution when modalities are erased.

```
Input:
l*!X.D@!X
Combinators abstraction:
((BD)(F!(I)))
Partial involution rules:
--- begin ---
l(<e, _X1>) <-> r(_X1)
--- end ---
Principal type:
((!e _X2) -> _X2)
```

```
Input:
l*!X.D@!(D@!X)
Combinators abstraction:
((BD)((B(F!(((BD)(F!(I))))))d))
Partial involution rules:
--- begin ---
l(<<e, e>, _X1>) <-> r(_X1)
--- end ---
Principal type:
((!<e, e> _X2) -> _X2)
```

4.2.4 The ξ-rule fails in general

The following sessions illustrate that the ξ-rule fails in the encoding of standard λ-calculus via application in the Kleisli category. This issue will be briefly discussed in Section 5.

```
Input:
l*!X.(l*!Y.Y@!Y)@!(X@!X)
Combinators abstraction:
((B(W((C((BB)D))(F!(I))))))((B(F!((W((C((BB)D))(F!(I)))))))d))
Partial involution rules:
--- begin ---
l(<<l(e), l(e)>, r(r(_X15))>) <-> r(_X15)
l(<<_X15, l(e)>, l(<_X16, _X17>)>) <-> l(<<_X15, r(_X16)>, _X17>)
l(<<l(e), l(e)>, r(l(<_X15, _X16>))>) <-> l(<<r(_X15), l(e)>, r(_X16)>)
--- end ---
Principal type:
(((!<e, e> (_Omega6 -> ((!_X17 _X18) -> _X21))) /\ (!<_X22, e> ((!_X19 _X20) -> _Omega10))) /\
                            /\ (((!<_X22, _X19> _X20) /\ (!<_X17, e> (_Omega4 -> _X18))) /\
                                                  /\(!<_X22, e> ((!_X19 _X20) -> _Omega10))))
                                                                                  -> _X21
```

but

```
Input:
l*!X.X@!X@!(X@!X)
Combinators abstraction:
(W((C((BB)(W((C((BB)D))(F!(I)))))))((B(F!((W((C((BB)D))(F!(I)))))))d)))
Partial involution rules:
--- begin ---
r(_X14) <-> l(<l(l(e)), r(r(_X14))>)
l(<l(l(e)), l(<_X14, _X15>)>) <-> l(<l(r(_X14)), _X15>)
l(<l(l(e)), r(l(<_X14, _X15>))>) <-> l(<r(<_X14, l(e)>), r(_X15)>)
l(<r(<_X14, l(e)>), l(<_X15, _X16>)>) <-> l(<r(<_X14, r(_X15)>), _X16>)
--- end ---
Principal type:
(((!e ((!_X20 _X21) -> ((!_X17 _X23) -> _X24))) /\ (!_X20 _X21))/\
              /\ ((!<_X17, e> ((!_X18 _X19) -> _X23)) /\ (!<_X17, _X18> _X19))) -> _X24
```

4.2.5 Church Binary Words in the λ^{EAL}-calculus

This is the encoding of Church binary word $\underline{< 0, 0, 1 >}$ as would appear in [6].

```
Input:
l*!X.l*!Y.!(l*Z.X@(X@(Y@Z)))
Combinators abstraction:
((BF)(W((BF)(F!((((C((BB)((BB)((BB)I)))))((C((BB)((BB)I)))((C((BB)I))I))))))))
Partial involution rules:
--- begin ---
l(<l(_X9), r(_X10)>) <-> r(r(<_X9, r(_X10)>))
l(<l(_X9), l(_X10)>) <-> l(<r(_X9), r(_X10)>)
l(<r(_X9), l(_X10)>) <-> r(l(<_X9, r(_X10)>))
r(r(<_X9, l(_X10)>)) <-> r(l(<_X9, l(_X10)>))
--- end ---
Principal type:
(((!_X11 (_X15 -> _X14)) /\ (!_X11 (_X16 -> _X15))) -> ((!_X11 (_X12 -> _X16)) -> (!_X11 (_X12 -> _X14))))
```

5 Final Remarks

In this paper we have introduced the web tool **Λ-symsym** which implements various algorithms for computing the game semantics, in terms of involutions, of $\lambda^!$- and λ^{EAL}-terms and their types. Using **Λ-symsym** we have uncovered many peculiarities of game semantics. The fact that in Abramsky's model, $\mathcal{I}$, the ξ-rule fails in the general case, but not in the strictly linear case is perhaps the most remarkable one. Actually the portal **Λ-symsym** makes available algorithms, similar to the ones introduced in this paper, for dealing with the λ^{LAL}-calculus, introduced by the authors in [18], which captures in our framework the polynomial calculi underpinning *Light Linear Logic*, *cf* [20, 26]. Furthermore, partial involutions easily generalize to *symmetric partial relations*, see [18]. In this broader setting one can model ML-types. **Λ-symsym** provides algorithms also for this case.

We list, in a cursory manner, a number of intriguing issues deserving future attention.

- There is a plausible duality between types in $\vdash^!$ and *partial involutions*, which should build up to a framework such as the one in [14, 1] for domain theory and intersection types, *cf* [9].
- Conjecture 22 appears to be very difficult to prove in full generality. A proof should relate two dual alternative ways of carrying out *unification*. In analogy to Abramsky's terminology, we could call the *wave-style* traditional unification and the *particle-style* unification, which goes on when linearly applying two specifications of involutions.
- The failure of the ξ-rule in the model $\mathcal{I}$ uncovers the fact that sometimes, rather than computing the *most general unifier* between the corresponding types, linear application of partial involutions stops short of that and just computes some sort of *least general ancestor* of the type. This has not been fully appreciated yet. In this respect, exploring the connections of the present work with Girard's Clause Algebras and the Execution Formula as in [19] should prove fruitful.
- The $\lambda^!$-calculus is worthwhile investigating *per se*, but many well established notions need to be generalized, most notably normal forms, because of stuck terms. The paper [21] should provide some suggestions.
- J.-Y. Girard in [20] introduced a *polynomial time set-theory*, which bears some relation to the one of Fitch as presented in [25], which we have studied in [22]. The experience gathered with λ^{LAL}-calculus should improve our understanding of its applicability.

In conclusion we think that the fine structure of game semantics of the λ-calculus needs more investigation. This could be quite rewarding, provided more concrete experiments are carried out. **Λ-symsym** is our contribution to making them a little more feasible.

References

1. Samson Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51(1):1–77, 1991. doi:10.1016/0168-0072(91)90065-T.
2. Samson Abramsky. Retracing some paths in process algebra. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory*, pages 1–17, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
3. Samson Abramsky. A structural approach to reversible computation. *Theoretical Computer Science*, 347(3):441–464, 2005. doi:10.1016/j.tcs.2005.07.002.
4. Samson Abramsky, Esfandiar Haghverdi, and Philip Scott. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science*, 12(5):625–665, 2002.
5. Samson Abramsky and Marina Lenisa. Linear realizability and full completeness for typed lambda-calculi. *Annals of Pure and Applied Logic*, 134(2):122–168, 2005. doi:10.1016/j.apal.2004.08.003.
6. Patrick Baillot, Erika De Benedetti, and Simona Ronchi Della Rocca. Characterizing Polynomial and Exponential Complexity Classes in Elementary Lambda-Calculus. In Josep Diaz, Ivan Lanese, and Davide Sangiorgi, editors, *Theoretical Computer Science*, pages 151–163, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
7. Patrick Baillot and Kazushige Terui. A Feasible Algorithm for Typing in Elementary Affine Logic. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications*, pages 55–70, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
8. Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *The Journal of Symbolic Logic*, 48(4):931–940, 1983. URL: http://www.jstor.org/stable/2273659.
9. Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.
10. A. Ciaffaglione, P. Di Gianantonio, F. Honsell, M. Lenisa, and I. Scagnetto. Reversible Computation and Principal Types in λ!-calculus. *The Bulletin of Symbolic Logic*, 25(2):931–940, 2019.
11. Alberto Ciaffaglione, Pietro Di Gianantonio, Furio Honsell, Marina Lenisa, and Ivan Scagnetto. Λ!-calculus, Intersection Types, and Involutions. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:16, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.FSCD.2019.15.
12. Alberto Ciaffaglione, Furio Honsell, Marina Lenisa, and Ivan Scagnetto. The involutions-as-principal types/application-as-unification Analogy. In *LPAR*, volume 57, pages 254–270, 2018.
13. Alberto Ciaffaglione, Furio Honsell, Marina Lenisa, and Ivan Scagnetto. Web Appendix with Erlang code. http://www.dimi.uniud.it/scagnett/pubs/automata-erlang.pdf, 2020. Accessed: 2020-01-19.
14. M. Coppo, M. Dezani-Ciancaglini, F. Honsell, and G. Longo. Extended type structures and filter lambda models. In G. Lolli, G. Longo, and A. Marcja, editors, *Logic Colloquium '82*, volume 112 of *Studies in Logic and the Foundations of Mathematics*, pages 241–262. Elsevier, 1984. doi:10.1016/S0049-237X(08)71819-6.
15. Paolo Coppola, Ugo Dal Lago, and Simona Ronchi Della Rocca. Light Logics and the Call-by-Value Lambda Calculus. *Logical Methods in Computer Science*, Volume 4, Issue 4, 2008. doi:10.2168/LMCS-4(4:5)2008.
16. Paolo Coppola and Simone Martini. Optimizing Optimal Reduction: A Type Inference Algorithm for Elementary Affine Logic. *ACM Trans. Comput. Logic*, 7(2):219–260, 2006. doi:10.1145/1131313.1131315.
17. Erlang Ecosystem Foundation. Erlang official website. http://www.erlang.org, 2020. Accessed: 2020-01-19.

18 I. Scagnetto F. Honsell, M. Lenisa. Types-as-game strategies for Hilbert style Computational Complexity. (submitted), 2020.

19 Jean-Yves Girard. Geometry of interaction III: accommodating the additives. *London Mathematical Society Lecture Note Series*, pages 329–389, 1995.

20 Jean-Yves Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.

21 Furio Honsell and Marina Lenisa. Semantical analysis of perpetual strategies in λ-calculus. *Theoretical Computer Science*, 212(1):183–209, 1999. doi:10.1016/S0304-3975(98)00140-6.

22 Furio Honsell, Marina Lenisa, Luigi Liquori, and Ivan Scagnetto. Implementing Cantor's Paradise. In Atsushi Igarashi, editor, *Programming Languages and Systems*, pages 229–250, Cham, 2016. Springer International Publishing.

23 André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 119(3), pages 447–468. [Cambridge, Eng.] Cambridge Philosophical Society., 1996.

24 Robin Milner. Is Computing an Experimental Science? *Journal of Information Technology*, 2(2):58–66, 1987. doi:10.1177/026839628700200202.

25 Dag Prawitz. *Natural deduction: A proof-theoretical study*. Courier Dover Publications, 2006.

26 Kazushige Terui. *Light logic and polynomial time computation*. PhD thesis, PhD thesis, Keio University, 2002.

A An Erlang implementation of application between specifications of partial involutions

The definition of linear application between specifications of partial involutions follows the definition in [3], namely the flow diagram in Definition 11 (see Section 3). Given f, g, $f \cdot g = f_{rr} \cup f_{rl}; g; (f_{ll}; g)^*; f_{lr}$, where $f_{ij} = \{(u, v) \mid (i(u), j(v)) \in f\}$ for $i, j \in \{l, r\}$. Hence, we must begin by implementing a function `extract`, which infers l- and r- rewriting rules from the partial involution represented by the list `L`, according to `Op1` and `Op2`:

```
extract(L,Op1,Op2) ->
  case L of
    [] -> [];
    [{e,_}|T] -> extract(T,Op1,Op2);
    [{_,e}|T] -> extract(T,Op1,Op2);
    [{{p,_,_},_}|T] -> extract(T,Op1,Op2);
    [{_,{p,_,_}}|T] -> extract(T,Op1,Op2);
    [{T1,T2}|T] -> {O1,S1}=T1,
                   {O2,S2}=T2,
                   if
                     (O1==Op1) and (O2==Op2) -> [{S1,S2} | extract(T,Op1,Op2)];
                     true -> extract(T,Op1,Op2)
                   end
  end.
```

Thus, if `F` represents a partial involution, then `extract(F,r,l)` will compute $\mathtt{F}_{rl}$.

Then, we define the *core* function `composeRuleList` which composes rule `R1`→`R2` with all the rules in `L` (exploiting the natural unification and substitution functions):

```
composeRuleList(R1,R2,L) ->
  case L of
    [] -> [];
    [{S1,S2}|T] -> {ExitStatus,MGU}=unify(R2,S1,[]),
                   if
                     (ExitStatus==ok) -> [{subListTerm(MGU,R1),subListTerm(MGU,S2)}
                                          | composeRuleList(R1,R2,T)];
                     true -> composeRuleList(R1,R2,T)
                   end
  end.
```

In order to avoid possible clashes between variable names, the `alpha` function defined below replaces all variables in Ruleset1 which also occur in Ruleset2 with freshly generated ones:

```
alpha(Ruleset1,Ruleset2) ->
  Vars1=ruleListVars(Ruleset1),
  Vars2=ruleListVars(Ruleset2),
  FreshSubst=separateVars(Vars1,Vars2),
  subListRuleset(FreshSubst,Ruleset1).
```

`alpha` is used fruitfully in the definition of `compose` which computes all possible chainings between rewriting rules of `L1` and `L2`:

```
compose(L1,L2) ->
  L1_Fresh=alpha(L1,L2),
  compose_fresh(L1_Fresh,L2).

compose_fresh(L1_Fresh,L2) ->
  case L1_Fresh of
    [] -> [];
    [H1|T1] -> {R1,R2}=H1,
               composeRuleList(R1,R2,L2)++compose_fresh(T1,L2)
  end.
```

The function `star` capitalizes on the definition of `compose`, in order to implement the computation of $\mathtt{H};(\mathtt{F};\mathtt{G})^*$:

```
star(H,F,G) ->
  S=compose(H,F),
  if
    S==[] -> H;
    true -> T=compose(S,G),
            if
              T==[] -> H;
              true -> H++star(T,F,G)
            end
  end.
```

At this point, the implementation of linear application, according to Definition 11 of Section 3, is straightforward:

```
lapp(F,G) ->
    FRR=extract(F,r,r),
    FRL=extract(F,r,l),
    FLL=extract(F,l,l),
    FLR=extract(F,l,r),
    FRL_G=compose(FRL,G),
    FRL_G_STAR=star(FRL_G,FLL,G),
    FRR++compose(FRL_G_STAR,FLR).
```