EPiC Series in Computing
Volume 57, 2018, Pages 254–270
LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

The *involutions-as-principal types/ application-as-unification* Analogy

Alberto Ciaffaglione, Furio Honsell, Marina Lenisa, and Ivan Scagnetto

Università di Udine, Italy
{alberto.ciaffaglione,furio.honsell,marina.lenisa,ivan.scagnetto}@uniud.it

Abstract

In 2005, S. Abramsky introduced various *universal* models of computation based on *Affine Combinatory Logic*, consisting of *partial involutions* over a suitable formal language of moves, in order to discuss *reversible computation* in a game-theoretic setting. We investigate Abramsky's models from the point of view of the model theory of λ-calculus, focusing on the *purely linear* and *affine* fragments of Abramsky's Combinatory Algebras.

Our approach stems from realizing a *structural analogy*, which had not been hitherto pointed out in the literature, between the partial involution interpreting a combinator and the *principal type* of that term, with respect to a *simple types discipline* for λ-*calculus*. This analogy allows for explaining as *unification* between principal types the somewhat awkward *linear application* of involutions arising from *Geometry of Interaction* (GoI).

Our approach provides immediately an answer to the open problem, raised by Abramsky, of characterising those *finitely describable partial involutions* which are denotations of combinators, in the purely affine fragment. We prove also that the (purely) linear combinatory algebra of partial involutions is a (purely) linear λ-*algebra*, albeit not a *combinatory model*, while the (purely) affine combinatory algebra is not. In order to check the complex equations involved in the definition of affine λ-algebra, we implement in Erlang the compilation of λ-terms as involutions, and their execution.

1 Introduction

In [3], S. Abramsky discusses *reversible computation* in a game-theoretic setting. In particular, he introduces various kinds of reversible *pattern-matching automata* whose behaviour can be *finitely* described as partial injective functions, actually *involutions*, over a suitable language of moves. These automata are *universal* in that they yield affine *combinatory algebras*. They are *reversible* not in the direct sense that application between combinators is itself reversible, as in *e.g.* [13]. What is "reversible" is the evaluation of the partial involutions interpreting the combinators, but this is surprisingly enough to achieve a reversible model of computation[1]. The crucial notion is that of application between automata, or between partial involutions. This is essentially the application between *history-free strategies* used in *Game Semantics*, which itself

[1]Since the partial involutions interpreting, say 0 and 1, have different behaviours on simple "tell-tale" words, we can *test reversibly* any characteristic function expressed in terms of applications of combinators, without evaluating the applications of combinators, see [3] for more details.

stems from Girard's *Execution Formula*, or Abramsky's *symmetric feedback* [1]. The former was introduced by J. Y. Girard [17, 18] in the context of "Geometry of Interaction" (GoI) to model, in a language-independent way, the fine semantics of *Linear Logic*.

Constructions similar to the Combinatory Algebra of partial involutions, introduced in [3], appear in various papers by S. Abramsky, *e.g.* [4, 5], and are special cases of a general categorical paradigm explored by E. Haghverdi [22] (Sections 5.3, 6), called "Abramsky's Programme". This Programme amounts to defining a *linear λ-algebra* starting from a *GoI Situation* in a "traced symmetric monoidal category". We shall not discuss here this abstract approach, actually the purpose of this paper is to factor it out and offer an alternative understanding of "why things work" in the context of involutions.

Reversible computation has both foundational and practical interest, spanning from low power design, reversible programming languages, optimal reductions, process algebras, quantum computation and many more [6, 21, 12, 13, 20, 28, 26]. In this paper, which is self-contained as far as technical definitions, we do not address reversibility as such, but discuss instead Abramsky's algebras from the point of view of the model theory of λ-calculus. We think that the *involutions-as-principal types/GoI application-as-unification* analogy, which we introduce, offers a new perspective on Girards's Geometry of Interaction, but also on how its reversible dynamics can arise.

We focus on the *purely linear* and *purely affine* fragments of Abramsky's affine algebras, *i.e.* without *replication*. More specifically, we introduce *purely linear* and *purely affine* combinatory logic, their λ-calculus counterparts, and their models, *i.e.* **BCI**-*combinatory algebras* and **BCK**-*combinatory algebras*. For each calculus we discuss also the corresponding notion of λ-*algebra*[2].

Our approach stems from realizing a *structural analogy*, which to our knowledge had not been hitherto pointed out in the literature, between the *Geometry of Interaction* interpretation of a λ-term in Abramsky's model of partial involutions and the *principal type* of that term, with respect to a *simple types discipline* for λ-*calculus*. We call this analogy the *involutions-as-types* analogy. In particular, we define an algorithm which, given a principal type of a λ-term, reads off the partial involution corresponding to the interpretation of that term. Thus showing that the principal type of an affine λ-term provides a characterisation of the partial involution interpreting the term in Abramsky's model. Conversely, we show how to extract a "principal type" from *any* partial involution, possibly not corresponding to any λ-term.

The *involutions-as-types* analogy is very fruitful. It allows for simply explaining as a *unification* between principal types the somewhat awkward *linear application* between involutions used in [3], deriving from the notion of application used throughout the literature on GoI and games semantics. We call this the "GoI application-as-unification of principal types" analogy, or more simply the *application-as-unification* analogy. The overall effect of linear application amounts, indeed, to *unifying* the left-hand side of the principal type of the operator with the principal type of the operand, and applying the resulting substitution to the right hand side of the operator. Hence, the notion of application between partial involutions, corresponding to λ-terms M and N, can be explained as computing the involution corresponding to the principal type of MN, given the principal types of M and N. Actually this unification mechanism works even if the types do not correspond to any λ-term.

Our analysis, therefore, unveils three conceptually independent, but ultimately equivalent, accounts of *application* in the λ-calculus: β-*reduction*, the GoI application of involutions based on symmetric feedback/Girard's *Execution Formula*, and *unification* of principal types.

These results provide an answer, for the affine part, to the open problem raised in [3] of

[2]This notion was originally introduced by D. Scott for the standard λ-calculus as the appropriate notion of categorical model for the calculus, see Section 5.2 of [8].

characterising the partial involutions which are denotations of combinators, or equivalently, arising from Abramsky's bi-orthogonal pattern matching automata. Namely, we show that these are precisely those partial involutions whose corresponding principal type is the principal type of a λ-term. In our view, this insight sheds new light on the deep nature of Game Semantics itself.

We prove, furthermore, that the purely linear combinatory algebra of partial involutions is also a purely linear λ-algebra, albeit not a purely linear combinatory model, while both the purely affine combinatory algebra and the full combinatory algebra (including replication) are not λ-algebras. We also show that the last step of Abramsky's programme, namely the one taking from a linear/affine combinatory algebra to a λ-*algebra*, is not immediate, since in general combinatory algebras cannot be quotiented non trivially to obtain λ-algebras. For the sake of readability, since we shall not discuss *replication*, throughout the paper we will refer to *purely linear* and *purely affine* combinatory algebras, and related concepts, simply as *linear* and *affine*.

In order to check all the necessary equations of λ-algebras, we implement in Erlang [14, 7] application of involutions, as well as compilation of λ-terms as combinators and their interpretation as involutions.

We conjecture that, by suitably generalizing the type discipline, the analogies introduced in this paper as well as all the results can be extended to the full affine algebra including replication.

Synopsis. In Section 2, we introduce the linear and affine versions of: combinatory logic, λ-calculus, combinatory algebra, and combinatory model, and we isolate the equations for the linear and affine combinatory algebras to be λ-algebras. In Section 3, we provide a type discipline for the linear and affine λ-calculus, and we define a corresponding notion of principal type. In Section 4, we recall Abramsky's combinatory algebra of partial involutions, and we provide a characterisation of partial involutions via principal types. Furthermore, we prove that partial involutions are a linear λ-algebra but they are not an affine λ-algebra. In Section 5, we discuss the implementation in Erlang of the application between partial involutions, and the compilation and interpretation of λ-terms. Concluding remarks appear in Section 6. The Web Appendix [31] includes the detailed Erlang programs implementing compilations and effective operations on partial involutions.

2 Linear Notions and their Affine Extensions

We introduce linear and affine versions of combinatory logic, λ-calculus, combinatory algebras, λ-algebras, and λ-models. These notions are the restrictions of the corresponding notions of combinatory logic, λ-calculus [8], and their models, to the purely linear (affine) terms. Some of these notions, although natural, are probably original, *e.g.* the equational characterization of (purely linear/affine) λ-algebras.

It is best if the reader has some familiarity with the basic notations and results in combinatory logic and λ-calculus, as presented *e.g.* in [8], and in [3], but we will be self-contained as much as possible.

Definition 1 (Linear (Affine) Combinatory Logic)**.** *The language of* linear (affine) combinatory logic $\mathbf{CL^L}$ *(*$\mathbf{CL^A}$*) is generated by variables* $x, y, \dots$ *and constants, which include the distinguished constants (combinators)* B, C, I *(and* K *in the affine case) and it is closed under application, i.e.:*

$$\frac{M \in \mathbf{CL^X} \quad N \in \mathbf{CL^X}}{M \cdot N \in \mathbf{CL^X}} \qquad \textit{for } X \in \{L, A\}$$

Combinators satisfy the following equations (we associate $\cdot$ to the left and omit it when clear from the context):

$$BMNP = M(NP) \qquad IM = M \qquad CMNP = (MP)N \qquad KMN = M$$

where M, N, P denote terms of combinatory logic.

Definition 2 (Linear (Affine) Lambda Calculus)**.** *The language $\mathbf{\Lambda^L}$ ($\mathbf{\Lambda^A}$) of the* linear (affine) λ-calculus, *i.e.* λ^L-calculus (λ^A-calculus) *is inductively defined from variables $x, y, z, \ldots \in Var$, constants $c, \ldots \in Const$, and it is closed under the following formation rules:*

$$\mathbf{\Lambda^L}: \quad \frac{M \in \mathbf{\Lambda^L} \quad N \in \mathbf{\Lambda^L}}{MN \in \mathbf{\Lambda^L}} \qquad \frac{M \in \mathbf{\Lambda^L} \quad \mathcal{E}(x, M)}{\lambda x.M \in \mathbf{\Lambda^L}}$$

$$\mathbf{\Lambda^A}: \quad \frac{M \in \mathbf{\Lambda^A} \quad N \in \mathbf{\Lambda^A}}{MN \in \mathbf{\Lambda^A}} \qquad \frac{M \in \mathbf{\Lambda^A} \quad \mathcal{O}(x, M)}{\lambda x.M \in \mathbf{\Lambda^A}}$$

where $\mathcal{E}(x, M)$ means that the variable x appears free in M exactly once*.*
where $\mathcal{O}(x, M)$ means that the variable x appears free in M at most once*.*

The rules of the λ^L-calculus (λ^A-calculus) are the restrictions of the standard β-rule and ξ-rule to linear (affine) abstractions, namely:

$$(\beta_L)\ (\lambda x.M)N = M[N/x] \qquad (\xi_L) \quad \frac{M = N \quad \mathcal{E}(x, M) \quad \mathcal{E}(x, N)}{\lambda x.M = \lambda x.N}\ .$$

$$(\beta_A)\ (\lambda x.M)N = M[N/x] \qquad (\xi_A) \quad \frac{M = N \quad \mathcal{O}(x, M) \quad \mathcal{O}(x, N)}{\lambda x.M = \lambda x.N}\ .$$

All the remaining rules are the standard rules which make $=$ a congruence.

Proposition 1. *Well-formedness in $\mathbf{\Lambda^L}$ ($\mathbf{\Lambda^A}$), i.e. linear (affine) λ-abstractions are preserved under λ-reduction. The corresponding reduction calculi are Church-Rosser.*

Proof. Routine. □

In the sequel of this section, for conciseness, we discuss only the λ^A-calculus, since the corresponding notions/results carry over straightforwardly to the linear version by simple restriction.

We start by specialising to the affine case the results in [8] on the encoding of λ-calculus into combinatory logic.

Definition 3. *We define two homomorphisms w.r.t. application:*
(i) $(\ )_{\lambda^A} : \mathbf{CL^A} \to \mathbf{\Lambda^A}$, given a term M of $\mathbf{CL^A}$, yields the term of $\mathbf{\Lambda^A}$ obtained from M by replacing each combinator with the corresponding $\mathbf{\Lambda^A}$-term as follows

$$(B)_{\lambda^A} = \lambda xyz.x(yz) \qquad (I)_{\lambda^A} = \lambda x.x \qquad (C)_{\lambda^A} = \lambda xyz.(xz)y \qquad (K)_{\lambda^A} = \lambda xy.x$$

(ii) $(\ )_{CL^A} : \mathbf{\Lambda^A} \to \mathbf{CL^A}$, given a term $M \in \Lambda^A$, replaces each λ-abstraction by a λ^*-abstraction. Terms with λ^*-abstractions amount to $\mathbf{CL^A}$-terms via the Abstraction Operation *defined below.*

Definition 4 (Affine Abstraction Operation)**.** *The following operation, defined by induction on $M \in \mathbf{CL^A}$, provides an encoding of λ^A-calculus into $\mathbf{CL^A}$:*

$$\lambda^* x.x = I \qquad \lambda^* x.c = Kc \qquad \lambda^* x.y = Ky\ , \textit{for } c \in Const,\ x \neq y$$

$$\lambda^* x.MN = \begin{cases} C(\lambda^* x.M)N & \textit{if } x \in FV(M), \\ BM(\lambda^* x.N) & \textit{if } x \in FV(N), \\ K(MN) & \textit{otherwise.} \end{cases}$$

Theorem 1 (Affine Abstraction Theorem)**.** *For all terms $M, N \in \mathbf{CL^A}$, $(\lambda^* x.M)N = M[N/x]$.*

Proof. By straightforward induction on the definition of λ^*. □

The notion of *linear (affine) combinatory algebra*, or **BCI**-algebra (**BCK**-algebra) is the restriction of the notion of *combinatory algebra* to linear (affine) combinatory logic:

Definition 5 (Linear (Affine) Combinatory Algebra, **BCI**-algebra (**BCK**-algebra))**.**
(i) A linear (affine) combinatory algebra, LCA, (ACA) $\mathcal{A} = (A, \cdot)$ *is an applicative structure, with distinguished elements (combinators) B, C, I (and K in the affine case) satisfying the following equations: for all $x, y, z \in A$,*

$$Bxyz =_{\mathcal{A}} x(yz) \qquad Ix =_{\mathcal{A}} x \qquad Cxyz =_{\mathcal{A}} (xz)y \qquad Kxy =_{\mathcal{A}} x$$

(ii) For a linear (affine) combinatory algebra $\mathcal{A}$, we define $[\![\ ]\!]_{\mathcal{A}} : \mathbf{CL^A} \to \mathcal{A}$ as the natural interpretation of closed terms of $\mathbf{CL^L}$ ($\mathbf{CL^A}$) into $\mathcal{A}$.
(iii) For a linear (affine) combinatory algebra $\mathcal{A}$, we define the set of linear (affine) combinatory terms $\mathcal{T}(\mathcal{A})$ as the extension of $\mathbf{CL^L}$ ($\mathbf{CL^A}$) with constants c_a for $a \in A$.

In what follows, when clear from the context, we will simply write $=$ in place of $=_{\mathcal{A}}$.

As we did earlier for the syntactic notions, we will discuss semantic notions only for the affine case. If not stated explicitly, the corresponding notions/theorems carry over straightforwardly, *mutatis mutandis*, to the linear case.

First we introduce *affine λ-algebras.* These were originally introduced by D. Scott for standard λ-calculus as the appropriate notion of categorical model for the calculus, see Definition 5.2.2(i) of [8].

Definition 6 (Affine λ-algebra)**.** *An ACA $\mathcal{A}$ is an* affine λ-algebra *if, for all closed $M, N \in \mathcal{T}(\mathcal{A})$,*

$$\vdash (M)_{\lambda^A} =_{\lambda^A} (N)_{\lambda^A} \implies [\![M]\!]_{\mathcal{A}} = [\![N]\!]_{\mathcal{A}} \ ,$$

where $=_{\lambda^A}$ denotes provable equivalence on λ-terms, and $[\![\ ]\!]_{\mathcal{A}}$ denotes (by abuse of notation) the natural extension to terms in $\mathcal{T}(\mathcal{A})$ of the interpretation $[\![\ ]\!]_{\mathcal{A}} : \mathbf{CL^A} \to \mathcal{A}$.

Given a **BCK**-algebra, there exists a smallest quotient giving rise to a (possibly trivial) affine λ-algebra, namely:

Definition 7. *Let $\mathcal{A} = (A, \cdot)$ be an ACA. For all $a, b \in A$, we define $a \equiv_{\mathcal{A}} b$ if and only if there exist* closed *$M, N \in \mathcal{T}(\mathcal{A})$ such that $a = [\![M]\!]_{\mathcal{A}}$, $b = [\![N]\!]_{\mathcal{A}}$, and $(M)_{\lambda^A} =_{\lambda^A} (N)_{\lambda^A}$.*

We have:

Proposition 2.
(i) Not all ACA's are affine λ-algebras.
(ii) Let $\mathcal{A} = (A, \cdot)$ be an ACA. Then the quotient $(A/\equiv_{\mathcal{A}}, \cdot_{\equiv_{\mathcal{A}}})$ is an affine λ-algebra.
(iii) Not all non-trivial ACA's can be quotiented to a non-trivial affine λ-algebra.

Proof. (i) A trivial example is the closed term model of affine combinatory logic, *i.e.* the quotient of closed terms under equality, *e.g.* CKK $\neq$ I. A more subtle example is the algebra of partial involutions $\mathcal{P}$ discussed in Section 4.
(ii) $\equiv_{\mathcal{A}}$ is a congruence w.r.t. application, since $=_\lambda$ is a congruence. Then the thesis follows from definitions.
(iii) Consider the closed term model of standard combinatory algebra induced by the equations $(SII)(SII) = I$ and $(S(BII)(BII))(S(BII)(BII)) = K$. This is clearly an ACA. S is the standard combinator from Combinatory Logic, see *e.g.* [8]. The lhs's are terms reducing to themselves, and thus can be consistently (*i.e.* without producing a trivial model) independently equated to whatever; but they are equated to each other in any affine λ-algebra.

Hence any quotient of this term model to an affine λ-algebra is trivial, because $I = K$. In the linear case the argument has to be modified by taking the second equation to be *e.g.* $(S(BII)(BII))(S(BII)(BII)) = B$. $\square$

We give now the notion of *affine combinatory model*. The corresponding one for standard λ-calculus was introduced by A. Meyer in his seminal paper [27].

Definition 8 (Affine Combinatory λ-model)**.** *An ACA $\mathcal{A}$ is an* affine combinatory λ-model *if there exists a* selector *combinator ϵ such that, for all $x, y \in A$, $\epsilon xy = xy$ and $(\forall z.\ xz = yz) \Longrightarrow \epsilon x = \epsilon y$.*

Proposition 3. *Not all affine λ-algebras are affine combinatory λ-models.*

Proof. In the case of standard combinatory logic, and hence affine combinatory logic, this is implied by the well known conjecture of Barendregt on the failure of the ω-rule, finally proved by G. Plotkin using *universal generators*, (see [8], Section 17.3-4). Theorem 6 below provides such a counterexample for the linear case, namely the algebra of partial involutions $\mathcal{P}$. $\square$

Curry was the first to discover that λ-algebras have *purely equational definitions*. We give corresponding results for linear and affine combinatory logic, which, although natural, are probably original. The significance of the following theorem is that a finite number of equations involving combinators, A^β, are enough to ensure that the congruence on $\mathbf{CL^A}$-terms is closed under the ξ_A-rule, as a rule of proof, namely if $\mathbf{CL^A} + A^\beta \vdash M = N$ then $\vdash [\![\lambda^* x.M]\!]_{\mathcal{A}} = [\![\lambda^* x.N]\!]_{\mathcal{A}}$.

Theorem 2. *An ACA $\mathcal{A}$ satisfying the following sets of equations is an affine λ-algebra:*

- $B = \lambda^* xyz.x(yz) = \lambda^* xyz.Bxyz$
  $C = \lambda^* xyz.(xz)y = \lambda^* xyz.Cxyz$
  $I = \lambda^* x.x = \lambda^* x.Ix$
  $K = \lambda^* xy.x = \lambda^* xy.Kxy$
- *equation necessary for $\lambda^* x.IP = \lambda^* x.P$ to hold: $\lambda^* y.BIy = \lambda^* yz.yz$*
- *equations necessary for $\lambda^* x.BPQR = \lambda^* x.P(QR)$ to hold:*
  - $\lambda^* uvw.C(C(BBu)v)w = \lambda^* uvw.Cu(vw)$
  - $\lambda^* uvw.C(B(Bu)v)w = \lambda^* uvw.Bu(Cvw)$
  - $\lambda^* uvw.B(Buv)w = \lambda^* uvw.Bu(Bvw)$
- *equations necessary for $\lambda^* x.CPQR = \lambda^* x.PRQ$ to hold:*
  - $\lambda^* uvw.C(C(BCu)v)w = \lambda^* uvw.C(Cuw)v$
  - $\lambda^* uvw.C(B(Cu)v)w = \lambda^* uvw.B(uw)v$
  - $\lambda^* uvw.B(Cuv)w = \lambda^* uvw.C(Buw)v$
- *equations necessary for $\lambda^* x.KPQ = \lambda^* x.P$ to hold :*
  - $\lambda^* xy.C(BKx)y = \lambda^* xyz.xz$
  - $\lambda^* xy.B(Kx)y = \lambda^* xyz.x$
- *2 more equations are necessary for K in dealing with ξ over axioms:*

– $\lambda^* xy.Bx(Ky) = \lambda^* xy.K(xy)$
– $\lambda^* xy.C(Kx)y = \lambda^* xy.K(xy)$

Proof. (Sketch) The proof follows closely the argument in [8], Section 7.3. The equations allow for proving that $\mathbf{CL^A}$ is closed under the ξ_A-rule. For each combinator we have therefore as many equations as there are possible branches in the Abstraction Operation. At the very end, suitable λ^*-abstractions need to be carried out in order to remove the parameters. □

The corresponding theorem in the linear case is obtained by deleting all the equations referring to K.

3 Linear and Affine Type Disciplines for the λ-calculus

In this section we introduce the key type-theoretic tools for understanding the fine structure of partial involutions, namely *principal simple type schemes.* Principal types were introduced by Hindley, see *e.g.* [24], but with a different purpose. We discuss the linear and affine cases separately, because they exhibit significantly different properties.

Definition 9 (Simple Types)**.** *(Type ∋)* $\mu ::= \alpha \mid \mu \to \mu$ *, where* $\alpha \in TVar$ *denotes a type variable.*

Definition 10 (Linear Type Discipline)**.** *The* linear type system *for the* λ^L*-calculus is given by the following set of rules for assigning* simple types *to terms of* $\mathbf{\Lambda}^L$*. Let* Γ, Δ *denote environments, i.e.* sets *of the form* $\Gamma = x_1 : \mu_1, \ldots, x_m : \mu_m$*, where each variable in* $dom(\Gamma) = \{x_1, \ldots, x_m\}$ *occurs exactly once:*

$$\frac{}{x : \mu \vdash_L x : \mu} \qquad \frac{\Gamma, x : \mu \vdash_L M : \nu}{\Gamma \vdash_L \lambda x.M : \mu \to \nu}$$

$$\frac{\Gamma \vdash_L M : \mu \to \nu \qquad \Delta \vdash_L N : \mu \quad (dom(\Gamma) \cap dom(\Delta)) = \emptyset}{\Gamma, \Delta \vdash_L MN : \nu} .$$

We introduce now the crucial notion of *principal type scheme*:

Definition 11 (Principal Type Scheme)**.** *Given a* λ^L*-term* M*, the judgement* $\Gamma \Vdash_L M : \sigma$ *denotes that* σ *is the* principal type scheme *of* M*:*

$$\frac{}{x : \alpha \Vdash_L x : \alpha} \qquad \frac{\Gamma, x : \mu \Vdash_L M : \nu}{\Gamma \Vdash_L \lambda x.M : \mu \to \nu}$$

$$\frac{\begin{array}{c}\Gamma \Vdash_L M : \mu \quad \Delta \Vdash_L N : \tau \quad (dom(\Gamma) \cap dom(\Delta)) = \emptyset \quad (TVar(\Gamma) \cap TVar(\Delta)) = \emptyset \\ (TVar(\mu) \cap TVar(\tau)) = \emptyset \quad U' = MGU(\mu, \alpha \to \beta) \quad U = MGU(U'(\alpha), \tau) \quad \alpha, \beta \ fresh\end{array}}{U(\Gamma, \Delta) \Vdash_L MN : U \circ U'(\beta)}$$

where MGU *gives the most general unifier, and it is defined (in a standard way) below. By abuse of notation,* U *denotes also the substitution on contexts induced by* U*.*

Definition 12 ($MGU(\sigma, \tau)$)**.** *Given two types* σ *and* τ*, the partial algorithm* MGU *yields a substitution* U *on type variables (the identity almost everywhere) such that* $U(\sigma) = U(\tau)$*:*

$$\frac{MGU(\alpha, \tau) = U \quad \alpha \in TVar \quad \tau \notin TVar}{MGU(\tau, \alpha) = U} \qquad \frac{\alpha \in TVar \quad \alpha \notin \tau}{MGU(\alpha, \tau) = id[\tau/\alpha]}$$

$$\frac{MGU(\sigma_1, \tau_1) = U_1 \quad MGU(U_1(\sigma_2), U_1(\tau_2)) = U_2}{MGU(\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) = U_2 \circ U_1}$$

where $U_2 \circ U_1$ denotes composition between the extensions of the substitutions to the whole set of terms; id denotes the identical substitution.

As is well known, the above algorithm yields a substitution which factors any other unifier, see *e.g.* [29].

The following theorem, which can be proved by induction on derivations, connects the systems defined above.

Theorem 3. *For all $M \in \Lambda^L$:*
(i) if $\Gamma \Vdash_L M : \sigma$ and $\Gamma' \Vdash_L M : \sigma'$ are derivable, then $\sigma =_\alpha \sigma'$ and $\Gamma =_\alpha \Gamma'$, i.e. *$dom(\Gamma) = dom(\Gamma')$ and $x : \mu \in \Gamma, x : \mu' \in \Gamma' \;\Rightarrow\; \mu =_\alpha \mu'$.*
(ii) if $\Gamma \Vdash_L M : \sigma$ is derivable, then each type variable occurs at most twice in $\Gamma \Vdash_L M : \sigma$.
(iii) if $\Gamma \Vdash_L M : \sigma$, then, for all substitutions U, $U(\Gamma) \vdash_L M : U(\sigma)$.
(iv) if $\Gamma \vdash_L M : \sigma$, then there exists a derivation $\Gamma' \Vdash_L M : \sigma'$ and a type substitution U, such that $U(\Gamma') = \Gamma$ and $U(\sigma') = \sigma$.

Here are some well known examples of principal types:

I	$\lambda x.x$	$\alpha \to \alpha$
B	$\lambda xyz.x(yz)$	$(\alpha \to \gamma) \to (\beta \to \alpha) \to \beta \to \gamma$
C	$\lambda xyz.xzy$	$(\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma$

Theorem 4 (Linear Subject Conversion)**.** *Let $M \in \Lambda^L$, $M =_{\beta^L} M'$, and $\Gamma \Vdash_L M : \sigma$, then $\Gamma \Vdash_L M' : \sigma$.*

Proof. First we prove subject conversion for $\vdash_L$, *i.e.*:
$\Gamma \vdash_L M : \sigma \;\wedge\; M =_{\beta^L} M' \implies \Gamma \vdash_L M' : \sigma$. This latter fact follows from:
$\Gamma, x : \mu \vdash_L M : \nu \;\wedge\; \Delta \vdash_L N : \mu \iff \Gamma, \Delta \vdash_L M[N/x] : \nu \;\wedge\; \Delta \vdash_L N : \mu$,
which can be easily proved by induction on M.
Now, let $M =_{\beta^L} M'$ and $\Gamma \Vdash_L M : \sigma$. Then, by Theorem 3(iii), $\Gamma \vdash_L M : \sigma$, and by subject conversion of $\vdash_L$, $\Gamma \vdash_L M' : \sigma$. Hence, by Theorem 3(iv), there exist U, Γ', σ' such that $\Gamma' \Vdash_L M' : \sigma'$ and $U(\Gamma') = \Gamma$, $U(\sigma') = \sigma$. But then, by Theorem 3(iii), $\Gamma' \vdash_L M' : \sigma'$, and by subject conversion of $\vdash_L$, $\Gamma' \vdash_L M : \sigma'$. Hence, by Theorem 3(iv), there exist U', Γ'', σ'' such that $\Gamma'' \Vdash_L M : \sigma''$ and $U'(\Gamma'') = \Gamma'$, $U'(\sigma'') = \sigma'$. Finally, by Theorem 3(i), $\Gamma =_\alpha \Gamma''$ and $\sigma =_\alpha \sigma''$, and hence also $\Gamma =_\alpha \Gamma'$ and $\sigma =_\alpha \sigma'$. $\square$

3.1 The Affine Case: Discussion

The extension to the λ^A-calculus of Definition 10 is apparently unproblematic. We can just add to $\vdash$ the natural rule

$$\frac{\Gamma \vdash_A M : \nu \quad x \notin dom(\Gamma) \quad TVar(\mu) \text{ fresh}}{\Gamma \vdash_A \lambda x.M : \mu \to \nu}$$

and its counterpart to $\Vdash$, namely

$$\frac{\Gamma \Vdash_A M : \nu \quad x \notin dom(\Gamma) \quad \alpha \text{ fresh}}{\Gamma \Vdash_A \lambda x.M : \alpha \to \nu}$$

However, in doing this, we get type assignment systems which satisfy the extension of Theorem 3 to the affine case, but the affine version of Theorem 4, *i.e.* subject conversion, fails. This cannot be recovered and it is the key reason for the failure of the affine combinatory algebra $\mathcal{P}$ defined in Section 4 to be an affine λ-algebra. As a counterexample consider $\lambda xyz.(\lambda w.x)(yz)$

and its β^A-reduct $\lambda xyz.x$. We have $\vdash \lambda xyz.x : \alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_1$, but we cannot derive $\Vdash_A \lambda xyz.(\lambda w.x)(yz) : \alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_1$. We can derive only $\Vdash_A \lambda xyz.(\lambda w.x)(yz) : \alpha_1 \to (\alpha_2 \to \alpha_3) \to \alpha_2 \to \alpha_1$, which is an instance of the former, because the variables, which are erased, are erased after having been applied, and the principal type keeps track of this.

4 Abramsky's Model of Reversible Computation

S. Abramsky, in [3], impressively exploits the connection between *automata* and strategies, and introduces various reversible universal models of computation. Building on earlier work, *e.g.* [5, 22], S. Abramsky defines models arising from *Geometry of Interaction* (*GoI*) *situations*, consisting of *history-free strategies*. He discusses $\mathcal{I}$, the model of *partial injections* and $\mathcal{P}$, its substructure consisting of *partial involutions*. In particular, S. Abramsky introduces notions of *reversible pattern-matching* (*bi-orthogonal*) *automata* as finitary concrete devices for implementing such strategies. In the rest of this paper, we focus on the model $\mathcal{P}$ of partial involutions and, apart from these introductory remarks, concepts and definitions are fully introduced, making this and the remaining sections essentially self-contained.

The model of partial involutions $\mathcal{P}$ yields a *full affine combinatory algebra*, *i.e.* including *replication*. This notion extends that of *affine combinatory algebra*, introduced in Definition 5, with a ! operation and extra combinators:

Definition 13 (Full Affine Combinatory Algebra, [3]). *A full affine combinatory algebra* $\mathcal{A} = (A, \cdot, !)$ *is an applicative structure* $(A, \cdot)$ *with a unary (injective) operation* $!$*, and combinators* $B, C, I, K, W, D, \delta, F$ *satisfying the following equations: for all* $x, y, z \in A$,

$$\begin{array}{llll} Bxyz = x(yz) & Ix = x & Cxyz = (xz)y & Kxy = x \\ Wx!y = x!y!y & \delta !x = !!x & D!x = x & F!x!y = !(xy). \end{array}$$

Full affine combinatory algebras are models of *full affine combinatory logic*, which extends *affine combinatory logic*, introduced in Definition 1, with !-operator and combinators W, δ, D, and F.

Partial involutions are defined over a suitable language of moves, and they can be endowed with a structure of a full affine combinatory algebra:

Definition 14 (The Model of Partial Involutions $\mathcal{P}$).
(i) T_Σ*, the language of* moves*, is defined by the signature* $\Sigma_0 = \{e\}$, $\Sigma_1 = \{l, r\}$, $\Sigma_2 = \{< , >\}$*; terms* $r(x)$ *are* output words*, while terms* $l(x)$ *are* input words *(often denoted simply by* rx *and* lx*);*
(ii) $\mathcal{P}$ *is the set of* partial involutions *over* T_Σ*, i.e. the set of all partial injective functions* $f : T_\Sigma \rightharpoonup T_\Sigma$ *such that* $f(u) = v \Leftrightarrow f(v) = u$*;*
(iii) the operation of replication *is defined by* $!f = \{(< t, u >, < t, v >) \mid t \in T_\Sigma \wedge (u, v) \in f\}$*;*
(iv) the notion of linear application *is defined by* $f \cdot g = f_{rr} \cup (f_{rl}; g; (f_{ll}; g)^*; f_{lr})$*, where* $f_{ij} = \{(u, v) | (i(u), j(v)) \in f\}$*, for* $i, j \in \{r, l\}$ *(see Fig. 1), where ";" denotes postfix composition.*

Following [3], we make a slight abuse of notation and assume that T_Σ contains pattern variables for terms. The intended meaning will be clear from the context. In the sequel, we will use the notation $u_1 \leftrightarrow v_1, \ldots, u_n \leftrightarrow v_n$, for $u_1, \ldots, u_n, v_1, \ldots, v_n \in T_\Sigma$, to denote the graph of the (finite) partial involution f defined by $\forall i.(f(u_i) = v_i \ \wedge \ f(v_i) = u_i)$. Again, following [3], we will use the above notation in place of a more automata-like presentation of the partial involution.

$$\begin{tikzcd}
\text{in} \arrow[r] & \bullet \arrow[r, "f_{rr}"] \arrow[d, "f_{rl}"'] & \bullet \arrow[r] & \text{out} \\
& \bullet \arrow[r, shift left, "g"] & \bullet \arrow[l, shift left, "f_{ll}"] \arrow[u, "f_{lr}"'] &
\end{tikzcd}$$

Figure 1: Flow of control in executing $f \cdot g$.

Proposition 4 ([3], Th.5.1)**.** *$\mathcal{P}$ can be endowed with the structure of a full affine combinatory algebra, $(\mathcal{P}, \cdot, !)$, where combinators are defined by the following partial involutions:*

$$\begin{array}{lll|lll}
B & : & r^3x \leftrightarrow lrx \ , \ l^2x \leftrightarrow rlrx \ , \ rl^2x \leftrightarrow r^2lx & I & : & lx \leftrightarrow rx \\
C & : & l^2x \leftrightarrow r^2lx \ , \ lrlx \leftrightarrow rlx \ , \ lr^2x \leftrightarrow r^3x & K & : & lx \leftrightarrow r^2x \\
F & : & l\langle x, ry\rangle \leftrightarrow r^2\langle x, y\rangle \ , \ l\langle x, ly\rangle \leftrightarrow rl\langle x, y\rangle & \delta & : & l\langle\langle x, y\rangle, z\rangle \leftrightarrow r\langle x, \langle y, z\rangle\rangle \\
W & : & r^2x \leftrightarrow lr^2x \ , \ l^2\langle x, y\rangle \leftrightarrow rl\langle lx, y\rangle \ , \ lrl\langle x, y\rangle \leftrightarrow rl\langle rx, y\rangle & D & : & l\langle e, x\rangle \leftrightarrow rx \ .
\end{array}$$

In Section 4 we focus on the purely linear and affine parts of the above combinatory algebra, *i.e.* $(\mathcal{P}, \cdot)$ together with combinators B, C, I (and K).

4.1 From Principal Types to Involutions and back

As pointed out in the Introduction, our approach builds on an *analogy*, which could be viewed also as a *duality* in the style of [2], between *principal type schemes* and the interpretation as involutions, in $\mathcal{P}$, of linear and affine combinators. The following algorithm is a *transform* which, given a principal type scheme, *i.e.* a *global* representation of an object, yields for each type-variable (a component of) an involution:

Definition 15. *Given a closed term M of λ^A-calculus such that $\Vdash_A M : \mu$, for each type variable $\alpha \in \mu$, the judgements $\mathcal{T}(\alpha, \mu)$ yield a pair in the graph of a partial involution, if α occurs twice in μ, or an element of T_Σ, if α occurs once in μ:*

$$\mathcal{T}(\alpha, \alpha) \ = \ \alpha \qquad \mathcal{T}(\alpha, \mu(\alpha) \to \nu(\alpha)) \ = \ l(\mathcal{T}(\alpha, \mu(\alpha))) \ \leftrightarrow \ r(\mathcal{T}(\alpha, \nu(\alpha)))$$

$$\mathcal{T}(\alpha, \mu(\alpha) \to \nu) \ = \ l[\mathcal{T}(\alpha, \mu(\alpha))] \qquad \mathcal{T}(\alpha, \mu \to \nu(\alpha)) \ = \ r[\mathcal{T}(\alpha, \nu(\alpha))]$$

where $r[x] = \begin{cases} rx_1 \leftrightarrow rx_2 & \textit{if } x = x_1 \leftrightarrow x_2 \ \wedge \ x_1, x_2 \in T_\Sigma \\ rx & \textit{otherwise} \end{cases}$

and similarly for $l[x]$.

We define the partial involution $f_\mu = \{\mathcal{T}(\alpha, \mu) \mid \alpha$ appears twice in $\mu\}$.

Vice versa, any partial involution interpreting a closed $\mathbf{CL^A}$-term M induces the corresponding *principal type*, inverting the clauses in Definition 15. Notice that so doing we can derive, actually, a "principal type scheme" from any partial involution, not just those which are indeed interpretations of λ-terms. This remark will be crucial in addressing Abramsky's open question in Section 4.1.1.

Definition 16. *We denote by $[\![\ ]\!]_\mathcal{P}$ the interpretation of closed $\mathbf{CL^A}$-terms in $(\mathcal{P}, \cdot)$.*

Theorem 5. *Given a closed term of $\mathbf{CL^A}$, say M, the partial involution interpreting M, namely $[\![M]\!]_\mathcal{P}$, can be read off the principal type scheme of $(M)_{\lambda^A}$, i.e. $\Vdash (M)_{\lambda^A} : \mu$ if and only if $[\![M]\!]_\mathcal{P} = f_\mu$.*

Proof. (Sketch) By induction on the structure of $\mathbf{CL}^{\mathbf{A}}$-terms. One can easily check that the thesis holds for combinators B, C, I, K. The inductive step amounts to showing that the notion of application in $\mathcal{P}$ corresponds to computing the principal type scheme of the application, *i.e.*, for MN closed $\mathbf{CL}^{\mathbf{A}}$-term, if $\Vdash M : \mu$, $\Vdash N : \tau$, $TVar(\mu) \cap TVar(\tau) = \emptyset$, $U' = MGU(\mu, \alpha \to \beta)$, $U = MGU(U'(\alpha), \tau)$, α, β fresh, then $f_{U \circ U'(\beta)} = f_\mu \cdot f_\tau$. This latter fact can be proved by chasing, along the control flow diagram in the definition of application, the behaviour of the MGU. □

We are finally in the position of justifying the claims, in the introduction, that our analysis unveils three conceptually independent, but ultimately equivalent, accounts of *application* in the λ-calculus: *β-reduction*, the GoI application of involutions based on symmetric feedback/-Girard's *Execution Formula*, and *unification* of principal types. In effect, computing the partial involutions $[\![M]\!]_{\mathcal{P}} \cdot [\![N]\!]_{\mathcal{P}}$, according to Definition 14, amounts by Theorem 5 to *unifying* the left-hand side of the principal type of M with the principal type of N, thus computing the principal type of MN. Using Definition 15 we can finally read off from this type scheme the partial involution $[\![MN]\!]_{\mathcal{P}}$.

Notice that the proof of Theorem 5 above shows also that linear application between partial involutions *per se*, even when these are not interpretations of combinators, can be explained as unification (resolution) between their corresponding "principal type scheme" in the general sense!

The following theorem concludes our model theoretic analysis:

Theorem 6.
(i) The linear combinatory algebra of partial involutions $(\mathcal{P}, \cdot)$ is a linear λ-algebra, albeit not a linear combinatory λ-model.
(ii) The affine combinatory algebra of partial involutions $(\mathcal{P}, \cdot)$ is not an affine λ-algebra.

Proof.
(i) All the equations in Theorem 2 have been verified. In order to avoid errors we used the Erlang program described in Section 5, where we will discuss also the epistemic impact of this approach.

The proof that there does not exist a term which behaves as a selector combinator, namely that ϵ does not exist, follows from Lemma 1 below. The combination of items (i) and (ii) of Lemma 1 below contradicts the unique selection property of ϵ, namely we exhibit two objects, *i.e.* $\emptyset$ and $\{l\alpha \leftrightarrow l\alpha\}$, which have the same empty applicative behaviour, but for any E which satisfies $\forall x, y.\ E \cdot x \cdot y = x \cdot y$, we have $E \cdot \emptyset \neq E \cdot \{l\alpha \leftrightarrow l\alpha\}$. Consider first the terms $X = \{r\alpha \leftrightarrow l\alpha, l\beta \leftrightarrow l\beta\}$ and $Y = \{\alpha \leftrightarrow \beta\}$, with $\alpha \neq \beta$. Clearly we have $X \cdot Y = \{\alpha \leftrightarrow \alpha\}$. But $E \cdot X = E \cdot \{r\alpha \leftrightarrow l\alpha\} \cup E \cdot \{l\beta \leftrightarrow l\beta\}$, by Lemma 1 (ii). Now $E \cdot \{r\alpha \leftrightarrow l\alpha\} \cdot Y = \{r\alpha \leftrightarrow l\alpha\} \cdot Y = \emptyset$. Hence $E \cdot \{l\beta \leftrightarrow l\beta\} \neq \emptyset$, since $E \cdot X \cdot Y = X \cdot Y \neq \emptyset$.

(ii) We have that $(BBK)_{\lambda^A} = (BKK)_{\lambda^A}$, but $[\![BBK]\!]_{\mathcal{P}} \neq [\![BKK]\!]_{\mathcal{P}}$. Namely, $[\![BBK]\!]_{\mathcal{P}} = [\![\lambda^* xyz.Kx(yz)]\!]_{\mathcal{P}} = \{lx \leftrightarrow r^3x, rl^2x \leftrightarrow r^2lx\}$ while $[\![BKK]\!]_{\mathcal{P}} = [\![\lambda^* xyz.x]\!]_{\mathcal{P}} = \{lx \leftrightarrow r^3x\}$. □

Lemma 1. *Assume that there exists $E \in \mathcal{P}$ such that $\forall x, y.\ E \cdot x \cdot y = x \cdot y$, then*
(i) $E_{rr} = \emptyset$, and hence $E \cdot \emptyset = \emptyset$;
(ii) $E_{ll} = \emptyset$, and hence E has an "additive" applicative behaviour, namely $E \cdot (A \cup B) = (E \cdot A) \cup (E \cdot B)$.

Proof.
(i) We proceed in stages.

- $r\alpha \leftrightarrow r\beta \notin E_{rr}$, for any α, β, otherwise $\alpha \leftrightarrow \beta \in E \cdot \emptyset \cdot X = \emptyset \cdot X = \emptyset$, contradiction.

- $r\alpha \leftrightarrow l\beta \notin E_{rr}$, for any α, β, otherwise let $A = \{r\alpha \leftrightarrow l\delta, r\delta \leftrightarrow l\alpha\}$ with δ and β not unifiable. Then $A \cdot \{\beta \leftrightarrow \beta\} = \emptyset$, but $E \cdot A \cdot \{\beta \leftrightarrow \beta\} = \{\alpha \leftrightarrow \alpha\}$. Contradiction.

- $l\alpha \leftrightarrow l\beta \notin E_{rr}$, for any $\alpha \neq \beta$, otherwise let $A = \{r\alpha \leftrightarrow l\alpha\}$, then $A \cdot \{\alpha \leftrightarrow \alpha\} = \{\alpha \leftrightarrow \alpha\}$. Now, since E is a selector, $E \cdot A \cdot \{\alpha \leftrightarrow \alpha\} = \{\alpha \leftrightarrow \alpha\}$, there must occur in $E \cdot A$ a term $r\alpha \leftrightarrow l\rho$ such that ρ can unify with the l.h.s. of the only pair in the argument, *i.e.* $\alpha = \rho$, but then $E \cdot A$ would no longer be a non-ambiguous reversible relation, because by assumption $l\alpha \leftrightarrow l\beta \in E \cdot A$. Contradiction.

- Similar arguments can be used to rule out the remaining cases, *i.e.* $l\alpha \leftrightarrow l\alpha \in E_{rr}$ and the case where one of the components is garbage, *i.e.* it has no functional effect.

(ii) We proceed in stages.

- From the very definition of application in $\mathcal{P}$, we have immediately that, if $A_{ll} = \emptyset$, then A has an "additive" behaviour under application, because it calls the argument only once.

- $E_{ll} = \emptyset$ because for all $l(\alpha)$ either $ll\alpha \leftrightarrow rr\alpha \in E$ or $lr\alpha \leftrightarrow rr\alpha \in E$ and $rl\alpha \leftrightarrow ll\alpha \in E$, and hence there are no $l\alpha \longrightarrow l\beta \in E_{ll}$, since E is an involution and the rewrite rules are deterministic. To see the above, first notice that $r\alpha \leftrightarrow l\alpha \in E \cdot \{r\alpha \leftrightarrow l\alpha\}$ for all α, otherwise, checking the control-flow diagram, one can easily see that we could not have that $E \cdot \{r\alpha \leftrightarrow l\alpha\} \cdot \{\alpha \leftrightarrow \alpha\} = \{\alpha \leftrightarrow \alpha\}$. But now, again with just a little case analysis on the control-flow diagram, one can see that there are only two alternatives in E_{rl} and E_{lr}, which give rise to the cases above.
  The only case left is $le \leftrightarrow le \in E_{ll}$. But then we would have that
  $\alpha \leftrightarrow \alpha \in (E \cdot \{r\alpha \leftrightarrow l\alpha\} \cdot \{\alpha \leftrightarrow e\})$, but $\{r\alpha \leftrightarrow l\alpha\} \cdot \{\alpha \leftrightarrow e\} = \emptyset$, contradiction.
  Hence we have that $\{r\alpha \leftrightarrow l\alpha, r\delta \leftrightarrow l\delta\} \cdot \{\alpha \leftrightarrow \beta, \gamma \leftrightarrow \delta\} = \emptyset$,
  but $E \cdot \{r\alpha \leftrightarrow l\alpha, r\delta \leftrightarrow l\delta\} \cdot \{\alpha \leftrightarrow \beta, \gamma \leftrightarrow \delta\} \supseteq \{\alpha \leftrightarrow \beta\}$, contradiction.

□

4.1.1 Abramsky's Open Problem

In [3], S. Abramsky raised the question: "Characterize those partial involutions which arise from *bi-orthogonal pattern-matching automata*, or alternatively, those which arise as denotations of combinators".

Theorem 5 suggests an answer to the above question for the affine fragment, *i.e.* without the operator $< \, , \, >$ in the language of partial involutions. The first issue to address is how to present partial involutions. To this end we consider the language T_{Σ^X}, which is the initial term algebra over the signature Σ^X for $\Sigma^X_0 \equiv X$, where X is a set of variables, and $\Sigma^X_1 = \{l, r\}$. Sets of pairs in T_{Σ^X} denote schemata of pairs over $T_{\Sigma \setminus \Sigma_2}$, *i.e.* partial involutions in $\mathcal{P}$. As pointed out in the previous section, given a partial involution defined by a finite collection of pairs in T_{Σ^X}, say $H \equiv \{u_i \leftrightarrow v_i\}_{i \in I}$ for $u_i, v_i \in \Sigma^X$, we can synthesize a type τ_H from H by gradually specifying its tree-like format. Finally we check whether τ_H is the principal type of a linear term. We proceed as follows. Each pair in H will denote two leaves in the type τ_H, tagged with the same type variable. The sequence of l's and r's, appearing in the prefix before a variable in a pair u_i, v_i, denotes the path along the tree of the type τ_H, which is under formation, where the type variable will occur. A fresh type variable is used for each different pair. At the end of this process we might not yet have obtained a complete type. Some leaves in the tree might not be tagged yet, these arise in correspondence of vacuous abstractions. We tag each such node with a new fresh type variable. H is finite otherwise we end up with an infinite type, which cannot be the principal type of a *finite* combinator. The type τ_H thus obtained has the property that each type variable occurs at most twice in it. Potentially it is a principal type.

The type τ_H is indeed a principal type of a closable λ-term (*i.e.* a term which reduces to a closed term) if and only if it is an *implication tautology* in minimal logic. This can be effectively checked in polynomial-space [30].

To complete the argument we need to show that if the type τ_H is inhabited it is indeed inhabited by a term for which it is the principal type.

Proposition 5. *If μ is a type where each type variable occurs at most twice and it is inhabited by the closed term M, then there exists N such that $\Gamma \Vdash_A N : \mu$ and $N =_{\beta_A \eta} M$.*

Proof. (Sketch) If M is a closed term, then there exists ν such that $\Gamma \Vdash_L M : \nu$. The variables in Γ will be eventually erased. If M inhabits μ, then by Theorem 3 there exists a substitution U such that $U(\nu) = \mu$. For each variable which is substituted by U, say α, two cases can arise, either α occurs twice or once. In the first case we will replace the term variable, say x, in M in whose type α occurs, which must exist, by a suitable long-η-expansion of x. This long η-expansion can always be carried out because the typed η-expansion rule is a derivable rule in the typing system.
In case the type variable α occurs only once in M, there is a subterm of M which is embedded in a vacuous abstraction. The term N is obtained by nesting that subterm with a new vacuous λ-abstraction applied to a long-η-expansion of the variable vacuously abstracted in M.
Here are two examples. From $H_1 = \{lllx \leftrightarrow rllx, llrx \leftrightarrow lrx, rrx \leftrightarrow rlx\}$ we can synthesize the type $((\alpha \to \beta) \to \gamma) \to (\alpha \to \beta) \to \gamma$. The identity, $\lambda x.x$, inhabits this type, but the type is not the principal type of the identity. It is instead the principal type of an η-expansion of the identity, namely $\lambda xy.x(\lambda z.yz)$.
From $H_2 = \{lllx \leftrightarrow lrrx, llrx \leftrightarrow lrlx, lrrx \leftrightarrow rrrx\}$ we can synthesize the type $((\alpha \to \beta) \to (\beta \to \alpha)) \to \gamma \to \gamma$. The term $\lambda yx.x$ inhabits this type which is the principal type of its β-expansion $\lambda yx.(\lambda w.x)(\lambda zw.yzw)$.

□

So we can finally state the result which provides an answer to Abramsky's open problem [3], in the purely linear case:

Theorem 7. *In the* affine *case, the denotations of combinators in $\mathcal{P}$ are precisely the partial involutions from which we can synthesize, according to the procedure outlined above, a principal type scheme which is a tautology in minimal logic.*

Proof. Use Proposition 5 above in one direction, and Definition 15 and Theorem 5 in the opposite direction. □

The above is a satisfactory characterisation because it is conceptually independent both from λ-terms and from involutions.

5 Machine Verification of Semantical Equalities

Since the manual verification of complicated equations like those appearing in Theorem 2 is a lengthy and error-prone task, we developed an Erlang program to automate the proof of equivalence of expressions involving involutions (see [31] for the details). The main components of this program are the implementations of the *Affine Abstraction Operator* (see Definition 4) and of the linear application operator $f \cdot g$ (see Definition 14) introduced by S. Abramsky in [3].

The epistemic significance of using a machine to verify a set of equations, according to a set of computation rules, is known to reduce to the problem of the correctness of the implementation of the language and the correctness of the specification of the program. Using a machine to this end is therefore different, and in principle less reliable, than checking a proof with the assistance of a Logical Framework, such as *e.g.* [23, 10]. Nevertheless when the amount of details, as in this case, escapes human alertness, even formal proof assistants resort to computation and rewriting tools to verify equations. Therefore we briefly discuss the program we developed to assist us in verifying the equations between involutions, both for the sake of completeness, but also because morally it is part of the proof of Theorem 6.

There are several reasons behind the choice of Erlang: expressive pattern matching mechanisms, avoidance of side effects thanks to one-time assignment variables, powerful libraries for smooth handling of lists, tuples etc. However, other functional languages can be an effective and viable choice as well.

In our program we allow the user to input involution rules, using `<->` to denote rewritings, much as one would do with pencil and paper, using the language $T_{\Sigma X}$ of Section 4.1.1. Capital letters or strings with initial capital letter denote variables, according to Erlang conventions. Parentheses need not be specified if they can be automatically inferred. The notation $\langle x, y\rangle$ stands for $p(x, y)$. We use `leex` and `yecc` (*i.e.* the Erlang versions of Lex and Yacc) to build lexical analyzers and parser for the language of partial involutions, combinators, and λ-terms in order to yield appropriate internal representations.

For instance, combinator **B** is defined by inputting the string representing its three rules (*i.e.* `"rrrX<->lrX, llX<->rlrX, rllX<->rrlX"`). From this we obtain the following internal representation (each rule involving `<->` yields two internal rules corresponding to the two possible directions of rewriting):

```
[{{r,{r,{r,{var,"X"}}}},{l,{r,{var,"X"}}}}, {{l,{r,{var,"X"}}},{r,{r,{r,{var,"X"}}}}},
 {{l,{l,{var,"X"}}},{r,{l,{r,{var,"X"}}}}}, {{r,{l,{r,{var,"X"}}}},{l,{l,{var,"X"}}}},
 {{r,{l,{l,{var,"X"}}}},{r,{r,{l,{var,"X"}}}}},
                                            {{r,{r,{l,{var,"X"}}}},{r,{l,{l,{var,"X"}}}}}]
```

We can compute the composition $f; g$ (`compose(F,G)`) of two involutions f and g as the set of rules $\{(R_1, R_2) \mid R_1 = s(F_1),\ R_2 = s(G_2),\ (F_1, F_2) \in f, (G_1, G_2) \in g, \text{and } s = m.g.u.(F_2, G_1)\}$, where m.g.u. stands for *most general unifier* which can be implemented following Robinson's unification algorithm [29]. There is only a subtle issue to take into account, namely unification may not work correctly if the sets of variables of f and g are not disjoint. Hence, we preventively rename variables of f, if this is not the case, in the computation of $f; g$. Once the implementation of $f; g$ is completed, it is trivial to define $f \cdot g$ by unfolding its definition in terms of the composition operator and calculating f_{rr} (`extract(F,r,r)`), f_{rl} (`extract(F,r,l)`), f_{ll} (`extract(F,l,l)`), f_{lr} (`extract(F,l,r)`), exploiting in `extract` the pattern matching features of Erlang.

Let us see, as an example, how the verification of equation $\lambda^* xyz.C(C(BBx)\ y)z = \lambda^* xyz.Cx(yz)$ from Theorem 2 is carried out. First, we parse the two members of the above mentioned equation, yielding an internal representation of the two λ-abstractions. Then, we apply the implementation of the *Affine Abstraction Operator* to yield the following expressions:

1. $((C((BC)((B(BB))((B(BC))((C((BB)((BC)((B(BB))I))))I))))))I)$

2. $((C((BB)((BB)((BC)I))))((C((BB)I))I))$

corresponding, respectively, to $\lambda^* xyz.C(C(BBx)y)z$ and to $\lambda^* xyz.Cx(yz)$.

Finally, since each combinator can be internally represented as a list of rewriting clauses (according to [3]), it is only a matter of applying the application operation between partial involutions (*i.e.* $f \cdot g$) in order to check that the combinator expressions are indeed equal, *i.e.*, they generate the same involution.

All the equations appearing in this paper have been machine checked using this program.

For further details and for the implementation of the replication operator (!) we refer the interested reader to the Web Appendix [31].

6 Final Remarks and Directions for Future Work

In this paper, we have analysed from the point of view of the model theory of λ-calculus the purely linear and affine fragments of the combinatory algebra of partial involutions, $\mathcal{P}$, introduced in [3]. The interest of this algebra lies in the fact that it yields a reversible model of computation. Moreover, we have shown that the last step of "Abramsky's Programme", taking from a full linear combinatory algebra to a λ-algebra is not immediate. Actually, we have proved that non trivial quotients do not exist in general. In the setting of partial involutions, only in the purely linear case we have a λ-algebra. Already in the affine case the model of partial involutions cannot be immediately turned into a λ-algebra. In order to check all the necessary complex details, we have implemented the verification of equalities between partial involutions in the language Erlang.

The key insight which has allowed us to analyze the fine structure of the partial involutions interpreting combinators has been what we termed the *involutions-as-principal types/application-as-unification* analogy, which highlights a form of structural duality between involutions and principal types, w.r.t. a suitable simple type discipline. This alternate and novel characterization of partial involutions suggested immediately an answer to an open problem raised by S. Abramsky in [3], in the purely linear and affine cases. Namely, the partial involutions which are interpretations of combinators are those whose corresponding principal type is inhabited.

We conjecture that all the above results can be generalized also to Abramsky's full combinatory algebra, including replication. In the abstract [11], we outlined a notion of affine λ-calculus with replication, the $\lambda^!$-calculus, which includes a unary ! operator on terms, and has two kinds of λ-abstractions, *affine abstraction* and *!-abstraction*. The former can abstract only affine terms, while the latter is a *pattern abstraction* which can abstract all λ-terms but can fire only if the operand has an outermost ! operator. The Abstraction Operation in Definition 4 and the Abstraction Theorem 1, as well as a counterpart to Theorem 2 can then be extended to the full affine case. We conjecture that the quotient of $\mathcal{P}$, obtained by an applicative equivalence defined by induction on a suitable measure of the complexity of the words in T_Σ, yields a $\lambda^!$-algebra.

In order to obtain the analogous of Theorems 3 and 5, we need to extend the type system to the $\lambda^!$-calculus, so as to be able to express some kind of "principal type". We conjecture that the simple type discipline needs to be extended to a suitable intersection type system ([9]), in the line of [15, 16], and equipped with a $!_u$ type operator, indexed by words $u \in T_\Sigma$. We also conjecture that a connection between partial involutions in the algebra $\mathcal{P}$ and principal type schemes in this extended intersection type discipline, generalizing Proposition 5, can be given. This latter result will provide an answer to the full version of Abramsky's open problem along the lines of the procedure described in the present paper for the affine case.

An interesting problem to address is to characterize the fine theory of $\mathcal{P}$. This can be done by proving a suitable Approximation Theorem, again relying on a complexity measure on involutions, induced by a complexity measure on words in T_Σ. A further line of investigation is that of generalizing to the combinatory algebra of partial injections or other GoI situations, in the sense of [4, 22], the duality and the results detected in the case of partial involutions.

We conclude this broad spectrum of future work by pointing out once again what we think is the novelty of our paper, namely the *involutions-as-principal types/GoI application-as-unification*, which, to our knowledge, had not been hitherto noticed. Clearly, there are interesting connections between the `l` and `r` language constructors of involutions and other operators such as Levy labels [25], or the p and q constructors in dynamic algebras [19, 12]. Hence our analogy can have a bearing, in providing alternate accounts based on principal types, also in these contexts, as well as in the field of optimal and reversible implementations of GoI machines (see, e.g., [20, 26, 12, 28]). But even more interesting, in our view, is to assess whether our reading based on unification of types, outlined in the case of involutions, can shed more light on the fine structure of application in GoI or game semantics, which are apparently type-less.

References

[1] S. Abramsky and R. Jagadeesan. New foundations for the geometry of interaction. *Information and Computation*, 111:53–119, 1994.

[2] Samson Abramsky. Domain theory in logical form. *Ann. Pure Appl. Logic*, 51(1-2):1–77, 1991.

[3] Samson Abramsky. A structural approach to reversible computation. *Theoretical Computer Science*, 347(3):441 – 464, 2005.

[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.

[6] T. Altenkirch and J. Grattage. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 249–258, June 2005.

[7] J. Armstrong. *Making reliable distributed systems in the presence of software errors.* PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003.

[8] Hendrik Pieter Barendregt et al. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic.* North-Holland Amsterdam, 1984.

[9] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[10] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA, May 1997. Projet COQ.

[11] A. Ciaffaglione, P. Di Gianantonio, F. Honsell, M. Lenisa, and I. Scagnetto. Reversible Computation and Principal Types in $\lambda^!$-calculus. In H. Dugald Macpherson, editor, *The Bulletin of Symbolic Logic, Logic Colloquium 2018*, to appear.

[12] Vincent Danos and Laurent Regnier. Reversible, irreversible and optimal λ-machines. *Theoretical Computer Science*, 227(1):79 – 97, 1999.

[13] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Reversible combinatory logic. *Mathematical Structures in Computer Science*, 16(4):621–637, 2006.

[14] Erlang official website. http://www.erlang.org. Last access: 19/01/2018.

[15] Pietro Di Gianantonio, Furio Honsell, and Marina Lenisa. A type assignment system for game semantics. *Theoretical Computer Science*, 398(1):150 – 169, 2008. Calculi, Types and Applications: Essays in honour of M. Coppo, M. Dezani-Ciancaglini and S. Ronchi Della Rocca.

[16] Pietro Di Gianantonio and Marina Lenisa. Innocent Game Semantics via Intersection Type Assignment Systems. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013)*, volume 23 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 231–247, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[17] Jean-Yves Girard. Geometry of Interaction 1: Interpretation of System F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloquium '88*, volume 127 of *Studies in Logic and the Foundations of Mathematics*, pages 221 – 260. Elsevier, 1989.

[18] Jean-Yves Girard. Geometry of interaction 2: Deadlock-free algorithms. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, pages 76–93, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[19] Jean-Yves Girard. *The Blind Spot: lectures on logic*. European Mathematical Society, 2011.

[20] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 15–26, New York, NY, USA, 1992. ACM.

[21] Alexander S. Green and Thorsten Altenkirch. From reversible to irreversible computations. *Electronic Notes in Theoretical Computer Science*, 210:65 – 74, 2008. Proceedings of the 4th International Workshop on Quantum Programming Languages (QPL 2006).

[22] Esfandiar Haghverdi. *A Categorical Approach to Linear Logic, Geometry of Proofs and full completeness.* PhD thesis, University of Ottawa, 2000.

[23] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993.

[24] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, pages 29–60, 1969.

[25] Jean-Jacques Lévy. An algebraic interpretation of the $\lambda\beta$k-calculus; and an application of a labelled λ-calculus. *Theoretical Computer Science*, 2(1):97 – 114, 1976.

[26] Ian Mackie. The geometry of interaction machine. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 198–208, New York, NY, USA, 1995. ACM.

[27] Albert R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52(1):87 – 122, 1982.

[28] Koko Muroya and Dan R. Ghica. Efficient implementation of evaluation strategies via token-guided graph rewriting. In Horatiu Cirstea and David Sabel, editors, Proceedings Fourth International Workshop on *Rewriting Techniques for Program Transformations and Evaluation,* Oxford, UK, 8th September 2017, volume 265 of *Electronic Proceedings in Theoretical Computer Science*, pages 52–66. Open Publishing Association, 2018.

[29] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[30] Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9:67–72, 1979.

[31] Web appendix with erlang code. http://www.dimi.uniud.it/scagnett/pubs/automata-erlang.pdf.