Mathematical Structures in Computer Science (2023), **33**, pp. 647–659
doi:10.1017/S0960129523000191

CAMBRIDGE UNIVERSITY PRESS

PAPER

A (machine-oriented) logic based on pattern matching

Tim Lethen

Department of Philosophy, University of Helsinki, Helsinki, Finland
Email: tim.lethen@gmx.de

(Received 16 October 2022; revised 1 May 2023; accepted 1 June 2023; first published online 5 July 2023)

Abstract

Robinson's unification algorithm can be identified as the underlying machinery of both C. Meredith's rule **D** (*condensed detachment*) in propositional logic and of the construction of *principal types* in lambda calculus and combinatory logic. In combinatory logic, it also plays a crucial role in the construction of Meyer, Bunder & Powers' *Fool's model*. This paper now considers pattern matching, the unidirectional variant of unification, as a basis for logical inference, typing, and a very simple and natural model for untyped combinatory logic. An analysis of the new typing scheme will enable us to characterize a large class of terms of combinatory logic which do not change their principal type when being weakly reduced. We also consider the question whether the major or the minor premisse should be used as the fixed pattern.

Keywords: Pattern matching; Unification; Condensed detachment; Principal types; Combinatory logic; Fool's model

1. Introduction

In propositional logic, *rule* **D**, or *condensed detachment* as it is often called, is a rule of inference invented by Carew Arthur Meredith in 1957 (cf. Lemmon et al. 1957, Section 9) as a means to study propositional calculi. As we shall see in Section 2, it combines a "minimal" amount of substitution with the rule of detachment or *modus ponens*. While condensed detachment, which has been called "Meredith's most widely used innovation in logic" (Meredith 1977), is often regarded as an abbreviative device in Hilbert-style proofs, it has also widely been used in the field of automated theorem proving,[1] and it is known that David Meredith, as early as 1957, "programmed UNIVAC I to find the *D*-derivatives of any given set of formulae, but with only one thousand 12-character words of memory ... didn't get any results that could not have been obtained by hand more quickly." (Kalman 1983).

A few years later, in 1965, J. A. Robinson's pioneering and influential paper introducing the *unification* algorithm appeared (Robinson 1965), and it was soon realized that it was this very algorithm which lay at the heart of Meredith's rule **D**.[2] Unification was a central part of Robinson's resolution principle, which can be regarded as a "machine-oriented" rule of inference for first-order logic. Nevertheless, as Robinson (1979, p. 292) states, the idea of unification "turns out to have been sitting there all these years, unnoticed, in Herbrand's doctoral thesis" (cf. Herbrand 1930, Chapter 5, Section 2.4). Further details about the history of condensed detachment and its connection to Robinson's unification algorithm can be found in Meredith (1977), Kalman (1983), and Hindley (1997, p. 103).

Another strand of research closely connected to unification is the theory of *simple types*. Here, the *principal type* of a term in lambda calculus or combinatory logic can be regarded as the term's

© The Author(s), 2023. Published by Cambridge University Press.

https://doi.org/10.1017/S0960129523000191 Published online by Cambridge University Press

Check for updates

most general type, and it was soon realized that the algorithm designed to compute the principal type of a term is not only based on Robinson's unification algorithm but also mirrors Meredith's rule **D** of the propositional calculus in a perfect manner. An overview over the history of this apparently independent field of development can be found in Hindley (1997, p. 33).

Finally, a type-lifted variant of rule **D** – and thus the unification algorithm – also lies at the heart of a model of untyped combinatory logic published in Meyer et al. (1991). It is called the "Fool's Model."

In this paper, we now shift the emphasis to an algorithm called *pattern matching*, which can be regarded as a simplified, "unidirectional variant of unification" (Knight 1989), and which is very popular in a wide range of applications. As we shall see, this variant gives rise to a new rule of inference in propositional logic, to a simplified typing scheme in lambda calculus and combinatory logic, and, finally, to a very natural model of combinatory logic.

We proceed as follows. In Section 2, we summarize the basics of condensed detachment, principal types, and the unification algorithm. In Section 3, we introduce a version of pattern matching upon which we will base the new rule of inference and the new typing scheme. As we shall see in Section 4, this typing scheme will lead to the identification of a large class of terms of combinatory logic which do not change their principal type when weakly reduced. In Section 5, we introduce a simple new model for untyped combinatory logic, which we call the *pattern model*. Finally, Section 6 will consider the question why the introduced logical inference and the typing based on pattern matching take the *minor* premisse (instead of the *major* premisse) as its fixed pattern.

2. Condensed Detachment, Principal Types, and Unification

In this section, we summarize some basic definitions and facts about condensed detachment, principal types, and the unification algorithm, which will help to introduce the new rule of inference and the new typing scheme in the section to follow.

2.1 Propositional logic and condensed detachment

In order to facilitate the comparison between condensed detachment and principal types, the considered propositional formulas will be purely implicational. To be more precise, propositional formulas are defined by the simple grammar rule:

$$\varphi ::= p \mid (\varphi \rightarrow \varphi) \tag{1}$$

where p ranges over a countable infinite set of propositional variables.

We call the formula α' an *alphabetic variant* of the formula α if, in order to obtain α', all or some variables in α have been substituted by variables distinct from each other and distinct from any variables in α which have not been replaced.[3] A substitution σ is called a *unifier* of the formulas α and β if $\sigma(\alpha) = \sigma(\beta)$. A unifier σ of α and β is called *most general* if $\sigma'(\alpha)$ is a substitution instance of $\sigma(\alpha)$ for every other unifier σ' of α and β. In this case, we call $\sigma(\alpha)$ the *most general unification* of α and β.

The rule **D** (condensed detachment) can then be given as:

$$\frac{\alpha \rightarrow \beta \qquad \gamma}{\sigma(\beta)} \tag{D}$$

where γ' is an alphabetic variant of γ which has no common variables with α, and where σ is a most general unifier of α and γ' for which no new variable in $\sigma(\alpha)$ occurs in β.[4] If rule **D** can be applied to $\alpha \rightarrow \beta$ and γ, we write $\mathbf{D}(\alpha \rightarrow \beta, \gamma) = \sigma(\beta)$.[5]

As an example for an application of rule **D**, let the major premise $\alpha \rightarrow \beta$ be $(a \rightarrow (b \rightarrow c)) \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow c))$, and let the minor premise γ be the formula $p \rightarrow p$. As

https://doi.org/10.1017/S0960129523000191 Published online by Cambridge University Press

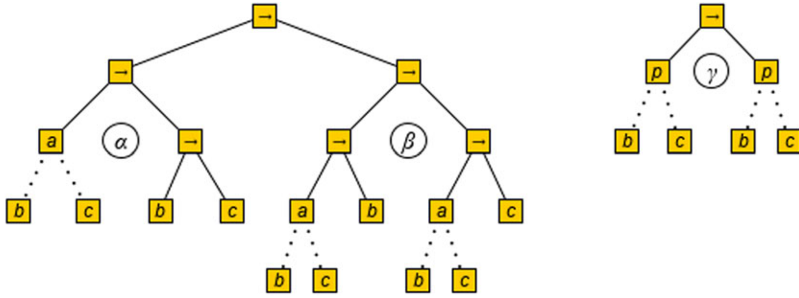

Figure 1. Rule **D** applied to the major premise $\alpha \to \beta = (a \to (b \to c)) \to ((a \to b) \to (a \to c))$ and the minor premise $\gamma = p \to p$, both represented as binary trees. The resulting formula is $\sigma(\beta) = ((b \to c) \to b) \to ((b \to c) \to c)$.

α and γ share no common variables, we can immediately proceed with their actual unification, which is realized by the most general unifier $\sigma = \{a \rightsquigarrow (b \to c), p \rightsquigarrow (b \to c)\}$. Thus, the resulting formula $\sigma(\beta)$ is $((b \to c) \to b) \to ((b \to c) \to c)$.

The representation of the just mentioned formulas as binary trees leads to the picture in Fig. 1, in which the original trees α and γ successfully try to grow into the same overall shape or structure. Note how the subtree β follows the same "growing rules" as α and γ do. Finally, variables in inner nodes will have to be replaced by the implication operator.

2.2 Combinatory logic and principal types

Terms of combinatory logic (*CL-terms*) are built according to the following production rule:

$$\varphi ::= \mathsf{S} \mid \mathsf{K} \mid (\varphi\, \varphi), \tag{2}$$

the third option being called an *application*. Example CL-terms are S, (KS), and ((KK)(KS)), the last one of which is usually simply written as KK(KS) following an association to the left.

Two rules of *weak reduction*[6] can be applied to CL-terms, and while the actual terms can be interpreted as algorithms, it is the reduction rules which bring in the dynamics of the computation. These rules are

$$\mathsf{S}xyz \rhd xz(yz), \tag{3}$$

$$\mathsf{K}xy \rhd x, \tag{4}$$

where the metavariables x, y, and z represent arbitrary CL-terms. Applying these rules, an example reduction leading to what is called an unreducible *normal form* might look as follows:

$$\mathsf{K(SKSS)K} \rhd \mathsf{K(KS(SS))K} \rhd \mathsf{KSK} \rhd \mathsf{S}$$

The reader should note that we could also have chosen a different reduction path, which, nevertheless, would have led to the same normal form, a property of combinatory logic known as the *Church–Rosser property*:

$$\mathsf{K(SKSS)K} \rhd \mathsf{SKSS} \rhd \mathsf{KS(SS)} \rhd \mathsf{S}$$

In *typed* combinatory logic, *principal types* can be assigned to certain CL-terms. These types follow the building law given by production rule (1), that is, every type can be interpreted as an implicational proposition. Two axioms assign the principal types $(a \to (b \to c)) \to ((a \to b) \to (a \to c))$ and $a \to (b \to a)$ (or any other alphabetic variant) to the CL-terms S and K, respectively. If two CL-terms M and N with principal types $\varphi \to \psi$ and ϑ are given, the principal type $\mathbf{D}(\varphi \to \psi, \vartheta)$, if it exists, is assigned to the application $(M\,N)$. Otherwise, $(M\,N)$ is regarded as not typable. The typing scheme is summarized in Fig. 2.

As an example, the tree in Fig. 3 assigns the principal type $(a \to a)$ to the CL-term SKK.

https://doi.org/10.1017/S0960129523000191 Published online by Cambridge University Press

$$\frac{}{\mathsf{S} : (a \to (b \to c)) \to ((a \to b) \to (a \to c))} \text{(Ax)}$$

$$\frac{}{\mathsf{K} : a \to (b \to a)} \text{(Ax)}$$

$$\frac{M : \varphi \to \psi \qquad N : \vartheta}{(M\,N) : \mathbf{D}(\varphi \to \psi, \vartheta)} \text{(pt)}$$

Figure 2. Assigning principal types to terms in CL.

$$\dfrac{\dfrac{\mathsf{S} : (a \to (b \to c)) \to ((a \to b) \to (a \to c)) \qquad \mathsf{K} : p \to (q \to p)}{\mathsf{SK} : (p \to q) \to (p \to p)} \text{(pt)} \qquad \mathsf{K} : a \to (b \to a)}{\mathsf{SKK} : a \to a} \text{(pt)}$$

Figure 3. A tree assigning the principal type $(a \to a)$ to the CL-term SKK, using two applications of rule (pt). The substitution in the first step is $\{a \rightsquigarrow p, b \rightsquigarrow q, c \rightsquigarrow p\}$ and in the second step $\{p \rightsquigarrow a, q \rightsquigarrow (b \to a)\}$.

2.3 The unification algorithm

Without going into two many details, we now present a version of the unification algorithm which is based on a variant given in Russell and Norvig (2010) and which, in our case, computes a most general unifier of two implicational propositional formulas α and β. Basically, Algorithm 1 considers four cases: if α and β are equal, the substitution built up so far is returned. If one of the formulas is a variable, the two formulas are passed on to Algorithm 2, *unify-var*, which unifies the variable with the second formula. Otherwise, the *left* and *right* parts of α and β, that is, their *antecedent* and *consequent*, are recursively unified.

Two things are worth noting about this algorithm. First, unification is "nearly" symmetrical in α and β, the only difference between α and β which breaks the symmetry being that the test if α is a variable comes *before* the test if β is a variable. Thus, in certain cases, *unify*(α, β) may lead to a result different from the one returned by *unify*(β, α).[7] However, it can be shown that the resulting unified formulas are always alphabetic variants of each other. Second, the algorithm is apparently far from being trivial: as Peter Norvig notes (Norvig 1991), many textbooks in the 1980s presented[8] versions of the algorithm which, for example, failed to unify the formulas $(a \to (b \to a)) \to (b \to (a \to b))$ and $p \to p$ due to a missing *dereferencing* clause when unifying variables (lines 3/4 in Algorithm 2).

Algorithm 1 unify(α, β, σ)

Input: α, β: implicational formulas
σ: the substitution built up so far (optional, defaults to empty)
Output: a substitution unifying α and β, or failure

```
1:  if σ = failure then
2:      return failure
3:  else if α = β then
4:      return σ
5:  else if variable?(α) then
6:      return unify-var(α, β, σ)
7:  else if variable?(β) then
8:      return unify-var(β, α, σ)
9:  else
10:     return unify(right(α), right(β), unify(left(α), left(β), σ))
11: end if
```

https://doi.org/10.1017/S0960129523000191 Published online by Cambridge University Press

Algorithm 2 unify-var(var, β, σ)

Input: var: a variable
β: an implicational formula
σ: a substitution
Output: a (modified) substitution
1: **if** $\langle var \rightsquigarrow \gamma \rangle \in \sigma$ **then**
2: **return** unify(γ, β, σ)
3: **else if** $\langle \beta \rightsquigarrow \gamma \rangle \in \sigma$ **then**
4: **return** unify(var, γ, σ)
5: **else if** occur-check?(var, β, σ) **then**
6: **return** failure
7: **else**
8: **return** $\sigma \cup \{\langle var \rightsquigarrow \beta \rangle\}$
9: **end if**

3. Pattern Matching, Pattern Detachment, and Pattern Typing

As mentioned in the Introduction, we now shift the focus to the unidirectional sibling of the unification algorithm, which is known as *pattern matching*. Pattern matching regards one of the given formulas as a fixed and unchangable pattern to which the other formula has to be matched using appropriate variable substitutions.[9] This asymmetry is reflected in Algorithm 3 (*match*) by the difference between the clause in lines 5/6 on the one hand and the clause in lines 7/8 on the other hand: if β is a variable but α is not, the matching is doomed to fail because β is not allowed to grow. Furthermore, Algorithm 4 (*match-var*) fails if a variable var has to be substituted by an expression β, but the substitution built so far already contains a binding $\langle var \rightsquigarrow \gamma \rangle$ for which $\beta \neq \gamma$ holds. In this case, there is no possibility to unify β and γ.

Algorithm 3 match(α, β, σ)

Input: α, β: implicational formulas
σ: the substitution built up so far (optional, defaults to empty)
Output: a substitution that makes α identical to β, or failure
1: **if** σ = failure **then**
2: **return** failure
3: **else if** $\alpha = \beta$ **then**
4: **return** σ
5: **else if** variable?(α) **then**
6: **return** match-var(α, β, σ)
7: **else if** variable?(β) **then**
8: **return** failure
9: **else**
10: **return** match(right(α), right(β), match(left(α), left(β), σ))
11: **end if**

While pattern matching is a widely used technique in Artificial Intelligence and related fields, it does not play an apparent role when it comes to logical rules of inference in propositional logic or to type inference in combinatory logic. In what follows, we take a closer look at the consequences of replacing the unification algorithm by pattern matching as the underlying mechanism of these kinds of inferences. To start with, we define an inference *rule* **P** (*pattern detachment*) as follows:

$$\frac{\alpha \to \beta \qquad \gamma}{\sigma(\beta)} \qquad (\mathbf{P})$$

https://doi.org/10.1017/S0960129523000191 Published online by Cambridge University Press

Algorithm 4 match-var(var, β, σ)

Input: var: a variable
β: an implicational formula
σ: a substitution
Output: a (modified) substitution
1: **if** $\langle var \rightsquigarrow \gamma \rangle \in \sigma$ **then**
2: **if** $\beta = \gamma$ **then**
3: **return** σ
4: **else**
5: **return** failure
6: **end if**
7: **else**
8: **return** $\sigma \cup \{\langle var \rightsquigarrow \beta \rangle\}$
9: **end if**

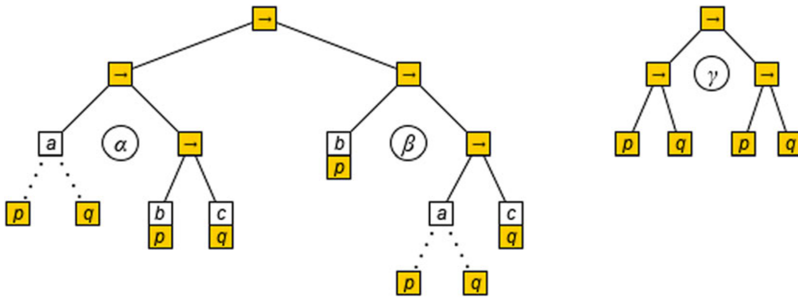

Figure 4. Rule **P** applied to the major premise $\alpha \to \beta = (a \to (b \to c)) \to (b \to (a \to c))$ and the minor premise $\gamma = (p \to q) \to (p \to q)$, both represented as binary trees. The resulting formula is $\sigma(\beta) = p \to ((p \to q) \to q)$. Note that the minor premise γ remains in its fixed shape.

Here, if it exists, σ is the substitution for which $\sigma(\alpha) = \gamma'$ holds, where γ' is an alphabetic variant of γ which does not share any variables with β.[10] In this case, we write $\mathbf{P}(\alpha \to \beta, \gamma) = \sigma(\beta)$, otherwise $\mathbf{P}(\alpha \to \beta, \gamma)$ remains undefined. The reader should note that if $\mathbf{P}(\alpha \to \beta, \gamma)$ is defined, then also $\mathbf{D}(\alpha \to \beta, \gamma)$ (condensed detachment) is defined and $\mathbf{P}(\alpha \to \beta, \gamma) = \mathbf{D}(\alpha \to \beta, \gamma)$ holds.

An example of pattern detachment is shown in Fig. 4, where the formulas $(a \to (b \to c)) \to (b \to (a \to c))$ and $(p \to q) \to (p \to q)$, both represented as binary trees, lead to the result $p \to ((p \to q) \to q)$, the (most general) unifier being the substitution $\sigma = \{a \rightsquigarrow (p \to q), b \rightsquigarrow p, c \rightsquigarrow q\}$.

When it comes to typed combinatory logic, the notion of *pattern typing* can be defined exactly as in Fig. 2, the third rule (pt) being replaced by the following rule (ptt):

$$\frac{M : \varphi \to \psi \qquad N : \vartheta}{(M\,N) : \mathbf{P}(\varphi \to \psi, \vartheta)}\ \text{(ptt)}$$

If $\mathbf{P}(\varphi \to \psi, \vartheta)$ is not defined, then the CL-term (MN) is not pattern typable. If a CL-term Q is pattern typable, we use the notation $ptt(Q)$ for its pattern type.

As an example for pattern typing, we refer the reader back to Fig. 3, where we simply replace the annotation (pt) by (ptt), stressing the fact that in this example both applications of rule (ptt) regard the type of the combinator K (the minor premise) as a fixed pattern, to which the antecedent of the types of the combinators S and SK have to be matched. The example demonstrates that every pattern type also is a principal type of its CL-term.

https://doi.org/10.1017/S0960129523000191 Published online by Cambridge University Press

4. Pattern Types and Weak Reduction

It is a widely known fact that weak reduction is capable to change the principal type of a CL-term. To be more precise, if φ is the principal type of a CL-term M which weakly reduces to N (i.e., $M \rhd N$), then N has a principal type ψ of which φ is a substitution instance.[11] As an example, consider the CL-term SKSI (where I abbreviates the term SKK). This term reduces to KI(SI). But while the former term has the principal type $(a \to b) \to (a \to b)$, the latter has the principal type $a \to a$. If we consider CL-terms as programs and a reduction step as part of a computation, then the possible change of the program's principal type during a computation may well disturb our picture of a type as a way to characterize the input and output of the computation. At this very point, it should be remarkable to note that the pattern type of a CL-term never changes when the term is reduced.

Theorem 1. *Let M and N be two CL-terms with $M \rhd N$. If M has pattern type φ, then N is also pattern typable and has the same pattern type φ.*

Proof. We consider three cases. In case 1, M is the CL-term $\mathsf{S}xyz$, x, y, and z denoting arbitrary CL-terms. If $\mathsf{S}xyz$ is pattern typable, x needs to have a pattern type of the structure $\sigma_1(a \to (b \to c))$ so that the type of S can match this fixed pattern in the first step:

$$\frac{\mathsf{S} : (a \to (b \to c)) \to ((a \to b) \to (a \to c)) \qquad x : \sigma_1(a \to (b \to c))}{\mathsf{S}x : \sigma_1((a \to b) \to (a \to c))}\ \text{(ptt)}$$

Similar arguments lead us to the structure of the types of y and z, and finally to the pattern type of $\mathsf{S}xyz$:

$$\frac{\dfrac{\mathsf{S}x : \sigma_1((a \to b) \to (a \to c)) \qquad y : \sigma_2 \circ \sigma_1(a \to b)}{\mathsf{S}xy : \sigma_2 \circ \sigma_1(a \to c)}\ \text{(ptt)} \qquad z : \sigma_3 \circ \sigma_2 \circ \sigma_1(a)}{\mathsf{S}xyz : \sigma_3 \circ \sigma_2 \circ \sigma_1(c)}\ \text{(ptt)}$$

As $\mathsf{S}xyz$ reduces to $xz(yz)$, we can now construct the pattern type of the reduct, taking into consideration the appropriate structures of the types of x, y, and z. As we can see, the deduction results in the same pattern type as before:

$$\frac{\dfrac{x : \sigma_1(a \to (b \to c)) \qquad z : \sigma_3 \circ \sigma_2 \circ \sigma_1(a)}{xz : \sigma_3 \circ \sigma_2 \circ \sigma_1(b \to c)}\ \text{(ptt)} \qquad \dfrac{y : \sigma_2 \circ \sigma_1(a \to b) \qquad z : \sigma_3 \circ \sigma_2 \circ \sigma_1(a)}{yz : \sigma_3 \circ \sigma_2 \circ \sigma_1(b)}\ \text{(ptt)}}{xz(yz) : \sigma_3 \circ \sigma_2 \circ \sigma_1(c)}\ \text{(ptt)}$$

In case 2, M is the CL-term $\mathsf{K}xy$, with x and y again denoting CL-terms. For $\mathsf{K}xy$ to be pattern typable, x and y may now have arbitrary pattern types, which we denote as $\sigma_1(a)$ and $\sigma_2(b)$, respectively. Here, we stipulate that

- σ_1 does not replace any other variables than a,
- $\sigma_1(a)$ does not contain the variable b,
- σ_2 does not replace any other variables than b,

which, for example, implies that $\sigma_1(b) = b$, a fact which is used in the following deduction:

$$\frac{\dfrac{\mathsf{K} : a \to (b \to a) \qquad x : \sigma_1(a)}{\mathsf{K}x : \sigma_1(b \to a)}\ \text{(ptt)} \qquad y : \sigma_2 \circ \sigma_1(b)}{\mathsf{K}xy : \sigma_2 \circ \sigma_1(a)}\ \text{(ptt)}$$

Due to the stipulations concerning the substitutions σ_1 and σ_2, the resulting pattern type is equal to $\sigma_1(a)$, which is the pattern type of x, which in turn is the reduct of $\mathsf{K}xy$.

https://doi.org/10.1017/S0960129523000191 Published online by Cambridge University Press

In case 3, M contains one of the redexes $\mathsf{S}xyz$ and $\mathsf{K}xy$ as a proper subterm, which is reduced to obtain the term N. Using structural induction, this case can easily be reduced to the first two cases. ■

As every pattern type is also a principal type, Theorem 1 identifies a large class of CL-terms which do not change their principal type when being reduced.

At this point, the question may arise how "natural" Theorem 1 is, or, to put it the other way round, one may wonder if the unchanged types in the above proof are just a mere coincidence. A look at other combinators presents a somewhat diffuse picture at first sight: while the combinators B and W with reduction rules $\mathsf{B}xyz \rhd x(yz)$ and $\mathsf{W}xy \rhd xyy$, respectively, preserve pattern types, the combinator C with reduction rule $\mathsf{C}xyz \rhd xzy$ and pattern type $(a \to (b \to c)) \to (b \to (a \to c))$ reveals a somewhat different behavior. Again, we can first determine the types of the terms x, y, and z as follows:

$$\frac{\mathsf{C} : (a \to (b \to c)) \to (b \to (a \to c)) \qquad x : \sigma_1(a \to (b \to c))}{\mathsf{C}x : \sigma_1(b \to (a \to c))}\,\text{(ptt)}$$

$$\frac{\dfrac{\mathsf{C}x : \sigma_1(b \to (a \to c)) \qquad y : \sigma_2 \circ \sigma_1(b)}{\mathsf{C}xy : \sigma_2 \circ \sigma_1(a \to c)}\,\text{(ptt)} \qquad z : \sigma_3 \circ \sigma_2 \circ \sigma_1(a)}{\mathsf{C}xyz : \sigma_3 \circ \sigma_2 \circ \sigma_1(c)}\,\text{(ptt)}$$

Trying to pattern type the redex xzy then leads to the insight that a pattern type might not exist if σ_3 is a substitution that modifies the type $\sigma_2 \circ \sigma_1(b \to c)$:

$$\frac{\dfrac{x : \sigma_1(a \to (b \to c)) \qquad z : \sigma_3 \circ \sigma_2 \circ \sigma_1(a)}{xz : \sigma_3 \circ \sigma_2 \circ \sigma_1(b \to c)}\,\text{(ptt)} \qquad y : \sigma_2 \circ \sigma_1(b)}{xzy : ?}\,\text{(ptt)}$$

As an example, we can now consider the term CTIK with pattern type $(a \to (b \to a))$, where T abbreviates the term $\mathsf{C}(\mathsf{SK}(\mathsf{SK}))$ with pattern type $(a \to ((a \to b) \to b))$. It is now easy to see that the reduct TKI is not pattern typable.[12]

The following theorem characterizes exactly those combinators which guarantee that reduction does preserve pattern types. If M is a combination of terms taken from the set $\{x_1, \ldots, x_n\}$, $index(M)$ denotes the greatest x-index occuring within M. For example, $index(x_1x_3(x_2x_3)) = 3$.

Theorem 2. *Let P be a combinator with reduction rule $Px_1, \ldots, x_n \rhd Q$ where Q is a combination of the variables $x_1, \ldots, x_n$. Then, for any CL-terms $x_1, \ldots, x_n$, Q is pattern typable with the same pattern type as $Px_1, \ldots, x_n$, if, for every subterm (MN) of Q, the inequation $index(M) \leq index(N)$ holds.*

Before we move on to the Theorem's proof, two examples may help to clarify the central point. First, the just regarded combinator C with reduction rule $\mathsf{C}x_1x_2x_3 \rhd x_1x_3x_2$ violates the subterm condition in the term $((x_1x_3)x_2)$ because $index(x_1x_3) > index(x_2)$. Second, the combinator B' with reduction rule $\mathsf{B}'x_1x_2x_3 \rhd x_2(x_1x_3)$ does indeed guarantee the preservation of pattern types, as can now immediately be read off the indices of the structure $(x_2(x_1x_3))$. We now proceed to the proof of Theorem 2.

Proof. In the first step, pattern typing of the term $Px_1 \ldots x_n$ leads to the following deduction, as can be shown by natural induction on n. Thus, every term x_i $(1 \leq i \leq n)$ has pattern type $\sigma_i\sigma_{i-1} \ldots \sigma_1(\varphi_i)$ for some type φ_i:[13]

https://doi.org/10.1017/S0960129523000191 Published online by Cambridge University Press

$$\dfrac{\dfrac{\dfrac{\dfrac{P:\varphi_1\to\varphi_1' \qquad x_1:\sigma_1(\varphi_1)}{Px_1:\sigma_1(\varphi_2\to\varphi_2')}\,\text{(ptt)} \qquad x_2:\sigma_2\sigma_1(\varphi_2)}{Px_1x_2:\sigma_2\sigma_1(\varphi_3\to\varphi_3')}\,\text{(ptt)}}{}}{}$$

$$\dfrac{\ddots \qquad\qquad x_n:\sigma_n\ldots\sigma_1(\varphi)}{Px_1\ldots x_n:\sigma_n\ldots\sigma_1(\varphi)}\,\text{(ptt)}$$

In the second step, we now consider subterms (MN) of Q with $\underbrace{index(M)}_{=:k} \leq \underbrace{index(N)}_{=:l}$. By structural induction, we can show that (MN) has pattern type $\sigma_l\ldots\sigma_1(\psi)$ for some type ψ. As the base case, we consider the subterm (x_kx_l) for two variables $x_k:\sigma_k\ldots\sigma_1(\varphi_k)$ and $x_l:\sigma_l\ldots\sigma_1(\varphi_l)$. As $k\leq l$ holds by the assumption of the theorem, we get $(x_kx_l):\sigma_l\ldots\sigma_1(\psi)$. For the induction step, we consider two terms M and N, which, by the induction hypothesis, have pattern types $\sigma_k\ldots\sigma_1(\varphi_M)$ and $\sigma_l\ldots\sigma_1(\varphi_N)$, respectively. Again, $k\leq l$ holds, and we conclude that the term (MN) has pattern type $\sigma_l\ldots\sigma_1(\psi)$. In the case of the complete term Q, this is exactly the type of the term $Px_1\ldots x_n$. ■

5. The Pattern Model of Untyped Combinatory Logic

Theorem 1 of the preceding section now gives rise to a model of untyped combinatory logic which can be regarded as a simplification of the Fool's model (Meyer et al. 1991). To start with, we define the notion of a *combinatory reduction algebra.*

Definition 1. *A* combinatory reduction algebra *is an algebraic structure* $(\mathcal{A},\leq,\bullet,s,k)$ *where* $\mathcal{A}$ *(the* domain*) is a nonempty set,* $\leq$ *is a partial order on* $\mathcal{A}$*,* $\bullet\colon\mathcal{A}\times\mathcal{A}\to\mathcal{A}$ *is a binary operation on* $\mathcal{A}$*, and* s *and* k *are two distinct elements of* $\mathcal{A}$*, and where the following stipulations hold for all* $x,y,z\in\mathcal{A}$*:*

$$s\bullet x\bullet y\bullet z\leq x\bullet z\bullet(y\bullet z) \tag{5}$$

$$k\bullet x\bullet y\leq x \tag{6}$$

$$x\leq y \text{ implies } z\bullet x\leq z\bullet y \tag{7}$$

$$x\leq y \text{ implies } x\bullet z\leq y\bullet z \tag{8}$$

Next, we interpret CL-terms (the set of which we denote by $\mathbb{T}$) by means of the elements of a combinatory reduction algebra.

Definition 2. *A* model *of untyped combinatory logic is a pair* $(\mathfrak{A},\mathfrak{I})$ *where* $\mathfrak{A}=(\mathcal{A},\leq,\bullet,s,k)$ *is a combinatory reduction algebra, and* $\mathfrak{I}\colon\mathbb{T}\to\mathcal{A}$ *is a mapping for which the following equalities hold:*

$$\mathfrak{I}(\mathsf{S})=s \tag{9}$$

$$\mathfrak{I}(\mathsf{K})=k \tag{10}$$

$$\mathfrak{I}((PQ))=\mathfrak{I}(P)\bullet\mathfrak{I}(Q) \text{ for all CL-terms } P,Q \tag{11}$$

In order to introduce the new model, which we will call the *pattern model*, we first identify propositional formulas which are alphabetic variants of each other. To this end, we use the notation $\{\varphi\}_\alpha$ to denote the set of all formulas which are alphabetic variants of φ. These kind of sets actually form the domain of the pattern model.

https://doi.org/10.1017/S0960129523000191 Published online by Cambridge University Press

Definition 3. *The* pattern model *is the pair* $(\mathfrak{P}, \mathfrak{I})$, *where*

$$\mathfrak{P} = (\mathcal{P}, \subseteq, \bullet, s, k), \tag{12}$$

$$\mathcal{P} = \left\{\{\varphi\}_\alpha \mid \varphi \text{ is an implicational propositional formula}\right\} \cup \{\emptyset\}, \tag{13}$$

$$\subseteq \text{ is ordinary set inclusion}, \tag{14}$$

$$x \bullet y = \left\{\vartheta \mid \exists\varphi \in x, \exists\psi \in y : \mathbf{P}(\varphi, \psi) = \vartheta\right\}_\alpha \text{ for all } x, y \in \mathcal{P}, \tag{15}$$

$$s = \left\{(a \to (b \to c)) \to ((a \to b) \to (a \to c))\right\}_\alpha, \tag{16}$$

$$k = \left\{a \to (b \to a)\right\}_\alpha, \tag{17}$$

$$\mathfrak{I}(Q) = \left\{ \begin{array}{ll} \left\{ptt(Q)\right\}_\alpha & \text{if } Q \text{ is pattern typable} \\ \emptyset & \text{otherwise} \end{array} \right\} \text{ for all CL-terms } Q. \tag{18}$$

A few remarks should be appropriate at this point.

- ◇ As defined in (13), the domain $\mathcal{P}$ of $\mathfrak{P}$ can simply be regarded as the set of implicational propositional formulas *modulo* alphabetic variation. $\mathcal{P}$ thus consists of equivalence classes of formulas which have the same "logical power."[14]
- ◇ If x and y are elements of the domain $\mathcal{P}$, $x \subseteq y$ only holds if $x = y$ or $x = \emptyset$.
- ◇ As defined in (15), $x = \emptyset$ or $y = \emptyset$ immediately lead to the fact that $x \bullet y = \emptyset$. This corresponds to the fact that any CL-term, which has an pattern-untypable subterm, is itself pattern-untypable.
- ◇ If both $x \neq \emptyset$ and $y \neq \emptyset$, then $x \bullet y$ is the result of (possibly) applying rule **P** (pattern detachment) to two representatives of the sets x and y, finally collecting the alphabetic variants of the result.[15]
- ◇ Concerning the role of the empty set $\emptyset$, computer experiments have been able to show that 310,210 out of 397,242 terms – with a maximum of eight applications – are indeed pattern typable and are thus not represented by the empty set.

Theorem 3. *The pattern model is a model of untyped combinatory logic.*

Proof. We first show that for every pattern model $(\mathfrak{P}, \mathfrak{I})$, $\mathfrak{P}$ is a combinatory reduction algebra. To this end, we show that stipulations (5) and (7) hold. Stipulations (6) and (8) follow in exactly the same manner.

For stipulation (5), we consider two cases. In the first case, the expression $s \bullet x \bullet y \bullet z$ is equal to the empty set. In this case, set inclusion is trivially fulfilled. Otherwise, one can successfully apply rule **P** three times to representatives of the sets s, x, y, and z, yielding a nonempty set of alphabetic variants of a formula ϑ. But now Theorem 1 guarantees that the expression $x \bullet z \bullet (y \bullet z)$ leads to exactly the same result. Again, set inclusion holds.

For stipulation (7), we also consider two cases. In the first case, $x \subseteq y$ holds because x is the empty set. But in this case, the expression $z \bullet x$ is also equal to the empty set, and thus $z \bullet x \subseteq z \bullet y$ is fulfilled. In the second case, $x \subseteq y$ holds because x and y are equal. But then $z \bullet x$ and $z \bullet y$ are also equal.

In the next step, we show that the interpretaion $\mathfrak{I}$ does indeed give rise to a model. As equalities (9) and (10) obviously hold, we can concentrate on equality (11). This time, we consider three cases. In case 1, at least one of the CL-terms P and Q is not pattern typable. This implies that the application (PQ) is not pattern typable, either. Thus, both sides of the equation evaluate to the empty set. In case 2, both P and Q are pattern typable, but the application (PQ) is not. Again, both sides of the equation yield the empty set, the right-hand side because the sets $\mathfrak{I}(P)$ and $\mathfrak{I}(Q)$ do not contain any formula to which rule **P** can be applied. In the third case, P, Q, and (PQ) are pattern

https://doi.org/10.1017/S0960129523000191 Published online by Cambridge University Press

typable. Thus, the term $\mathfrak{I}(P) \bullet \mathfrak{I}(Q)$ evaluates exactly to the set of alphabetic variants of the pattern type of the application (PQ). ■

We conclude this section with a short comparison between the pattern model and the Fool's model. Details about the latter can be found in Meyer et al. (1991) and Bimbo (2011, Chapter 6). The Fool's model can actually be obtained by simply changing Definition 3 as follows:

(1) Change the index α to an index s all the way through, where the notation $\{\varphi\}_s$ represents the set of all substitution instances of φ.
(2) In (18), change "*ptt*" and "pattern typable" into "*pt*" and "principal-typable," respectively.

The advantages of the pattern model over the Fool's model now lie at hand: First, the former is based on the very natural domain consisting of propositional formulas (modulo the renaming of variables). Second, two arbitrary CL-terms P and Q with $P \rhd Q$ are always represented by the same object, thus supporting the idea that a computation is not supposed to change its input and output types while running.

6. Major versus Minor Premisses as Fixed Patterns

In this last section, we answer the question why we have decided to consider the *minor* premise as the fixed and unchangable pattern both in rule **P** and in rule (ptt). The main reason is that any proof which also allows for the consideration of the major premise as the fixed pattern can – under certain very "mild" conditions – be transformed into an equivalent proof which uses rule **P** only. To be more precise, we first define the following rule **P**′, which is applicable if and only if there is a substitution σ for which $\sigma(\gamma) = \alpha$ holds:

$$\frac{\alpha \to \beta \qquad \gamma}{\beta} \tag{$\mathbf{P}'$}$$

Using the notation $\mathbb{A} \vdash_{\mathbf{P}} \varphi$ ($\mathbb{A} \vdash_{\mathbf{P},\mathbf{P}'} \varphi$) for the existence of a proof for φ which is based on the axioms $\mathbb{A}$ and rule **P** (*and* rule **P**′, resp.), we can now state the following theorem.

Theorem 4. *If* $\mathbb{A} \vdash_{\mathbf{P},\mathbf{P}'} \varphi$ *and* $\mathbb{A} \vdash_{\mathbf{P}} (a \to ((a \to b) \to a))$, *then* $\mathbb{A} \vdash_{\mathbf{P}} \varphi$ *(for all propositional implicational formulas* φ*).*

Proof. All we need to do is to show how an application of rule **P**′ in a proof tree can be replaced by applications of rule **P** which possibly make use of the formula $a \to ((a \to b) \to a)$. Let us assume that a proof contains the following passage where $\sigma(\gamma) = \alpha$ holds for some substitution σ:

$$\cfrac{\begin{matrix}\vdots \\ \alpha \to \beta\end{matrix} \qquad \begin{matrix}\vdots \\ \gamma\end{matrix}}{\begin{matrix}\beta \\ \vdots\end{matrix}}\ (\mathbf{P}')$$

We replace this passage by the following structure:

$$\cfrac{\begin{matrix}\vdots \\ \alpha \to \beta\end{matrix} \qquad \cfrac{\cfrac{\begin{matrix}\vdots \\ a \to ((a \to b) \to a)\end{matrix} \qquad \begin{matrix}\vdots \\ \gamma\end{matrix}}{(\gamma \to b) \to \gamma}\ (\mathbf{P}) \qquad \begin{matrix}\vdots \\ \alpha \to \beta\end{matrix}}{\sigma'(\gamma)}\ (\mathbf{P})}{\begin{matrix}\beta \\ \vdots\end{matrix}}\ (\mathbf{P})$$

For this structure, we assume that b is a fresh variable with respect to α, β, γ, and the substitution σ. σ' is the substitution σ, extended by the substitution $\{b \rightsquigarrow \beta\}$, that is, $\sigma' = \sigma \cup \{b \rightsquigarrow \beta\}$. As b does not occur in γ, $\sigma'(\gamma) = \sigma(\gamma)$ holds. ■

We finish our considerations with some remarks and open questions concerning the interplay between rules **P** and **P**′ in the presence of the two axioms $(a \to (b \to c)) \to ((a \to b) \to (a \to c))$ and $a \to (b \to a)$, which, on the one hand, form a basis for the implicational fragment of intuitionistic propositional logic, and, on the other hand, serve as types for combinatory logic based on the combinators S and K.

- ◇ It remains an open question whether the formula $a \to ((a \to b) \to a)$, which plays such a crucial role in the elimination of rule **P**′, can be pattern-proved using the two just mentioned axioms. As computer experiments have shown, none of the CL-terms with less than eight combinators have pattern type $a \to ((a \to b) \to a)$.
- ◇ If the formula $(a \to (b \to c)) \to (b \to (a \to c))$, which is the principal type of the combinator C, is added as an axiom, the formula $a \to ((a \to b) \to a)$ can be pattern-proved. It is the pattern type of the CL-term CSK. It remains an open question whether the formula $(a \to (b \to c)) \to (b \to (a \to c))$ is a pattern type for the basis S and K. (Again, there is no CL-term with less than eight combinators which serves our purpose.)
- ◇ It also remains open whether the combination of rules **P** and **P**′ can replace rule **D**, condensed detachment.

Acknowledgments

The research for this article is a part of the GÖDELIANA project led by Jan von Plato, to whom I remain grateful for his encouraging support. Also, I would like to thank Anna Maiworm and Isabelle Sauer for the fruitful discussions which finally lead to the discovery of Theorems 1 and 3. Finally, I thank the anonymous referees for their careful reading of the submitted paper. The project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 787758) and from the Academy of Finland (Decision No. 318066).

Notes

1 For an early publication of the underlying algorithm, see Peterson (1978).
2 According to Hindley (1997, p. 103), it was "around 1978, when David Meredith was the first to realize the parallel."
3 To give an example, the formulas $(a \to (b \to a))$, $(p \to (q \to p))$, and $(c \to (a \to c))$ are alphabetic variants of each other.
4 That is, $(Vars(\sigma(\alpha)) \setminus Vars(\alpha)) \cap Vars(\beta) = \emptyset$.
5 For an alternative (but equivalent) definition of condensed detachment, see Bunder (1995).
6 As we only consider *weak* reduction (as opposed to *strong* reduction) in this paper, we will simply speak of *reduction*.
7 For example, $unify(a \to b, b \to a)$ returns the substitution $\{\langle a \rightsquigarrow b\rangle\}$, while $unify(b \to a, a \to b)$ returns $\{\langle b \rightsquigarrow a\rangle\}$.
8 See for example Abelson and Sussman (1984). The mistake has been corrected in the second edition.
9 In our case, the minor premise constitutes the fixed pattern to which the major premise is matched. This seemingly arbitrary choice is taken up again in Section 6.
10 Producing an alphabetic variant is needed in order to avoid unwanted variable capture. As an example, the reader may want to apply rule **P** to the premisses $a \to (b \to a)$ and $b \to b$ *without* renaming the variable b.
11 This is a direct consequence of the *subject-reduction theorem*, see for example Hindley and Seldin (1957, p. 132) or Bimbo (2011, p. 248).
12 I am indebted to the third anonymous referee, who gave the decisive hint to this example and especially to the "Thrush" combinator T.
13 For brevity, we omit the circle operator in the composition of substitutions.
14 Concerning this logical power and the role of the variables, Schönfinkel (1924, p. 307) writes: "The variable in the logical proposition is nothing more than a marker to indicate certain argument places and operators as belonging together, and

https://doi.org/10.1017/S0960129523000191 Published online by Cambridge University Press

thus has the character of a mere auxiliary item that is actually inappropriate to the constant 'eternal' nature of the logical proposition."

15 The reader should note that we could dispense with the index "α" in definition (15): As x and y contain every alphabetic variant, definition (15) automatically "produces" every alphabetic variant in the resulting set.

References

Abelson, H. and Sussman, G. J. (1984). *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge, MA.

Bimbó, K. (2011). *Combinatory Logic: Pure, Applied and Typed*, CRC Press, Boca Raton.

Bunder, M. W. (1995). A simplified form of condensed detachment. *Journal of Logic, Language, and Information* **4** (2) 169–173.

Herbrand, J. (1930). *Recherches sur la théorie de la démonstration*. Phd dissertation, University of Paris. Reprinted in: J. Herbrand, *Écrits logiques*, Paris: Presses Universitaires de France 1968, 35–153. English translation of Chapter 5 in: van Heijenoort, J. (ed.) *From Frege to Gödel: a Source Book in Mathematical Logic*, 1879–1931. Harvard University Press, Cambridge, 1967, 529–567.

Hindley, J. R. (1997). *Basic Simple Type Theory*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge.

Hindley, J. R. and Seldin, P. (2008). *Lambda-Calculus and Combinators – an Introduction*, Cambridge University Press, Cambridge.

Kalman, J. A. (1983). Condensed detachment as a rule of inference. *Studia Logica* **42** (4) 443–451.

Knight, K. (1989). Unification: A multidisciplinary survey. *ACM Computing Surveys (CSUR)* **21** (1) 93–124.

Lemmon, E. J., Meredith, C. A., Meredith, D., Prior, A. N. and Thomas, I. (1957). *Calculi of pure strict implication*. Christchurch: Canterbury University College. Reprinted in: Davis, J. W., Hockney, D. J. and Wilson, W. K. (eds.) *Philosophical Logic*, Reidel, Dordrecht, 1969, 215–250.

Meredith, D. (1977). In memoriam: Carew Arthur Meredith (1904–1976). *Notre Dame Journal of Formal Logic* **18** (4) 513–516.

Meyer, R. K., Bunder, M. W. and Powers, L. (1991). Implementing the 'Fool's model' of combinatory logic. *Journal of Automated Reasoning* **7** (4) 597–630.

Norvig, P. (1991). Correcting a widespread error in unification algorithms. *Software: Practice and Experience* **21** (2) 231–233.

Peterson, J. G. (1978). An automatic theorem prover for substitution and detachment systems. *Notre Dame Journal of Formal Logic* **19** (1) 119–122.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)* **12** (1) 23–41.

Robinson, J. A. (1979). *Logic: Form and Function – The Mechanization of Deductive Reasoning*, Edinburgh University Press, Edinburgh.

Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*, Prentice Hall Series in Artificial Intelligence, 3rd ed., Pearson, London.

Schönfinkel, M. (1924). Über die Bausteine der mathematischen Logik. *Mathematische Annalen* **92** (3) 305–316.

Cite this article: Lethen T (2023). A (machine-oriented) logic based on pattern matching. *Mathematical Structures in Computer Science* **33**, 647–659. https://doi.org/10.1017/S0960129523000191

https://doi.org/10.1017/S0960129523000191 Published online by Cambridge University Press