Journal of Philosophical Logic (2008) 37: 23–43
DOI: 10.1007/s10992-007-9051-5
© Springer 2008

WAYNE AITKEN and JEFFREY A. BARRETT

# ABSTRACTION IN ALGORITHMIC LOGIC

Received 27 October 2006

ABSTRACT. We develop a functional abstraction principle for the type-free algorithmic logic introduced in our earlier work. Our approach is based on the standard combinators but is supplemented by the novel use of evaluation trees. Then we show that the abstraction principle leads to a Curry fixed point, a statement $C$ that asserts $C \Rightarrow A$ where $A$ is any given statement. When $A$ is false, such a $C$ yields a paradoxical situation. As discussed in our earlier work, this situation leaves one no choice but to restrict the use of a certain class of implicational rules including modus ponens.

KEY WORDS: abstraction, algorithmic logic, Curry paradox

## 1. Introduction

In [2] we introduced and developed an algorithmic logic that is type-free with natural self-application. This logic was suggested by our earlier work on the Curry paradox ([1]). In light of this paradox, one expects some limitations to the accepted rules of classical logic in a system this expressive. A main aim of [2] was to investigate such limitations in algorithmic logic, but also to describe some strengths. However, the status of the abstraction principle, an important desideratum of type-free systems, was not addressed.

In this paper we formulate an internal functional abstraction principle and show it holds in algorithmic logic. We then show how any internal implication powerful enough to implement certain rules concerning evaluation will be able to implement this internal functional abstraction principle. Finally, building on [2], we show that the price of using such implications is the restriction of certain rules of classical logic; even modus ponens must be limited. In particular, the logic that emerges is more restrictive with respect to rules of implication than even intuitionist logic.[1]

Before developing the internal functional abstraction principle for algorithmic logic, we begin with two expository sections. Section 2 describes algorithmic logic. Section 3 explains what we mean by an internal functional abstraction principle, not just for algorithmic logic, but more generally.

## 2. Algorithmic Logic

Algorithmic logic is the logic of *algorithmic statements*. An algorithmic statement is a statement of the form "When algorithm $a$ is applied to input $b$, the output is $c$". To be specific about what one means by *algorithm*, *input*, and *output* one needs to fix an underlying theory of computation. We assume such an underlying theory has been chosen.[2] We use the term *datum*, plural *data*, to refer to any object that the underlying theory admits as a possible input or output for an algorithm. We suppose that the underlying theory of computation allows one to express, or code, any algorithm as a datum, so algorithms can take as input algorithms, and $a$, $b$, and $c$ above are data. Finally, we assume that the underlying theory allows as a datum any finite list of data. Then algorithmic statements can be regarded as data: the list $[a, b, c]$ expresses the algorithmic statement quoted above.[3]

A major concern of algorithmic logic is logical connectives which combine algorithmic statements into algorithmic statements. For example, if $A$ and $B$ are algorithmic statements, both $A \wedge B$ and $A \vee B$ can be expressed as algorithmic statements.[4]

In earlier work ([1] and [2]) we also introduced a more subtle logical connective: a rule-based form of implication $\overset{\rho}{\Rightarrow}$ where $\rho$ is a library of algorithmically implemented deduction rules. The algorithmic statement $A \overset{\rho}{\Rightarrow} B$ asserts that the algorithmic statement $B$ can be derived from the algorithmic statement $A$ by means of the rules contained in the library $\rho$. Thus the connective $\overset{\rho}{\Rightarrow}$ can be regarded as an algorithmic implementation of both a conditional and a provability predicate.

A theme of [1] and [2] is that certain expected rules, including the $\overset{\rho}{\Rightarrow}$-version of modus ponens, cannot be implemented in a sufficiently strong library $\rho$ without making $\rho$ invalid. The $\overset{\rho}{\Rightarrow}$-version of modus ponens is in fact a valid rule of algorithmic logic (given that $\rho$ is valid); it just cannot be in the valid library $\rho$. The present paper will show how weak a library $\rho$ can be and still be strong enough to restrict modus ponens.

In [1] we constructed an algorithmic version of the Curry paradox using the connective $\overset{\rho}{\Rightarrow}$ and a specially designed algorithm which we called CURRY. We put an ad hoc, but valid, rule in the library $\rho$ relating to the obvious behavior of CURRY, and showed that the $\overset{\rho}{\Rightarrow}$-version of modus ponens could not possibly be in $\rho$. We interpreted this as a failure of modus ponens, not as a failure of the ad hoc rule. In other words, we advocated that given a choice for inclusion in the library $\rho$ between the rule modus ponens or rules similar to the ad hoc rule, that preference be given to the latter. Indeed, in our second paper [2], we showed that rules like the ad hoc rule for CURRY have a stability property that is lacking in rules such as modus ponens.

Moreover, there is a sense in which the ad hoc rule for CURRY is just an instance of a general functional abstraction principle. Functional abstraction principles, like comprehension principles, are usually considered to be much more suspect than modus ponens, but in the present paper we show that a functional abstraction principle does indeed hold.[5] We show also that if $\rho$ contains a small collection of simple, uncontentious rules concerning obvious properties of algorithmic evaluation, the *Evaluation Rules*, then the seemingly ad hoc rule for CURRY is a actually a consequence of the functional abstraction principle. Indeed, it can be argued that rules such as the ad hoc rule for CURRY would be a (possibly unintended) consequence of any sufficiently complete library $\rho$ designed to prove results concerning algorithms. So the only safe library $\rho$ is one that does not include $\overset{\rho}{\Rightarrow}$-modus ponens.

Building on [2], this paper completes the development of algorithmic logic as a natural exemplar or model for type-free propositional logic.[6] This logic has an internal truth predicate and an internal functional abstraction principle, and allows the use of a rich collection of inferential rules. There are no concerns about consistency since the logic is built around a model. Although this logic restricts rules such as modus ponens, by varying the library $\rho$ one can avoid many of the limitations engendered by the restriction. In addition, in many situations the algorithmic version of the law of the excluded middle will hold and $\overset{\rho}{\Rightarrow}$ will be essentially equivalent to the material conditional. In these situations, the restrictions on the use of rules of classical logic can be removed by using the material conditional.

## 3. Functional Abstraction

In this section we discuss, in a general setting, functional abstraction principles and comprehension principles. We explain what we mean when we say such a principle is *internal*. Then, in the following sections, we consider the functional abstraction principle in the particular setting of algorithmic logic.

A functional abstraction principle asserts that every description of a function determines an object that instantiates that description. More specifically, given a term $\tau(x)$ in a formal language with free variable $x$, it asserts the existence of a function $f$ that corresponds to the rule $x \mapsto \tau(x)$. In a type-free setting, the function $f$ and the possible values of the variable $x$ belong to the same domain. A functional abstraction principle can be viewed as a bridge between the syntax (the term with free variable) and the semantics (the function in the domain that instantiates the term).

To make this more precise, assume for simplicity a type-free system whose semantics is represented by a fixed domain, and whose formal syntax is as economical as possible for the formulation of the principle.

We begin with the semantics. Imagine a domain $\mathcal{D}$ where some of the members of $\mathcal{D}$ are functions. If $f$ is a function in $\mathcal{D}$ and if $b \in \mathcal{D}$ is in the domain of $f$, then we define $f \cdot b$ to the image of $f$ applied to $b$. As usual, one can write $fb$ or $f(b)$ for $f \cdot b$. We call $\cdot$ the *application operator* of $\mathcal{D}$.

There is no requirement that every object in $\mathcal{D}$ be a function, nor that every function have domain equal to all of $\mathcal{D}$. Consequently, we allow the application operation to be *partial*: we do not assume $a \cdot b$ is defined for all $a, b \in \mathcal{D}$. The application operator defines a partial function from $\mathcal{D} \times \mathcal{D}$ to $\mathcal{D}$. In this general setting, we do not require that coextensive functions be equal.

The formal syntax requires variables, constants, and a binary application symbol '$\cdot$' or equivalent for term formation.[7] For the purposes of a functional abstraction principle, the formal syntax requires terms, but not formulas.

A *valuation* is an assignment of an object of $\mathcal{D}$ to each of the formal variables. Given a valuation $\sigma$ and a term $\tau$, we define $\|\tau\|_\sigma$, the *value of* $\tau$ *under* $\sigma$, in the usual way. The result, when it is defined, is an object of $\mathcal{D}$. Since $\cdot$ can be partial, we allow for the possibility that $\|\tau\|_\sigma$ is undefined for some terms $\tau$.

Let $\mathcal{T}$ be the set of all terms. A *functional abstraction principle* asserts the existence of operators $\lambda_x : \mathcal{T} \to \mathcal{T}$, one for each variable $x$, sending a term $\tau$ to a term $\lambda_x \tau$ denoting the function defined by the rule $x \mapsto \tau$. To be more precise, for every term $\tau$ and valuation $\sigma$,

(a) $\lambda_x \tau$ does not contain $x$ as a free variable,
(b) $\|\lambda_x \tau\|_\sigma$ is defined,
(c) $\|\lambda_x \tau \cdot x\|_\sigma$ is defined if and only if $\|\tau\|_\sigma$ is defined, and
(d) if $\|\tau\|_\sigma$ is defined, then

$$\|\lambda_x \tau \cdot x\|_\sigma = \|\tau\|_\sigma.$$

Note that $\lambda_x$ is not assumed to be a symbol in the formal syntax, but rather a function $\mathcal{T} \to \mathcal{T}$. It is a classical result of combinatory logic that if the domain $\mathcal{D}$ has objects corresponding to the traditional $K$ and $S$ combinators, and if the formal syntax has constants assigned to $K$ and $S$, then $\lambda_x$ can be defined in a uniform manner such that the above principle holds. The best known case is where application $\cdot$ is total (see [4]), but the partial case holds as well (see Chapter VI of [3]).

A functional abstraction principle yields a corresponding principle, often called a *comprehension principle*, for properties. For this, fix two objects of $\mathcal{D}$ to represent truth and falsity. Functions on $\mathcal{D}$ whose values are in the set containing these two objects are *Boolean functions*. The comprehen-

sion principle states that certain formal descriptions of properties determine Boolean functions that instantiates the descriptions.

Properties represented by Boolean functions in $\mathcal{D}$ are *internal properties*. The more internal properties that $\mathcal{D}$ possesses, the more expressive the type-free system. In algorithmic logic, for instance, implication and other logical connectives can be formulated in terms of internal properties.

One might also want to express a functional abstraction principles itself in terms of internal properties. Such a functional abstraction principle will be said to be *internal*.

A main obstacle for the existence of an internal functional abstraction principle is that the functional abstraction principle makes use of $\|\lambda_x\tau{\cdot}x\|_\sigma$ and $\|\tau\|_\sigma$, expressions that are not always defined; they do not always denote objects in the semantics $\mathcal{D}$. In the present paper we replace these expressions with terms that uniformly denote objects in $\mathcal{D}$. We do so by employing *application trees*.

Application trees make possible, for each valuation $\sigma$, an interpretation of *all terms* $\tau$, even when the application operator is partial. For example, $\|x{\cdot}y\|_\sigma$ is usually defined as the result of applying $\|x\|_\sigma$ to $\|y\|_\sigma$. When the application operation is partial, expression such as $\|x{\cdot}y\|_\sigma$ might be undefined. With trees, however, we will interpret $|x{\cdot}y|_\sigma$ not as the result of applying $|x|_\sigma$ to $|y|_\sigma$, but instead as a tree with nodes $|x|_\sigma$ and $|y|_\sigma$. When that tree is *evaluated*, we recover $\|x{\cdot}y\|_\sigma$, assuming the tree successfully evaluates.

Note that we use double bars, as in $\|\tau\|_\sigma$, for the traditional interpretation, and we use single bars, as in $|\tau|_\sigma$, for the tree interpretation. To be able to define such a tree interpretation, our domain $\mathcal{D}$ must have objects instantiating trees. This can be done in algorithmic logic using lists.

An internal abstraction principle formulated in terms of trees requires a notion of *tree equivalence* $\sim$ such that if $T_1 \sim T_2$, where $T_1$ and $T_2$ are trees in $\mathcal{D}$, then (a) $T_1$ evaluates if and only if $T_2$ evaluates, and (b) if they evaluate then they evaluate to the same object of $\mathcal{D}$. Of course, the tree equivalence $\sim$ needs to be definable in terms of internal properties. In this setting, an *internal functional abstraction principle* asserts that

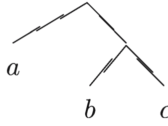
$$\left|\lambda_x\tau{\cdot}x\right|_\sigma \;\sim\; |\tau|_\sigma$$

for all valuations $\sigma$ and all terms $\tau$.

## 4. Evaluation Trees

For the remainder of the paper, we will adopt the notations and conventions above and of [2].

Let $a, b$, and $c$ be three pieces of data where $a$ and $b$ are algorithms. Suppose one applies the algorithm $b$ to the input $c$, then applies $a$ to the resulting output. A natural way to describe this sequence of evaluations is with a tree:




Such an *evaluation tree* can be thought of as a description of how to organize the evaluation of data.

Evaluation trees themselves can be thought of as a type of datum in algorithmic logic. This can be accomplished by representing a node of a tree as a list of length two, and a terminus as a singleton list.

DEFINITION 4.1. An *evaluation tree* is defined recursively as follows.

(a) A singleton list $[c]$ is an evaluation tree. This type of tree is called a *simple tree*.
(b) If $T_1$ and $T_2$ are evaluation trees, then so is $[T_1, T_2]$.

CONVENTION 4.2. The simple tree $[c]$ is usually written $\mathrm{Tree}(c)$. If $T_1$ and $T_2$ are trees, we usually write $T_1 \cdot T_2$ for the tree $[T_1, T_2]$.

CONVENTION 4.3. We abbreviate $T \cdot \mathrm{Tree}(c)$ as $T \cdot c$, and $\mathrm{Tree}(c) \cdot T$ as $c \cdot T$, when context allows.[8]

For example, the above evaluation tree is formally the nested list

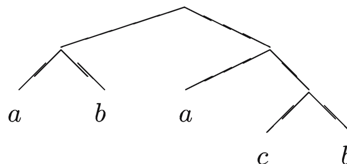
$$\Big[[a], \big[[b], [c]\big]\Big],$$

but is more conveniently represented by $a \cdot (b \cdot c)$.

For a more complex example,

$$\Bigg[\Big[[a], [b]\Big], \Big[[a], \big[[c], [b]\big]\Big]\Bigg]$$

is the datum corresponding to the tree



which can also be written as $(a \cdot b) \cdot \big(a \cdot (c \cdot b)\big)$.

DEFINITION 4.4. Given an evaluation tree $T$, the *evaluation of T* is recursively defined as follows:

(a) The simple tree $\text{Tree}(c)$ evaluates to $c$.
(b) If $T_1$ and $T_2$ are evaluation trees, then $T_1 \cdot T_2$ evaluates if and only if

   (1) $T_1$ evaluates to some datum, say $a_1$,
   (2) $T_2$ evaluates to some datum, say $a_2$, and
   (3) $a_1$ halts when applied to input $a_2$.

   And, if these conditions hold, then $T_1 \cdot T_2$ evaluates to the output resulting from applying $a_1$ to the input $a_2$.

DEFINITION 4.5. Fix an *evaluation algorithm* EVAL that does the following. EVAL expects as input a tree $T$. If $T$ evaluates to $c$ then EVAL outputs $c$. If $T$ does not evaluate, then EVAL does not halt. The algorithmic statement $[\text{EVAL}, T, c]$ is written $T \mapsto c$.

In this notion, the algorithmic statement $\text{Tree}(c) \mapsto c$ is true for all data $c$. Suppose $[a, b, c]$ is a true algorithmic statement, $T_1 \mapsto a$, and $T_2 \mapsto b$. Then $T_1 \cdot T_2 \mapsto c$.

## 5. The Basic Combinators

In this section we define algorithmic versions of the traditional combinators $K$ and $S$. In addition, we define a tree corresponding to the standard identity combinator $I$.

DEFINITION 5.1. Let $a$ be a datum. An *a-constant algorithm* is one that, when applied to any datum, halts with output $a$.

Let $k$ be an algorithm that, when applied to an input $a$, halts with an $a$-constant algorithm as the output. In other words, for all data $a$ and $b$,

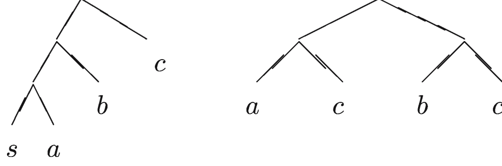
$$(k \cdot a) \cdot b \mapsto a.$$

DEFINITION 5.2. Let $a$ and $b$ be data. An $s_{a,b}$-algorithm is an algorithm that, when applied to input $c$, halts with output $d$ if and only if $(a \cdot c) \cdot (b \cdot c) \mapsto d$.

Let $a$ be a datum. An $s_a$-algorithm is an algorithm that, when applied to input $b$, halts with output an $s_{a,b}$-algorithm.

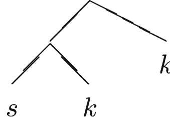
Let $s$ be an algorithm that, when applied to input $a$, halts with output an $s_a$-algorithm.

In summary, consider the trees $\big((s\cdot a)\cdot b\big)\cdot c$ and $(a\cdot c)\cdot(b\cdot c)$:




These trees have the property that if one tree evaluates, then both do; and if they evaluate, then they evaluate to the same datum.

DEFINITION 5.3. The evaluation tree $I$ is defined to be $(s\cdot k)\cdot k$:




PROPOSITION 5.4. *The statement $I\cdot c \mapsto c$ holds for all data $c$.*

*Proof.* The evaluation tree $I\cdot c$ is $\big((s\cdot k)\cdot k\big)\cdot c$. This tree evaluates if and only if $(k\cdot c)\cdot(k\cdot c)$ evaluates, and if they evaluate, they evaluate to the same datum. By definition of $k$, the tree $k\cdot c$ always evaluates to some $b$ that is a $c$-constant algorithm. By the definition of $k$ again, $(k\cdot c)\cdot b \mapsto c$. Thus $I\cdot c \mapsto c$. □

## 6. Structural Equivalence between Evaluation Trees

DEFINITION 6.1. The relation $\sim$ between evaluation trees is defined to be the equivalence relation generated by the following rules:

(a) $(k\cdot T)\cdot c \sim T$.
(b) $\big((s\cdot T_1)\cdot T_2\big)\cdot T_3 \sim (T_1\cdot T_3)\cdot(T_2\cdot T_3)$.
(c) If $T_1 \sim T_2$ then $T\cdot T_1 \sim T\cdot T_2$.
(d) If $T_1 \sim T_2$ then $T_1\cdot T \sim T_2\cdot T$.
(e) If $T\mapsto c$ then $T \sim \mathrm{Tree}(c)$.

In the above rules, $T, T_1, T_2$, and $T_3$ are evaluation trees, and $c$ is a datum.

PROPOSITION 6.2. *Suppose that $T_1 \sim T_2$, where $T_1$ and $T_2$ are evaluation trees. Then $T_1$ evaluates if and only if $T_2$ evaluates; and if they evaluate, they evaluate to the same datum.*

*Proof.* This follows directly from the definitions of Sections 4 and 5. □

PROPOSITION 6.3. *The equivalence* $I \cdot c \sim \mathrm{Tree}(c)$ *holds for all data* $c$.

*Proof.* This follows from Proposition 5.4 and Definition 6.1(e). □

## 7. A Formal Language for Algorithmic Logic

The language of pure combinatory logic is strikingly simple, yet expressive enough to formulate an internal abstraction principle. As mentioned above, we do not need formulas in our language, so we use only the terms of pure combinatory logic.

DEFINITION 7.1. A *pure CL-term*[9], or *CL-term* for short, is recursively defined to be an expression of either of the following types.

(a) An *atom* which is any of a countable number of variable symbols or the constant symbols $K$ or $S$.
(b) An *application term* which is a term of the form $(\tau_1\tau_2)$ where $\tau_1$ and $\tau_2$ are pure CL-terms.

Since there are no quantifiers in this language, all variables are free.

DEFINITION 7.2. A *valuation* $\sigma$ is an assignment of data to each variable (or at least to each variable occurring in the CL-terms under consideration). Given a CL-term $\tau$ the associated evaluation tree $|\tau|_\sigma$ is defined as follows.

(a) $|K|_\sigma$ is $\mathrm{Tree}(k)$ and $|S|_\sigma$ is $\mathrm{Tree}(s)$.
(b) If $x$ is a variable, then $|x|_\sigma$ is $\mathrm{Tree}(c)$ where $c$ is the datum assigned to $x$ by $\sigma$.
(c) For CL-terms $\tau_1$ and $\tau_2$, $|(\tau_1\tau_2)|_\sigma$ is $|\tau_1|_\sigma \cdot |\tau_2|_\sigma$.

DEFINITION 7.3. Let $\sigma$ be a valuation, and $\tau$ a CL-term. If the tree $|\tau|_\sigma$ evaluates to $c$, then $\|\tau\|_\sigma$ is defined to be the datum $c$:

$$|\tau|_\sigma \mapsto \|\tau\|_\sigma.$$

If the tree $|\tau|_\sigma$ does not evaluate, then $\|\tau\|_\sigma$ is undefined.

DEFINITION 7.4. Let $I$ be the CL-term $((SK)K)$. (Context will distinguish between the *term* $I$ and the *tree* $I$.)

The following is a direct consequence of the above definitions.

LEMMA 7.5. *The equation* $|I|_\sigma = I$ *holds for all valuations* $\sigma$. (*On the left hand side I is a CL-term, and on the right hand side I is a tree.*)

## 8. The Lambda Operator

There are several ways to define the lambda operator, each with its own advantages.[10] We choose the simplest for our purposes.

DEFINITION 8.1. Let $x$ be a variable. The *lambda operator* $\lambda_x$ associated with $x$ is a map from the set of CL-terms to the set of CL-terms. The definition is recursive:

(a) $\lambda_x x$ is $I$.
(b) If $\tau$ is an atom not equal to $x$, then $\lambda_x \tau$ is $(K\tau)$.
(c) $\lambda_x(\tau_1\tau_2)$ is $\big((S\,\lambda_x\tau_1)\lambda_x\tau_2\big)$.

The lambda operator is also called the *abstraction operator*.

PROPOSITION 8.2. *Let* $\tau$ *be a CL-term. The variable* $x$ *is not a variable of* $\lambda_x\tau$. *If* $y$ *is a variable distinct from* $x$, *then* $y$ *is a variable of* $\lambda_x\tau$ *if and only if it is a variable of* $\tau$.

*Proof.* This follows directly from the definition. □

PROPOSITION 8.3. *Let* $\tau$ *be a CL-term and* $\sigma$ *an valuation. Then the tree* $|\lambda_x\tau|_\sigma$ *evaluates. In other words,* $\|\lambda_x\tau\|_\sigma$ *is defined.*

*Proof.* We prove this by induction on the length of $\tau$.

Case 1: $\tau$ *is* $x$. By Definition 8.1(a), $\lambda_x\tau$ is the term $I$. By Lemma 7.5, $|\lambda_x\tau|_\sigma$ is the tree $I$ which is just $(s\cdot k)\cdot k$. By definition of $s$, this tree evaluates to an $s_{k,k}$-algorithm.

Case 2: $\tau$ *is an atom not equal to* $x$. By Definition 8.1(b), $|\lambda_x\tau|_\sigma$ is $|(K\tau)|_\sigma$ which, by Definition 7.2, is $k\cdot|\tau|_\sigma$. In this case $|\tau|_\sigma$ is of the form $\mathrm{Tree}(c)$ for some datum $c$. Thus $|\lambda_x\tau|_\sigma$ is $k\cdot c$ (using Convention 4.3). By the definition of $k$, the tree $k\cdot c$ evaluates to a $c$-constant algorithm.

Case 3: $\tau$ *is* $(\tau_1\tau_2)$ *where* $\tau_1$ *and* $\tau_2$ *are two CL-terms for which the proposition holds.* By assumption, $\|\lambda_x\tau_i\|_\sigma$ is defined and equal to some $a_i$. So, with $=$ referring to equality of trees treated as data,

$$\begin{aligned} |\lambda_x\tau|_\sigma &= |\lambda_x(\tau_1\tau_2)|_\sigma \\ &= \big|\big((S\,\lambda_x\tau_1)\lambda_x\tau_2\big)\big|_\sigma && \text{by Definition 8.1(c)} \\ &= \big(s\cdot|\lambda_x\tau_1|_\sigma\big)\cdot|\lambda_x\tau_2|_\sigma && \text{by Definition 7.2.} \end{aligned}$$

By definition of $s$, the tree $\left(s \cdot |\lambda_x \tau_1|_\sigma\right) \cdot |\lambda_x \tau_2|_\sigma$ evaluates to an $s_{a_1,a_2}$-algorithm. □

## 9. The Internal Abstraction Principle

Now that we have the appropriate definitions and conventions, the internal abstraction principle follows straightforwardly.

THEOREM 9.1. (*Internal Abstraction Principle) For all CL-terms* $\tau$, *variables* $x$, *and valuations* $\sigma$,

$$|((\lambda_x \tau)x)|_\sigma \sim |\tau|_\sigma.$$

*Proof.* The proof is by induction on the length of $\tau$. Throughout suppose that the valuation $\sigma$ assigns the datum $c$ to the variable $x$. In other words, that $|x|_\sigma$ is Tree(c) and $\|x\|_\sigma$ is $c$. Below, the symbol $=$ represents equality of data.

Case 1: *$\tau$ is $x$*. By Definition 8.1(a), $\lambda_x \tau$ is the term $I$. By Lemma 7.5, $|\lambda_x \tau|_\sigma$ is the tree $I$. So

$$\begin{aligned} |((\lambda_x \tau)x)|_\sigma &= |\lambda_x \tau|_\sigma \cdot |x|_\sigma && \text{by Definition 7.2(c)} \\ &= I \cdot \mathrm{Tree}(c) \\ &= I \cdot c && \text{using Convention 4.3.} \end{aligned}$$

But $I \cdot c \sim \mathrm{Tree}(c)$ by Proposition 6.3, and $\mathrm{Tree}(c)$ is $|\tau|_\sigma$ in this case.

Case 2: *$\tau$ is an atom not equal to $x$*. So $\lambda_x \tau$ is $(K\tau)$ by Definition 8.1(b).

$$\begin{aligned} |((\lambda_x \tau)x)|_\sigma &= |((K\tau)x)|_\sigma \\ &= \left(k \cdot |\tau|_\sigma\right) \cdot |x|_\sigma && \text{by Definition 7.2} \\ &= \left(k \cdot |\tau|_\sigma\right) \cdot \mathrm{Tree(c)} \\ &= \left(k \cdot |\tau|_\sigma\right) \cdot c && \text{using Convention 4.3.} \end{aligned}$$

However, $\left(k \cdot |\tau|_\sigma\right) \cdot c \sim |\tau|_\sigma$ by Definition 6.1(a).

Case 3: *$\tau$ is $(\tau_1 \tau_2)$ where $\tau_1$ and $\tau_2$ are two CL-terms for which the proposition holds*. Then

$$\begin{aligned} |((\lambda_x \tau)x)|_\sigma &= \left|\left(((S\, \lambda_x \tau_1)\lambda_x \tau_2)x\right)\right|_\sigma && \text{by Definition 8.1(c)} \\ &= \left(\left(s \cdot |\lambda_x \tau_1|_\sigma\right) \cdot |\lambda_x \tau_2|_\sigma\right) \cdot |x|_\sigma && \text{by Definition 7.2.} \end{aligned}$$

But, by Definition 6.1(b),

$$\Big(\big(s\cdot|\lambda_x\tau_1|_\sigma\big)\cdot|\lambda_x\tau_2|_\sigma\Big)\cdot|x|_\sigma\sim\big(|\lambda_x\tau_1|_\sigma\cdot|x|_\sigma\big)\cdot\big(|\lambda_x\tau_2|_\sigma\cdot|x|_\sigma\big).$$

Now, by Definition 7.2(c),

$$|\lambda_x\tau_1|_\sigma\cdot|x|_\sigma=|((\lambda_x\tau_1)x)|_\sigma \quad\text{and}\quad |\lambda_x\tau_2|_\sigma\cdot|x|_\sigma=|((\lambda_x\tau_2)x)|_\sigma,$$

and, by the inductive hypothesis,

$$|((\lambda_x\tau_1)x)|_\sigma\sim|\tau_1|_\sigma \quad\text{and}\quad |((\lambda_x\tau_2)x)|_\sigma\sim|\tau_2|_\sigma.$$

Therefore,

$$\begin{aligned}|((\lambda_x\tau)x)|_\sigma &\sim |((\lambda_x\tau_1)x)|_\sigma\cdot|((\lambda_x\tau_2)x)|_\sigma\\ &\sim |\tau_1|_\sigma\cdot|((\lambda_x\tau_2)x)|_\sigma && \text{by Definition 6.1(d)}\\ &\sim |\tau_1|_\sigma\cdot|\tau_2|_\sigma && \text{by Definition 6.1(c).}\end{aligned}$$

By Definition 7.2(c), $|\tau_1|_\sigma\cdot|\tau_2|_\sigma$ is just $|(\tau_1\tau_2)|_\sigma$, and, in the current case, $|(\tau_1\tau_2)|_\sigma$ is $|\tau|_\sigma$. □

In order to justify the claim that the above theorem is an *internal* abstraction principle, we need to verify that statements of the form $T_1\sim T_2$ are algorithmic statements, where $T_1$ and $T_2$ are evaluation trees.

LEMMA 9.2. *There is an algorithm* TEQUIV *which, when applied to an input of the form* $[T_1, T_2]$ *with* $T_1$ *and* $T_2$ *evaluation trees, halts with output* 1 *if* $T_1\sim T_2$*, and does not halt otherwise. In other words, for trees* $T_1, T_2$*, the algorithmic statement* $[\text{TEQUIV}, [T_1, T_2], 1]$ *is true if and only if* $T_1\sim T_2$.

*Proof.* For any positive integer $n$, define $\sim_n$ to be the weakest equivalence relation among trees with data size less than $n$ such that

(a) $(k\cdot T)\cdot c\sim_n T$.
(b) $((s\cdot T_1)\cdot T_2)\cdot T_3\sim_n(T_1\cdot T_3)\cdot(T_2\cdot T_1)$.
(c) If $T_1\sim_n T_2$ then $T\cdot T_1\sim_n T\cdot T_2$.
(d) If $T_1\sim_n T_2$ then $T_1\cdot T\sim_n T_2\cdot T$.
(e) If $T\mapsto c$ with a process of runtime less than $n$, then $T\sim_n \mathrm{Tree}(c)$,

where all trees in the above rules are required to have data size less than $n$.

The first step is to observe that $T_1 \sim_n T_2$ is effectively decidable for all trees $T_1$ and $T_2$ of data size less than $n$. The second step is to observe that $T_1 \sim T_2$ holds if and only if $T_1 \sim_n T_2$ holds for some $n$.[11] □

So $T_1 \sim T_2$ can be regarded as the algorithmic statement $[\text{TEQUIV}, [T_1, T_2], 1]$.

THEOREM 9.3. (*Simplified Internal Abstraction Principle*) *Let* $\tau$ *be a CL-term*, $x$ *a variable*, *and* $\sigma$ *a valuation. Then*

$$|((\lambda_x \tau)x)|_\sigma \sim \|\lambda_x \tau\|_\sigma \cdot \|x\|_\sigma,$$

*so*

$$\|\lambda_x \tau\|_\sigma \cdot \|x\|_\sigma \sim |\tau|_\sigma.$$

*Proof.* First note that $\|\lambda_x \tau\|_\sigma$ is defined by Proposition 8.3. In other words,

$$|\lambda_x \tau|_\sigma \mapsto \|\lambda_x \tau\|_\sigma.$$

So, by Definition 6.1(e),

$$|\lambda_x \tau|_\sigma \sim \text{Tree}\Big(\|\lambda_x \tau\|_\sigma\Big).$$

By Definition 6.1(d) (and Convention 4.3),

$$|\lambda_x \tau|_\sigma \cdot \|x\|_\sigma \sim \|\lambda_x \tau\|_\sigma \cdot \|x\|_\sigma.$$

By Definition 7.2(c) (and Convention 4.3 again),

$$|((\lambda_x \tau)x)|_\sigma = |\lambda_x \tau|_\sigma \cdot |x|_\sigma = |\lambda_x \tau|_\sigma \cdot \text{Tree}\Big(\|x\|_\sigma\Big) = |\lambda_x \tau|_\sigma \cdot \|x\|_\sigma.$$

Therefore, $|((\lambda_x \tau)x)|_\sigma \sim \|\lambda_x \tau\|_\sigma \cdot \|x\|_\sigma$. The full statement now follows from the Internal Abstraction Principle (Theorem 9.1). □

The following are weaker internal abstraction principles in the sense that they only yield algorithmic statements in the case where $\|\tau\|_\sigma$ is defined.

COROLLARY 9.4. *Let* $\tau$ *be a CL-term*, $x$ *a variable*, *and* $\sigma$ *a valuation such that* $\|\tau\|_\sigma$ *is defined. Then*

$$\|\lambda_x \tau\|_\sigma \cdot \|x\|_\sigma \mapsto \|\tau\|_\sigma \quad \text{and} \quad \big[\|\lambda_x \tau\|_\sigma, \|x\|_\sigma, \|\tau\|_\sigma\big].$$

*are true algorithmic statements.*

*Proof.* First observe that the algorithmic statements in question are indeed well-defined since $\|\lambda_x \tau\|_\sigma$ is defined by Proposition 8.3, $\|x\|_\sigma$ is defined for all valuations $\sigma$, and $\|\tau\|_\sigma$ is defined by assumption.

By the Simplified Internal Abstraction Principle (Theorem 9.3),

$$\|\lambda_x \tau\|_\sigma \bullet \|x\|_\sigma \sim |\tau|_\sigma.$$

By assumption, $|\tau|_\sigma \mapsto \|\tau\|_\sigma$. So, by Proposition 6.2,

$$\|\lambda_x \tau\|_\sigma \bullet \|x\|_\sigma \mapsto \|\tau\|_\sigma.$$

Finally, by the definition of evaluation (Definition 4.4), the algorithmic statement $\left[\|\lambda_x \tau\|_\sigma, \|x\|_\sigma, \|\tau\|_\sigma\right]$ holds. □

## 10. The Evaluation Rules

In order to make use of the internal abstraction principle, the inference library $\rho$ needs valid rules concerning the evaluation of trees. We define three such rules using the conventions for rules and rules diagrams from [2]. These three rules together will be called the *Evaluation Rules*.

RULE 1. The *Tree Equivalence Rule* is an algorithm that implements the following rule diagram:

$$\frac{\begin{array}{c} T_1 \sim T_2 \\ T_1 \mapsto a \end{array}}{T_2 \mapsto a.}$$

PROPOSITION 10.1. *The Tree Equivalence Rule is $\rho$-valid for all libraries $\rho$.*

*Proof.* This follows from Proposition 6.2 (see also Definition 4.5). □

RULE 2. The *Tree Translation Rule* is an algorithm that simultaneously implements the following two diagrams:

$$\frac{[a, b, c]}{a \bullet b \mapsto c} \qquad \frac{a \bullet b \mapsto c}{[a, b, c]} \quad .$$

PROPOSITION 10.2. *The Tree Translation Rule is $\rho$-valid for all libraries $\rho$.*

*Proof.* Recall that $[a, b, c]$ is true if and only if $a$ applied to $b$ yields $c$. So, by the definition of evaluation (Definition 4.4), $[a, b, c]$ is true if and only if $a{\cdot}b \mapsto c$ is true. □

A brute-force method of proving results about algorithms is to emulate the algorithm and see what happens. In particular, if a tree $T$ evaluates, one can, by brute-force emulation if necessary, prove that $T$ evaluates. This technique is implemented in the following:

RULE 3. The *Direct Evaluation Rule* is an algorithm generating the sentence $T{\mapsto}c$ whenever $T$ is a tree that evaluates to $c$.

As with all rules (as defined in [2]), it expects an input of the form $[H, \rho, m]$ where $H$ is a list of algorithmic statements, $\rho$ is expected (but not required) to be a library of rules, and $m$ is a natural number. The Direct Evaluation Rule ignores $\rho$ and all the statements on $H$, but uses $m$ to bound its calculations. Its main task is to systematically searches for trees that evaluate. A full search would take an infinite number of steps (or, in the terminology of [2], an infinite *runtime*), but rules are required to halt. So the Direct Evaluation Rule stops its search after runtime $m$. Then, for every evaluating tree $T$ that was detected, it adds the statement $T{\mapsto}c$ to $H$ where $c$ is the value of the evaluation of $T$. Finally it outputs this extended list $H$.

Since trees that evaluate form a recursively enumerable set, we conclude that for all true statements of the form $T{\mapsto}c$ there will be an $m$ such that $T{\mapsto}c$ will be added to the output list.

PROPOSITION 10.3. *The Direct Evaluation Rule is $\rho$-valid for all libraries $\rho$. Suppose $\rho$ contains this rule, and suppose that $T$ is a tree that evaluates to $c$. Then*

$$\vdash_{\rho} T{\mapsto}c.$$

*Proof.* The rule only appends true statements to the input list, so it is $\rho$-valid. If $\rho$ contains the rule, then the deduction algorithm (DEDUCE, described in [2]) will eventually invoke the rule with a runtime $m$ sufficient to generate $T{\mapsto}c$. □

*Remark.* The Direct Evaluation Rule is similar to the Universal Rule of [2]. In fact, the following shows that if an algorithmic statement is true, then the Evaluation Rules will prove it true.

PROPOSITION 10.4. *If $\rho$ contains the Evaluation Rules and if $A$ is a true algorithmic statement, then $\vdash_{\rho} A$.*

*Proof.* Write $A = [a, b, c]$. Since $A$ is true, $a \cdot b \mapsto c$. By Proposition 10.3, $\vdash_{\rho} a \cdot b \mapsto c$. Since $\rho$ contains the Tree Translation Rule, $\vdash_{\rho} [a, b, c]$. □

This proposition allows us to strengthen the Internal Abstraction Principle whenever $\rho$ contains the three Evaluation Rules. In this case the Internal Abstraction Principle is not merely true, but it is $\rho$-provable.

THEOREM 10.5. *Let $\tau$ be a CL-term, $x$ a variable, and $\sigma$ a valuation. If the library $\rho$ contains the Evaluation Rules then*

$$\vdash_{\rho} \left|\left((\lambda_x \tau)x\right)\right|_{\sigma} \sim |\tau|_{\sigma}.$$

*Proof.* The result follows by Proposition 10.4 and the Internal Abstraction Principle (Theorem 9.1). □

## 11. Applications to Fixed Points

### 11.1. Definition of the Curry Algorithm

A *Curry fixed point* with respect to a statement $A$ is a statement $Q_\rho$ such that $Q_\rho \stackrel{\rho}{\Longleftrightarrow} (Q_\rho \stackrel{\rho}{\Rightarrow} A)$ is true: it is a logical fixed point of the transformation that sends $B$ to $B \stackrel{\rho}{\Rightarrow} A$. In classical or intuitionistic logic such a fixed point would lead to a contradiction if $A$ is false; in algorithmic logic, however, such fixed points do not necessarily lead to contradictions.

We consider the case where $A$ is $\mathcal{F}$, the canonical false sentence in algorithmic logic. Since $\stackrel{\rho}{\neg} Q_\rho$ is defined to be $Q_\rho \stackrel{\rho}{\Rightarrow} \mathcal{F}$, a Curry fixed point for $\mathcal{F}$ has the property that $Q_\rho \stackrel{\rho}{\Longleftrightarrow} \stackrel{\rho}{\neg} Q_\rho$.

In [2], the algorithm CURRY was used to define a Curry fixed point which was then used in several of the theorems in Section 14 of [2]. We now show that CURRY can be defined using the lambda operator.

The definition of CURRY uses algorithmic statements of the form $\left[\alpha, [\alpha, \rho], 1\right]$, which, for convenience, we write as $\Sigma_{\alpha,\rho}$. Let $\mu$ be an algorithm that, when applied to the input $[\alpha, \rho]$, outputs $\left[[\Sigma_{\alpha,\rho}], \rho, \mathcal{F}\right]$. In other words,

$$\mu \cdot [\alpha, \rho] \mapsto \left[[\Sigma_{\alpha,\rho}], \rho, \mathcal{F}\right].$$

Here $\rho$ is expected, but not required, to be a library and $\alpha$ is expected, but not required, to be an algorithm.

Informally, CURRY can be described as an algorithm that, when applied to input $[\alpha, \rho]$ with $\alpha$ an algorithm and $\rho$ a library, attempts to disprove the algorithmic statement $\Sigma_{\alpha,\rho} = \left[\alpha, [\alpha, \rho], 1\right]$ using the library $\rho$. More specifically, when applied to $[\alpha, \rho]$, the algorithm CURRY runs the process DEDUCE applied to $\left[[\Sigma_{\alpha,\rho}], \rho, \mathcal{F}\right]$, and the output of this process, if any, becomes the output of CURRY.[12] So $\left[\text{CURRY}, [\alpha, \rho], 1\right]$ asserts that $\left[\alpha, [\alpha, \rho], 1\right]$ is false.

Using the lambda operator, and $\mu$, we can express the definition of CURRY more succinctly as $\lambda_x(\text{DEDUCE}(\mu x))$. Of course, this is not a pure CL-term since DEDUCE and $\mu$ are not constants of the formal language. We get around this expressive limitiation by using a suitable valuation $\sigma$.

DEFINITION 11.1. Define CURRY to be $\|\lambda_x(z(yx))\|_\sigma$ where $x$, $y$, and $z$ are distinct variables, and where $\sigma$ is a valuation assigning the algorithm $\mu$ to $y$, and the algorithm DEDUCE to $z$. Observe that $\|\lambda_x(z(yx))\|_\sigma$ is well-defined by Proposition 8.3.

LEMMA 11.2. *For all data $\alpha$ and $\rho$,*

$$\text{CURRY}^\bullet[\alpha, \rho] \sim \text{DEDUCE}^\bullet\left[[\Sigma_{\alpha,\rho}], \rho, \mathcal{F}\right].$$

*Proof.* Let $\sigma$ be a valuation which assigns the datum $[\alpha, \rho]$ to $x$, the algorithm $\mu$ to $y$, and the algorithm DEDUCE to $z$.

By the Simplified Internal Abstraction Principle (Theorem 9.3),

$$\|\lambda_x(z(yx))\|_\sigma{}^\bullet\|x\|_\sigma \sim |(z(yx))|_\sigma.$$

In other words,

$$\text{CURRY}^\bullet[\alpha, \rho] \sim |(z(yx))|_\sigma.$$

By Definition 7.2 and Convention 4.3,

$$|(z(yx))|_\sigma = |z|_\sigma{}^\bullet|(yx)|_\sigma = |z|_\sigma{}^\bullet\left(|y|_\sigma{}^\bullet|x|_\sigma\right) = \text{DEDUCE}^\bullet\left(\mu^\bullet[a, \rho]\right).$$

So

$$\text{CURRY}^\bullet[\alpha, \rho] \sim \text{DEDUCE}^\bullet(\mu^\bullet[\alpha, \rho]).$$

Since $\mu^{\bullet}[\alpha, \rho] \mapsto \big[[\Sigma_{\alpha,\rho}], \rho, \mathcal{F}\big]$, Definition 6.1(e) gives

$$\mu^{\bullet}[\alpha, \rho] \sim \text{Tree}\Big(\big[[\Sigma_{\alpha,\rho}], \rho, \mathcal{F}\big]\Big).$$

Finally, Definition 6.1(c) and Convention 4.3 give

$$\text{DEDUCE}^{\bullet}\big(\mu^{\bullet}[\alpha, \rho]\big) \sim \text{DEDUCE}^{\bullet}\big[[\Sigma_{\alpha,\rho}], \rho, \mathcal{F}\big].$$

□

THEOREM 11.3. Let $\rho$ be a library containing the Evaluation Rules. Then, for any datum $\alpha$,

$$\big[\text{CURRY}, [\alpha, \rho], 1\big] \overset{\rho}{\Longleftrightarrow} \overset{\rho}{\dashv}\big[\alpha, [\alpha, \rho], 1\big].$$

*Proof.* Let $W$ be the $\rho$-deductive closure of $\big[\text{CURRY}, [\alpha, \rho], 1\big]$ (in the sense of [2]). By the Tree Translation Rule, $\text{CURRY}^{\bullet}[\alpha, \rho] \mapsto 1$ is in $W$. By the above lemma and Proposition 10.4,

$$\text{CURRY}^{\bullet}[\alpha, \rho] \sim \text{DEDUCE}^{\bullet}\big[[\Sigma_{\alpha,\rho}], \rho, \mathcal{F}\big].$$

is in $W$. So by the Tree Equivalence Rule,

$$\text{DEDUCE}^{\bullet}\big[[\Sigma_{\alpha,\rho}], \rho, \mathcal{F}\big] \mapsto 1$$

is in $W$. By the Tree Translation Rule,

$$\Big[\text{DEDUCE}, \big[[\Sigma_{\alpha,\rho}], \rho, \mathcal{F}\big], 1\Big]$$

is in $W$. Now, by the definitions and notation of [2], the statement $\overset{\rho}{\dashv}\Sigma_{\alpha,\rho}$ is exactly $\Big[\text{DEDUCE}, \big[[\Sigma_{\alpha,\rho}], \rho, \mathcal{F}\big], 1\Big]$. Thus $\overset{\rho}{\dashv}\Sigma_{\alpha,\rho}$ is in $W$. Therefore,

$$\big[\text{CURRY}, [\alpha, \rho], 1\big] \vdash_{\rho} \overset{\rho}{\dashv}\Sigma_{\alpha,\rho}.$$

A similar argument gives

$$\overset{\rho}{\dashv}\Sigma_{\alpha,\rho} \vdash_{\rho} \big[\text{CURRY}, [\alpha, \rho], 1\big].$$

□

*Remark.* The argument of paper [1] was based on the presence in the inference library of a valid but ad hoc rule describing the behavior of CURRY. The above theorem shows that if the library $\rho$ contains the (non-ad hoc) Evaluation Rules, then the argument of [1] applies without an ad hoc rule in $\rho$.

## 11.2. Definition of the curry Fixed Point

DEFINITION 11.4. Let $\rho$ be a library. Then $Q_\rho$ is defined to be the algorithmic statement $\left[\text{CURRY}, \left[\text{CURRY}\ \rho\right], 1\right]$.

THEOREM 11.5. *If $\rho$ is a library containing the Evaluation Rules, then*

$$Q_\rho \overset{\rho}{\Longleftrightarrow} \overset{\rho}{\neg} Q_\rho.$$

*Proof.* By Theorem 11.3,

$$\left[\text{CURRY}, \left[\text{CURRY}, \rho\right], 1\right] \overset{\rho}{\Longleftrightarrow} \overset{\rho}{\neg} \left[\text{CURRY}, \left[\text{CURRY}, \rho\right], 1\right].$$ □

## 12. Consequence of the Curry Fixed Point

Several of the results of Chapter 14 of [2] can now be strengthened for libraries $\rho$ containing the Evaluation Rules.

THEOREM 12.1. *Let $\rho$ be a valid library containing the Evaluation Rules. Then $Q_\rho, \overset{\rho}{\neg} Q_\rho \vdash_\rho \mathcal{F}$ is false. Thus the law $A, \overset{\rho}{\neg} A \vdash_\rho \mathcal{F}$ fails.*

*Proof.* Assume otherwise. So, using the terminology of [2], the statement $\mathcal{F}$ is in any $\rho$-deductively closed set containing both $Q_\rho$ and $\overset{\rho}{\neg} Q_\rho$. By Theorem 11.5, $Q_\rho \overset{\rho}{\Longleftrightarrow} \overset{\rho}{\neg} Q_\rho$. In other words, the statement $Q_\rho$ is in a $\rho$-deductively closed set if and only if $\overset{\rho}{\neg} Q_\rho$ is. Also, by the validity of $\rho$, the statement $Q_\rho$ holds if and only if its negation $\overset{\rho}{\neg} Q_\rho$ holds.

Let $W$ be the $\rho$-deductive closure of $Q_\rho$. Then, as mentioned above, $\overset{\rho}{\neg} Q_\rho$ is in $W$, so $W$ must contain $\mathcal{F}$. This shows $Q_\rho \overset{\rho}{\Rightarrow} \mathcal{F}$ holds; that is, $\overset{\rho}{\neg} Q_\rho$ is true. As mentioned above, this implies that $Q_\rho$ is true. Since $Q_\rho, \overset{\rho}{\neg} Q_\rho \vdash_\rho \mathcal{F}$ and since $\rho$ is valid, $\mathcal{F}$ is true. □

COROLLARY 12.2. *Let $\rho$ be a valid library containing the Evaluation Rules. Then the following law fails:*

$$\Gamma \vdash_\rho A \overset{\rho}{\Rightarrow} B \text{ implies } \Gamma, A \vdash_\rho B$$

*Proof.* Consider the case where $A$ is $Q_\rho$ and $B$ is $\mathcal{F}$. So $\overset{\rho}{\neg} Q_\rho \vdash_\rho Q_\rho \overset{\rho}{\Rightarrow} \mathcal{F}$ since $\overset{\rho}{\neg} Q_\rho \vdash_\rho \overset{\rho}{\neg} Q_\rho$. However, $\overset{\rho}{\neg} Q_\rho, Q_\rho \vdash_\rho \mathcal{F}$ is false by Theorem 12.1. □

COROLLARY 12.3. *Let $\rho$ be a valid library containing the Evaluation Rules. Then $Q_\rho \overset{\rho}{\Rightarrow} \mathcal{F}, Q_\rho \vdash_\rho \mathcal{F}$ is false. In particular, the modus ponens law $A \overset{\rho}{\Rightarrow} B, A \vdash_\rho B$ fails.*

*Proof.* As above, $\overset{\rho}{\neg} Q_\rho, Q_\rho \vdash_\rho \mathcal{F}$ is false, and $\overset{\rho}{\neg} Q_\rho$ is just $Q_\rho \overset{\rho}{\Rightarrow} \mathcal{F}$. □

*Remark.* The above shows that any valid library containing the Evaluation Rules cannot contain either of the following rules

$$\frac{A \overset{\rho}{\Rightarrow} B \qquad A}{B} \qquad\qquad \frac{A \qquad \overset{\rho}{\neg} A}{\mathcal{F}}.$$

Recall from [2] that each of these rules is in fact valid if $\rho$ is a valid library (the second is a special case of the first).

## ACKNOWLEDGEMENTS

The authors would like to thank the referee for helpful suggestions.

## NOTES

[1] Some have argued that such restrictions on rules of classical logic undermines the usefulness of type-free logics. We believe, to the contrary, that the expressive power of a suitable type-free logic, much of which comes from the abstraction principle, will compensate for any loss of classical rules of logic, and that such classical rules will be demonstratively valid in most situations where one needs them.

It has been clear since the era of Church and Curry that any type-free logic will require some limitations. What is not so clear is exactly what limitations to make. Different type-free systems have been proposed, each with different limitations. We view the role of algorithmic logic as a guide to help in sorting which limitations to classical logical rules are necessary, and which are not, and to suggest how to work around the limitations that emerge.

[2] In [2] we discuss specific constraints on the selection of a background theory of computation. We use the same assumptions throughout the present paper.

[3] The data set is assumed to be countable. In fact, one can follow a standard practice and take the data set to be $\mathbb{N}$. One then codes all data as natural numbers, including algorithms, lists of natural numbers, strings, etc. For example, one can fix an effective enumeration of all algorithms, and use $e \in \mathbb{N}$ to code for the $e$th algorithm under this enumeration. From this point of view an algorithmic statement is a natural number coding a triple $[e, a, b]$ of natural numbers. Such an algorithmic statement is assigned the value “true” if and only if $\{e\}(a) = b$ where, following Kleene, $\{e\}$ denotes the partial function on $\mathbb{N}$ defined by the $e$th algorithm. We do not set up such codings in this paper, but instead use more informal language.

[4] See [2] for details.

[5] From one point of view this might be expected since the set of data forms a partial combinatory algebra (the Kleene model), and all partial combinatory algebras satisfy an abstraction principle involving a suitable $\lambda$-operator (see [3] Chapter VI). In this paper, however, we not only show that the abstraction principle holds, but that it can be expressed *internally*. The main technical innovation to the usual development of the abstraction operator is the use of trees to make the pairing total and to preserve in the semantics the structure of a computation.

[6] This paper completes the development of the propositional logic. The development of quantifiers will appear later.
[7] In this section we will adopt '•' in this capacity. In Section 7, we will adopt a different syntactic convention.
[8] This notation can result in ambiguity if $c$ is itself a tree. The following conventions will help: lower-case roman letters in tree expressions refer to termini, but upper-case letters refer to subtrees. So '$(b{\bullet}U){\bullet}V$' is short for '(Tree(b)•U)•V.' Expressions such as '$|\tau|_\sigma$' occurring in tree expressions refer to subtrees, while expressions such as '$\|\tau\|_\sigma$' refer to termini. To see why care is required, suppose, for instance, that $\gamma$ is the evaluation tree $\bigwedge$ which is $\big[[a], [a]\big]$. Then '$\gamma{\bullet}\gamma$' can, without some clarification, refer to either $\gamma \bigwedge \gamma$ (with $a$, $a$) which is $\Big[[\gamma], [\gamma]\Big] = \Big[\big[[[a], [a]]\big], \big[[[a], [a]]\big]\Big]$ or to the tree with leaves $a$ $a$ $a$ $a$ which is $\Big[\big[[a], [a]\big], \big[[a], [a]\big]\Big]$.
[9] This is essentially Definition 2.1 of Hindley and Seldin [4].
[10] See Remark 2.16 of Hindley and Seldin [4].
[11] We stipulated in [2] that every datum has a *size* in $\mathbb{N}$, and that there is a finite number of data of any given size. We also stipulated that every halting process has a *runtime* in $\mathbb{N}$ representing the number of steps of the computation.
[12] Here DEDUCE is as in [2].

## REFERENCES


1. Aitken, W. and Barrett, J. A.: Computer implication and the Curry paradox, *Journal of Philosophical Logic* 33(6) (2004), 631–637.
2. Aitken, W. and Barrett, J. A.: Stability and paradox in algorithmic logic, *Journal of Philosophical Logic* 36(1) (2007), 61–95.
3. Beeson, Michael J.: *Foundations of Constructive Mathematics*, Springer, Berlin Heidelberg New York, 1985.
4. Hindley, J. R. and Seldin, J. P.: *Introduction to Combinators and λ-calculus*, Cambridge University Press, Cambridge, UK, 1986.



WAYNE AITKEN
*Department of Mathematics,*
*California State University,*
*San Marcos, CA 92096,USA*
*E-mail: waitken@csusm.edu*

JEFFREY A. BARRETT
*Department of Logic and Philosophy of Science,*
*UC Irvine,*
*Irvine, CA 92697,USA*
*E-mail: jabarret@uci.edu*