A FOUNDATION FOR TYPED CONCATENATIVE LANGUAGES

ROBERT KLEFFNER

A Thesis Presented to

THE DEPARTMENT OF COMPUTER & INFORMATION SCIENCE

in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Northeastern University
Boston, Massachusetts

April 2017
ABSTRACT

The design space for concatenative programming languages, also known as stack-based programming languages, is largely under-explored. Most formal investigations of higher-order stack-based languages are done in the functional setting, requiring a cumbersome and unnecessary encoding. This dissertation describes a core programming language that captures the essence of the concatenative paradigm, provides a reduction semantics for this language, and introduces a sound type system. It also presents a sound and complete type inference algorithm. We expect that this model will serve as a starting point for future explorations of the concatenative language design space.
INTRODUCTION

Stack-oriented programming languages form an important niche in the programming community. Forth has a rich history as a low-level systems development language that can, with proper discipline, exhibit high-level readability and expressivity; today it is most popular in the development of embedded devices [16]. Similarly, PostScript is a popular stack-based language for printing documents, but on occasion, it is used as a standalone programming language [17].

The early 2000’s saw a strong interest in higher-order stack-based languages, beginning with von Thun’s invention of Joy [20]. Joy is a combinatory language, similar to the SKI-calculus in that it has no notion of variable binding. The language found a small but active online following, a significant portion of which was interested in the formal foundations of what are often called concatenative languages [4]. The culmination of this development was the creation of the Factor language by Pestov [11], which continues to be maintained, and which has been extended with a wide range of libraries for many software development domains. Factor has a small but dedicated community of developers.

1.1 WHAT IS CONCATENATIVE?

While most people know Forth as a stack-based programming language, Factor uses the label concatenative language. This dissertation also uses the term concatenative languages.

The lack of a widely agreed upon definition has been a source of argument within the concatenative programming community and without, and inspired a widely-read blogpost [15] that attempted to contrast concatenative languages with applicative (i.e. standard functional) programming languages. Indeed, many concatenative languages can be seen as a subset of λ-calculus based languages, where every function takes a tuple and returns a tuple. We thus make the tentative and loose assertion that concatenative programming is a subset of functional programming in which:

1. Expressions are built by composing rather than applying functions.

2. All functions map a sequence of arguments to a sequence of results.

Additionally, traditional stack-based languages often feature a set of stack shuffling operators or combinators to move items into the desired position on the stack. Without practice and discipline on behalf of the programmer, this variable-free style can result in highly obfuscated programs. We therefore break from tradition and add variables to our language, which alleviates some of this burden at the expense of complicating the language semantics.
1.2 WHY TYPED?

The majority of popular concatenative languages are untyped. Forth is notorious in this regard: most variants provide basic operations on only one type: a sequence of bits usually the size of a machine register. All other type information is encoded and checked by programmer-defined functions. Factor is dynamically typed in the traditional sense, providing runtime-checked operations on lists, numbers, objects, and other primitive types natively.

Discipline is required to make programs readable when developing in concatenative programming languages; still more discipline is required to make sure that a function is passed arguments of the types it expects, and in the right order. In many cases, we would like to provide ahead-of-time guarantees that our programs do not have type conflicts. Furthermore, the type system should infer the majority of this information so that developers do not have to write it themselves. Languages that remain dynamically-typed can make use of inferred type information at runtime to improve the performance of generated code.

Much of the prior work on type inference for stack-based languages comes in two flavors: an algebraic system called stack effects, and a nested pair encoding based on the standard Hindley-Milner type system.

The research on stack effects has mostly been done in the Forth setting. There are several formal descriptions of stack effect notation and stack effect composition \([12, 14, 19]\), which appear to have been developed independently of the type inference literature. Although these systems can support many standard features studied in type theory \([13, 18, 19]\), including intersection types, parametric polymorphism, and subtyping, they all suffer from the same core problem: the type system cannot express the types of functions that are polymorphic on stacks, an essential idiom in this space. As a result, it is essentially impossible to infer the type of functions which themselves require function values passed on the stack.

One solution for stack effect systems is to have a schema of primitive call operators that specify the input and output arity of the functions they can apply:

\[
call_{1,1} : (a \to z) \to z
\]
\[
call_{2,1} : (a b \to z) \to z
\]
\[
call_{1,2} : (a \to y z) \to y z
\]

eetc...

With this solution, the programmer must redefine higher-order functions at each arity; naturally, they would like to write \textit{map} and \textit{compose} once and be done with it. Another solution is to have one call operator with a type of an infinite intersection of stack effects. Unfortunately, all higher-order functions written with this call operator would also have an infinite intersection of stack effects, and whether there is a way to compute and represent these stack effects finitely is an open question.

Type systems that encode stacks as nested pairs are an alternative to algebraic stack-effect systems. These were first developed in the context of func-
tional languages [3, 10] and have since been used to infer types for some concatenative languages [1].

These languages provide a general call operator, usually as a primitive part of the syntax, which has the following type:

\[
\text{call} : (s, (s \to s')) \to s'
\]

That is, call requires a function on top of the stack in addition to the stack elements s required by said function. It returns exactly those elements s' returned by the called function. This type already allows developers to write many useful higher-order functions, including function composition, partial application, and recursion patterns like map and fold from standard functional languages.

On the downside, these type systems restrict stack variables to appear only at the leftmost position of a function signature. The question arises whether we can lift this restriction, allowing stack variables to appear in the middle of a sequence or allowing more than one per sequence. In theory this is possible because sequence unification is decidable [5]. However, sequence unification is infinitary in the general case.

1.3 WHY BOTHER?

The thesis I defend is that higher-order concatenative languages can be profitably equipped with expressive and sound type systems; the system I develop here also admits a type inference algorithm with principal types. Moreover, I maintain that it is useful to study concatenative languages in their own context, rather than from the viewpoint of an embedding in the \(\lambda\)-calculus. Although there is a well-known correspondence between concatenative languages and familiar functional languages [10], it can be cumbersome and limiting to do research for one language in the setting of another.

Concatenative programming is an often overlooked paradigm that I conjecture has much to offer. However, it is difficult to persuade others of this value when the foundations for these languages are still shaky. The design space for these languages remains under-explored compared to other paradigms, especially in the typed setting. Diggins’ work [1, 2] is a good starting point but mostly informal, and he does not provide a treatment of variables, focusing instead on a purely combinatory language.

In this dissertation I set concatenative languages on firm theoretical ground by introducing a core language called \(\lambda_{c}\), pronounced “lambda compose”. This language attempts to capture the essence of concatenative programming in existing languages. In section 2, I present the syntax and semantics for the core language. In section 3, I detail a first-rank polymorphic type system for the language, and prove it sound. In section 4, I describe a type inference algorithm, and prove soundness and completeness of the algorithm with regard to the type system. In section 5, I point toward a reference implementation of \(\lambda_{c}\) built in Racket. This implementation supports both type inference and evaluation of type-checked programs.
THE LANGUAGE $\lambda_0$

Hidden within the nested-pair representation discussed in Section 1 is a distinction between two kinds of type variables: those that are substituted with a single type, and those that are substituted with a sequence of types. The former are the familiar Hindley-Milner type variables, whereas the latter are referred to as sequence variables. Our model, the language $\lambda_0$ \footnote{The symbol for composition as a subscript emphasizes the importance our language places on composability as a design concept.}, is a concatenative language with a type system that enforces this distinction syntactically.

2.1 SYNTAX

The syntax for $\lambda_0$ is given in Fig. 1. In contrast to languages based on the $\lambda$-calculus, $\lambda_0$ has a sequence structure. Thus we have two levels of syntax: expressions $e$ which are lists of words $w$, a term that comes from Forth. Expressions in $\lambda_0$ are read from left to right, and individual words can be added on to either side of an expression by whitespace-separated juxtaposition. We denote concatenation of two expressions as $e_1 \cdot e_2$. The constant literals are the non-negative integers $n$ and boolean values $b$, but this set of constants can be extended as desired; we focus on these two for simplicity. Similarly, $+$ and $\mathtt{if}$ are included as examples of primitive operators.

One key distinction in the syntax is our separation of variable binding from function abstraction. Function abstractions consist of an arbitrary expression wrapped in curly braces, called blocks; this leaves variable bindings free to occur anywhere in an expression. While this separation between notions of scope and function value is not strictly necessary, it is both useful and familiar to programmers in the concatenative community, and it simplifies our presentation of the semantics and the type system. This distinction can also be found, with different semantics, in the call-by-push-value calculus of Levy \cite{Levy91}.

$$e ::= \overrightarrow{w}$$
$$w ::= n \mid b \mid x \mid + \mid \mathtt{call} \mid \mathtt{fix} \mid \mathtt{if}$$
$$\mid \{e\} \mid \lambda x.e \mid \mathtt{let } x = e \mathtt{ in } e$$
$$n ::= 0,1,2...$$
$$b ::= \mathtt{true} \mid \mathtt{false}$$
$$x ::= \mathtt{variables}...$$

Figure 1: $\lambda_0$ Syntax
Because our expressions are list-like, the only parentheses we write in larger expressions are those around variable bindings and let-expressions. These parentheses limit the scope of the bound variables in the case that the scope does not extend to the end of the expression. We also sometimes denote empty expressions with $\varepsilon$ for extra clarity.

2.2 Semantics

Our notion of evaluation results, presented in Fig. 2, is likewise split among two layers: stacks $s$, which are finite sequences of individual values $v$. Like expressions, stacks are read from left to right, with the rightmost value representing the ‘top’ element of the stack. The model has three types of value: numbers, booleans, and blocks. Blocks are essentially closures, and are related to the notions of ‘suspended computation’ appearing in call-by-push-value [7] and Frank [8].

Our reduction relation models a stack-based semantics as a set of transition rules on machine states $m$. A machine state is a pair composed of a stack $s$ and an expression $e$, which is the remainder of the computation to be performed on $s$. Since expressions are read from left to right, the leftmost word of $e$ is applied to the stack at each step.

The transition rules are presented in Fig. 3. The first three rules are simple: numbers, booleans, and blocks are pushed on top of the stack. The rules for $+$ and $if$ are also intuitive: $+$ consumes two numbers from the top of the stack, and pushes the result of adding them, while $if$ consumes a boolean value and chooses to keep one of the two values below the condition based on whether it is $true$ or $false$. A call expects a block at the top of the stack, and evaluates the expression inside the block before continuing to the rest of the expression that follows it. The instruction $fix$ does something similar, but first wraps its function value in a fix-point block and pushes it to the stack to enable recursion. A value binding $\lambda x.e$ consumes a value from the top of the stack and substitutes it for all occurrences of $x$ in the enclosed expression $e$. Likewise, a let-expression substitutes its argument expression into the body expression.

The semantics is dependent on a technical change to the definition of substitution. Our variables do not necessarily map to a single value or word, but to
expressions of arbitrary length. Hence, we cannot simply replace occurrences of \( x \) with their mapped value. Rather, we concatenate the expression mapped by \( x \) with the rest of the expression surrounding \( x \). This change is forced by the inclusion of let-bound expressions; in a language where let-bound variables are syntactically restricted to map to a single word, capture-avoiding substitution can be adapted from \( \lambda \)-based languages with only minor changes. The resulting language is slightly less convenient to use but admits a similar type system and inference algorithm.

We evaluate an expression \( e \) by creating an initial state comprised of \( e \) and an empty stack. Let \( \rightarrow \) be the reflexive and transitive closure of \( \rightarrow \). We would like \( \rightarrow \) on some machine state \( m \) to result in a state with no further words left to apply to the stack or an infinite sequence of machine states. We say that \( m \) is stuck if none of the transition rules can be applied and the expression part \( e \) is not \( \varepsilon \). This gives our first result.

**Proposition 1.** \( \rightarrow \) can get stuck.

**Proof.** The following initial state reduces to a state where no transition rule applies, but the expression part is not empty:

\[
\langle s \mid n \ e \rangle \rightarrow \langle s \ n \mid e \rangle \\
\langle s \mid b \ e \rangle \rightarrow \langle s \ b \mid e \rangle \\
\langle s \mid \{e_1\} \ e_2 \rangle \rightarrow \langle s \ e_1 \mid e_2 \rangle \\
\langle s \ n_2 \ n_1 \mid + \ e \rangle \rightarrow \langle s \ n_2 + n_1 \mid e \rangle \\
\langle s \ v_2 \ v_1 \ true \mid if \ e \rangle \rightarrow \langle s \ v_1 \mid e \rangle \\
\langle s \ v_2 \ v_1 \ false \mid if \ e \rangle \rightarrow \langle s \ v_2 \mid e \rangle \\
\langle s \ e_1 \mid call \ e_2 \rangle \rightarrow \langle s \ e_1 \cdot e_2 \rangle \\
\langle s \ e_1 \mid fix \ e_2 \rangle \rightarrow \langle s \ e_1 \mid fix \mid e_1 \cdot e_2 \rangle \\
\langle s \ v_1 \mid (\lambda x . e_1) \ e_2 \rangle \rightarrow \langle s \ (v/x) e_1 \mid e_2 \rangle \\
\langle s \mid (let \ x = e_1 in \ e_2) \ e_3 \rangle \rightarrow \langle s \ (e_1/x) e_2 \cdot e_3 \rangle
\]

Figure 3: Machine Transition Rules

The inclusion of variables via value bindings and let expressions is a departure from most theoretical investigations of concatenative languages. However,
including variables in our language eliminates the need for primitive stack shuffle combinators: using value bindings and blocks, we can write our own:

\[
\begin{align*}
\ldots \text{let } \text{Dup} &= \lambda x. x \; \text{in } e \ldots \\
\ldots \text{let } \text{Swap} &= \lambda x. \lambda y. x \; y \; \text{in } e \ldots \\
\ldots \text{let } \text{Pop} &= \lambda x. \; \text{in } e \ldots
\end{align*}
\]

We may also write these terms inline:

\[
\begin{align*}
1 \; 2 \; (\lambda x. \lambda y. x \; y) \; 3 \rightarrow 1 \; (\lambda y. 2 \; y) \; 3 \rightarrow 2 \; 1 \; 3
\end{align*}
\]

Variables are useful for more than defining arbitrary stack shuffle combinators: they also allow programmers to write higher-order functions more succinctly:

\[
\begin{align*}
\ldots \text{let } \text{Compose} &= \lambda f. \lambda g. \{ g \; \text{call} \; f \; \text{call} \} \; \text{in } e \ldots \\
\ldots \text{let } \text{Partial} &= \lambda f. \lambda x. \{ x \; f \; \text{call} \} \; \text{in } e \ldots
\end{align*}
\]

Allowing arbitrary expressions as the arguments of let-expressions also permits developers to write ‘partially applied’ functions. These functions are not partially applied in the usual sense but look similar to the automatically ‘curried’ functions of ML and Haskell. They are often as expressive and useful:

\[
\begin{align*}
\ldots \text{let } \text{Add1} &= 1 + \; \text{in } e \ldots \\
\ldots \text{let } \text{Double} &= \text{Dup} + \; \text{in } e \ldots
\end{align*}
\]
TYPES FOR $\lambda_0$

3.1 TYPE SYNTAX

The type syntax defined in Fig. 4 is a variant of the Hindley-Milner type syntax in which functions may have zero or more inputs and outputs. The set of individual types $\tau$ consists of base types Int and Bool, standard type variables $\alpha$, and function types $f$. Sequences of types $i$ (also denoted $o$ for function outputs) may be optionally terminated with at most one sequence type variable $\overrightarrow{\alpha}$ on the left-hand side. We assume two disjoint infinite sets of type variables for $\alpha$ and $\overrightarrow{\alpha}$. All expressions have a function type $f$ in $\lambda_0$, so our type schemes $\sigma$ need only quantify over function types.

$$
\begin{align*}
\tau & := \text{Int} \mid \text{Bool} \mid \alpha \mid f \\
i, o & := \tau \mid \overrightarrow{\tau} \\
f & := i \rightarrow o \\
\sigma & := \forall \overrightarrow{\alpha}.f
\end{align*}
$$

Figure 4: Types for $\lambda_0$

As for expressions, we append a single type onto either side of a type sequence by whitespace-separated juxtaposition, and denote concatenation of two type sequences by $i_1 \cdot i_2$. Sequence type variables are substituted with sequences of types $i$ rather than individual types $\tau$, but in contrast to the term language, we must syntactically distinguish between individual and sequence type variables to make type inference work. We let $\overrightarrow{\alpha}$ range over both individual and sequence type variables in types schemes.

We write $\text{ftv}(f)$ for the set of both individual and sequence type variables that occur free in function types $f$:

$$
\begin{align*}
\text{ftv}(\text{Int}) = \text{ftv}(\text{Bool}) = \emptyset \\
\text{ftv}(\alpha) = \{\alpha\} \\
\text{ftv}(\overrightarrow{\alpha}) = \{\overrightarrow{\alpha}\} \\
\text{ftv}(i \rightarrow o) = \text{ftv}(i) \cup \text{ftv}(o) \\
\text{ftv}(\tau_1 \ldots \tau_n) = \text{ftv}(\tau_1) \cup \ldots \cup \text{ftv}(\tau_n)
\end{align*}
$$

For type schemes $\sigma := \forall \overrightarrow{\alpha}.f$, we treat $\overrightarrow{\alpha}$ as a set of type variables. The free type variables of $\sigma$ are then defined as

$$
\text{ftv}(\forall \overrightarrow{\alpha}.f) = \text{ftv}(f) - \overrightarrow{\alpha}
$$
In the rest of this text, we do not distinguish between the empty type scheme \(\forall f\) and unquantified types \(f\), choosing to write them both as the latter.

As for the Hindley-Milner type system, our type schemes are related to each other by an instantiation relation. Instantiation is similar to its Hindley-Milner counterpart, except that individual type variables may only be instantiated with individual types \(\tau\), whereas sequence type variables are instantiated with a finite sequence of types \(i\). We denote instantiation by \(\sigma_2 \sqsubseteq \sigma_1\) with \(\sigma_1\) more specific than \(\sigma_2\):

\[
S = [\alpha_i \mapsto \tau_i, ..., \alpha_j \mapsto i_j, ...] \quad \{\alpha_i, ..., \alpha_j, ...\} \subset \vec{\alpha} \quad \sigma_1 = S(\sigma_2)
\]

\(\forall \vec{\alpha}. \sigma_2 \sqsubseteq \sigma_1\)

### 3.2 Type Environments

Type environments are a finite map from term-level variables to type schemes:

\[
\Gamma := x_1 : \sigma_1; ...; x_n : \sigma_n
\]

We write \(\Gamma; x : \sigma\) to specify that a judgment is valid for any environment containing at least \(x : \sigma\). We assume weakening to be valid for our purposes and do not prove it here: the proof is a straightforward adaptation of weakening from \(\lambda\)-based languages.

We can also get the set of type variables free in a type environment:

\[
\text{ftv}(x_1 : \sigma_1, ..., x_n : \sigma_n) = \text{ftv}(\sigma_1) \cup ... \cup \text{ftv}(\sigma_n)
\]

This enables us to define the familiar notion of a generalized function type \(f\) with respect to some type environment \(\Gamma\), which we denote \(\hat{\Gamma}(f)\):

\[
\hat{\Gamma}(f) = \forall \vec{\alpha}. f \quad \text{where} \quad \vec{\alpha} = \text{ftv}(f) - \text{ftv}(\Gamma)
\]

### 3.3 Substitutions

Substitutions \(S\) (sometimes denoted by \(R\) and \(\Phi\)) are finite mappings from individual type variables to individual type \(\tau\), and sequence type variables to a sequence of types. A substitution applied to a type replaces free variables present in the substitution domain with the mapped type or sequence if the variable is present in the substitution. As for substitution in expressions, we must flatten nested sequences using concatenation when replacing for sequence type variables. Since all sequence type variables occur in a sequence of types in \(\lambda_o\), the operation remains closed on types. When applying a substitution to a type scheme, we first remove all variables present in the quantifier from the substitution domain:

\[
S(\forall \vec{\alpha}. f) = \forall \vec{\alpha}. (S - \vec{\alpha})(f)
\]

To apply a substitution to a type environment, we apply the substitution to each type scheme in the range of the environment. Applying a substitution to another substitution, sometimes written \(S \circ S'\), is defined by

\[
S \circ S' = S \cup [\alpha \mapsto S(i), \forall(\alpha, i) \in S' \text{ such that } \alpha \notin \text{Dom}(S)]
\]
The syntax-directed typing rules for \( \lambda_o \) are given in Fig. 5. The rules define the typing judgment \( \Gamma \vdash e : f \), read as "in type environment \( \Gamma \), expression \( e \) has function type \( f \)". Expressions are treated as left-associative, in contrast to the right-associative structure used in the specification of the transition rules. Since our expressions are list-like, we can switch between the representations as necessary. The base case is the empty expression \( \varepsilon \), but the type rule for \( \varepsilon \) is somewhat unintuitive. One might expect empty expressions to consume no input and produce no output, and the stack-effect systems discussed in Section 1.2 take this route. However, that choice makes it difficult to thread stack-polymorphism through the other type rules. Instead, we allow arbitrary choice of the input

\begin{align*}
\text{Empty} & \quad \Gamma \vdash e : i \rightarrow i \\
\text{Num} & \quad \Gamma \vdash e : i \rightarrow o \\
\text{Bool} & \quad \Gamma \vdash e : i \rightarrow o \\
\text{Add} & \quad \Gamma \vdash e : i \rightarrow o \ \text{Int} \quad \Gamma \vdash e + : i \rightarrow o \ \text{Int} \\
\text{If} & \quad \Gamma \vdash e : i \rightarrow o \ \tau \ \tau \quad \Gamma \vdash e \ \text{if} : i \rightarrow o \ \tau \\
\text{Let} & \quad \Gamma \vdash e : f \\
\text{Call} & \quad \Gamma \vdash e : i \rightarrow o_1 \ (i_2 \rightarrow o_2) \\
\text{Stack} & \quad \Gamma \vdash s : o \\
\text{VNum} & \quad \Gamma \vdash \text{v} : \tau \\
\text{VBool} & \quad \Gamma \vdash \text{b} : \text{Bool} \\
\text{VBlock} & \quad \Gamma \vdash \{e\} : f \\
\text{Machine} & \quad \Gamma \vdash s : i \\
\end{align*}

\[ \Gamma \vdash \langle s \mid e \rangle : o \]

Figure 5: Type System for \( \lambda_o \)
and output sequences in the type of $e$, so long as the two sequences are syntactically equal. This fits with our intuition that the empty expression is the identity function on all stacks.

In the recursive rules, we specify that the input consumed by a word must be exactly the output produced by the previous expression, even if the word ‘does not care’ about some of those values and simply passes them through. Hence, all expressions in $\lambda_0$ are polymorphic with regard to the part of the stack they ‘do not care’ about. This is a significant difference from the stack effect systems, and the benefit we gain is that we do not need the notion of stack-effect composition from the algebraic systems; instead, we use simple function composition.

The rule for value bindings, $\text{Lam}$, is also strange at first glance. The bound variable $x$ has a quantified function type, which means we only need one type rule for both let-bound and lambda-bound variables (recall that $\text{let}$ binds an expression to a variable, and all expressions have function type). Our intuition says that lambda-bound variables are indeed functions, albeit functions that just push the variable’s value onto the stack. We would like to be able to push this value onto any stack, which justifies our use of the quantified sequence type variable in the type rule: we can then instantiate the type of $x$ at multiple different stack types in $e_2$.

### 3.5 Type Soundness

The proof of soundness for the type rules in Fig. 5 uses the standard progress and preservation technique [21]. To continue with the main proofs, we require a few minor lemmas.

**Lemma 2.** (Well-typed Concatenation) If $\Gamma \vdash e_1 : i_1 \rightarrow o_1$ and $\Gamma \vdash e_2 : i_2 \rightarrow o_2$ and $o_1 = i_2$, then $\Gamma \vdash e_1 \cdot e_2 : i_1 \rightarrow o_2$.

**Proof.** By structural induction on $e_2$, taking the structure of $e_2$ to be left-associative. The base case has $e_2 = e$, and $e_1 \cdot e = e_1$, so $\Gamma \vdash e_1 \cdot e : i_1 \rightarrow o_1$ with $\Gamma \vdash e : o_1 \rightarrow o_1$ according to the premises. In each inductive case, we have $e_2 \equiv e_2' \cdot w$ for the word $w$ corresponding to the particular case. Here, $\Gamma \vdash e_2' : o_1 \rightarrow o_1'$ and $\Gamma \vdash w : o_1' \rightarrow o_2$. We apply the inductive hypothesis to get $\Gamma \vdash e_1 \cdot e_2' : i_1 \rightarrow o_1'$, and then the corresponding type rule to get $\Gamma \vdash (e_1 \cdot e_2') \cdot w : i_1 \rightarrow o_2$. $\square$

**Lemma 3.** (Well-typed Split) If $\Gamma \vdash e_1 \cdot e_2 : i \rightarrow o$, then there exists an $i'$ such that $\Gamma \vdash e_1 : i \rightarrow i'$ and $\Gamma \vdash e_2 : i' \rightarrow o$.

**Proof.** By structural induction on $e_2$. In the base case, $e_2 = e$, and so for any $i'$ such that $\Gamma \vdash e_1 : i \rightarrow i'$, we can derive $\Gamma \vdash e : i' \rightarrow i'$. In the inductive cases, we have $e_2 = e_2' \cdot w$ for the word $w$ corresponding to the particular case. Applying the inductive hypothesis, there exists some $i'$ such that $\Gamma \vdash e_1 : i \rightarrow i'$ and $\Gamma \vdash e_2' : i' \rightarrow o'$, with $\Gamma \vdash w : o' \rightarrow o$. Then we apply the type rule that gives us $\Gamma \vdash e_2' \cdot w : i' \rightarrow o$ to complete the case. $\square$
Lemma 4. (Substitution) If $\Gamma; x : \forall \alpha_1 \ldots \alpha_n. f \vdash e_1 : f'$ and $x \not\in \Gamma$ and $\Gamma \vdash e_2 : f$ and $(\alpha_1, \ldots, \alpha_n) \cap \text{ftv}(\Gamma) = \emptyset$, then $\Gamma \vdash [e_2/x]e_1 : f'$.

Proof. By induction on the typing derivation of $e_1$, and case analysis on the last rule used in the derivation. This is an adaptation of a similar proof in Wright & Felleisen [21].

We start by showing that reduction preserves types. The techniques used here are heavily adapted from Wright & Felleisen [21], but we rely on well-typed splitting and concatenation rather than evaluation contexts.

Lemma 5. (Preservation) If $\Gamma \vdash \langle s \mid e \rangle : o$ and $\langle s \mid e \rangle \to \langle s' \mid e' \rangle$, then $\Gamma \vdash \langle s' \mid e' \rangle : o$.

Proof. We proceed by case analysis on the transition rule $\langle s \mid e \rangle \to \langle s' \mid e' \rangle$. All rules begin by destructing the machine state into its typed components by Machine, so we only explicitly list this step in the first rule.

CASE $\langle s \mid n \ e \rangle \to \langle s \ n \ e \rangle$. We have $\Gamma \vdash s : i$ and $\Gamma \vdash n : e : i \to o$ by Machine. By Lemma 3 and Num we have $\Gamma \vdash n : i \to i \text{ Int}$ and $\Gamma \vdash e : i \text{ Int} \to o$. Then by VNum we have $\Gamma \vdash s \ n \ e : i \text{ Int}$, and by Machine we have $\Gamma \vdash \langle s \ n \ e \rangle : o$ as required.

CASE $\langle s \mid b \ e \rangle \to \langle s \ b \ e \rangle$. Similar to the case for numbers.

CASE $\langle s \mid \langle e_1 \rangle \ e_2 \rangle \to \langle s \mid \langle e_1 \rangle \ e_2 \rangle$. Similar to the case for numbers.

CASE $\langle s \mid n_2 \ n_1 \mid + \ e \rangle \to \langle s \mid (n_2 + n_1) \mid e \rangle$. By the type rules we have $\Gamma \vdash s \ n_2 \ n_1 : i \text{ Int} \text{ Int}$ and $\Gamma \vdash + \ e : i \text{ Int} \text{ Int} \to o$. By Lemma 3 and the type rules we have $\Gamma \vdash s : i$ and $\Gamma \vdash e : i \text{ Int} \to o$. Then by VNum we have $\Gamma \vdash s \ (n_2 + n_1) : i \text{ Int}$, and by Machine we have $\Gamma \vdash \langle s \mid (n_2 + n_1) \mid e \rangle : o$.

CASE $\langle s \ v_2 \ v_1 \ true \mid if \ e \rangle \to \langle s \ v_1 \mid e \rangle$. By the type rules we have $\Gamma \vdash s \ v_2 \ v_1 \ true : i \tau \tau \text{ Bool}$ and $\Gamma \vdash if \ e : i \tau \tau \text{ Bool} \to o$. By Lemma 3 and If, we have $\Gamma \vdash e : i \tau \to o$, and by multiple stack rules we have $\Gamma \vdash s : i$. Then by Stack we have $\Gamma \vdash s \ v_1 : i \tau$, and by Machine we have $\Gamma \vdash \langle s \ v_1 \mid e \rangle : o$ as required.

CASE $\langle s \ v_2 \ v_1 \ false \mid if \ e \rangle \to \langle s \ v_2 \mid e \rangle$. Similar to the case for if-expressions on a true condition, with $v_2$ in place of $v_1$ in the final line.

CASE $\langle s \mid \langle e_1 \rangle \mid \text{call} \ e_2 \rangle \to \langle s \mid e_1 \mid e_2 \rangle$. By the type rules we have $\Gamma \vdash s \mid \langle e_1 \rangle : i \ (i \to i')$ and $\Gamma \vdash \text{call} \ e_2 : i \ (i \to i') \to o$. By Stack and VBlock, we have $\Gamma \vdash s : i$ and $\Gamma \vdash e_1 : i \to i'$. By Lemma 3 and Call we have $\Gamma \vdash e_2 : i' \to o$. It follows from Lemma 2 that $\Gamma \vdash e_1 \cdot e_2 : i \to o$. Hence, by Machine we have $\Gamma \vdash \langle s \mid e_1 \cdot e_2 \rangle : o$ as required.
We continue by showing that a well-typed state can either take a step, or has
considering only the case where
is similar across all other derivation rules, so we skip it for the rest of them,
rule used in the derivation.

Next, by \textbf{Block, Fix}, and \textbf{VBlock} we have \( \Gamma \vdash \{e_1\} : i (i \to i') \to i' \). By \textbf{Lemma 3} and
\textbf{Fix} we have \( \Gamma \vdash e_2 : i' \to o \). By \textbf{Lemma 2} we have \( \Gamma \vdash e_1 \cdot e_2 : i (i \to i') \to o \).

Hence by \textbf{Stack} we have \( \Gamma \vdash s \{\{e_1\}\} : i (i \to i') \), and by \textbf{Machine} we have
\( \Gamma \vdash s \{\{e_1\}\} : e_1 \cdot e_2 \) as required.

\( \langle s \mid v \mid (\lambda x.e_1) \cdot e_2 \rangle \mapsto \langle s \mid [v/x]e_1 \cdot e_2 \rangle \). By the type rules we have \( \Gamma \vdash s \mathbf{\tau} : i \to o \) and \( \Gamma \vdash (\lambda x.e_1) \cdot e_2 : i \mathbf{\tau} \to o \). By \textbf{Lemma 3} and \textbf{Lam} we have \( \Gamma; x : \forall \tau. \mathbf{\alpha} \to \exists \mathbf{\alpha} \vdash e_1 : i \to i' \) where \( \exists \mathbf{\alpha} \not\in \text{ftv}(\Gamma) \), and \( \Gamma \vdash e_2 : i' \to o \). By \textbf{Stack} we have
\( \Gamma \vdash v : \mathbf{\tau} \) and \( \Gamma \vdash s : i \), and thus by \textbf{Lemma 4} we have \( \Gamma \vdash [v/x]e_1 : i \to i' \).

Hence by \textbf{Lemma 2} we have \( \Gamma \vdash [v/x]e_1 \cdot e_2 : i \to o \), and by \textbf{Machine} we have
\( \Gamma \vdash \langle s \mid [v/x]e_1 \cdot e_2 \rangle : o \) as required.

\( \langle s \mid \langle\text{let } x = e_1 \text{ in } e_2 \rangle \rangle \mapsto \langle s \mid [e_1/x]e_2 \cdot e_3 \rangle \). Then we have \( \Gamma \vdash s : \mathbf{i} \) and \( \Gamma \vdash \langle\text{let } x = e_1 \text{ in } e_2 \rangle \cdot e_3 : i \to o \). By \textbf{Lemma 3} and \textbf{Let} we have \( \Gamma \vdash e_1 : f \),
\( \Gamma; x : \hat{f}(x) \vdash e_2 : i \to i' \) and \( \Gamma \vdash e_3 : i' \to o \). Since \( \hat{f}(x) = \forall \alpha_1, \ldots, \alpha_n.f \) where \( \{\alpha_1, \ldots, \alpha_n\} = \text{ftv}(f) - \text{ftv}(\Gamma) \), we have by \textbf{Lemma 4} \( \Gamma \vdash [e_1/x]e_2 : i \to i' \), and
then by \textbf{Lemma 2} we have \( \Gamma \vdash [e_1/x]e_2 \cdot e_3 : i \to o \). Finally, by \textbf{Machine} we have
\( \Gamma \vdash \langle s \mid [e_1/x]e_2 \cdot e_3 \rangle : o \) as required.

We continue by showing that a well-typed state can either take a step, or has
an empty expression part.

\textbf{Lemma 6}. (Progress) If \( \emptyset \vdash \langle s \mid e \rangle : o \), then either \( e = e \) or there exists some \( e', e' \) such that \( \langle s \mid e \rangle \mapsto \langle s' \mid e' \rangle \).

\textbf{Proof}. By induction on the typing derivation of \( e \), and case analysis on the last
rule used in the derivation.

\textbf{Case Empty}. Then \( e = e \) and we have our result.

\textbf{Case Num}. Then \( e = e' \cdot n \). By the inductive hypothesis, we have either that
\( \langle s \mid e' \rangle \) can take a step, or \( e' = e \). Suppose \( \langle s \mid e' \rangle \mapsto \langle s' \mid e'' \rangle \); then we have
\( \langle s \mid e' \cdot n \rangle \mapsto \langle s \mid e'' \cdot n \rangle \) by \textbf{Lemma 2}. This case of the inductive hypothesis
is similar across all other derivation rules, so we skip it for the rest of them,
considering only the case where \( e' = e \). For the present rule, we then have
\( \langle s \mid n \rangle \mapsto \langle s \mid n \rangle \), as required.

\textbf{Case Bool, Block}. Similar to the case for \textbf{Num}.

\textbf{Case Var}. Vacuous: variables are not well-typed when \( \Gamma = \emptyset \).

\textbf{Case Add}. Then \( e = +. \) For \( \emptyset \vdash \langle s \mid + \rangle \) to be well-typed, \( s \) must be of the
form \( s' \cdot n_2 \cdot n_1 \). Hence we have \( \langle s' \cdot n_2 \cdot n_1 \mid + \rangle \mapsto \langle s' \cdot (n_2 + n_1) \mid e \rangle \).
case If. Then $e = \text{if}$. For $\emptyset \vdash \langle s \mid \text{if} \rangle$ to be well-typed, $s$ must be of the form $s' \; v_2 \; v_1 \; \text{true}$ or $s' \; v_2 \; v_1 \; \text{false}$. Hence we have either $\langle s' \; v_2 \; v_1 \; \text{true} \mid \text{if} \rangle \mapsto \langle s' \; v_1 \; e \rangle$ or $\langle s' \; v_2 \; v_1 \; \text{false} \mid \text{if} \rangle \mapsto \langle s' \; v_2 \; e \rangle$.

case Call. Then $e = \text{call}$. For $\emptyset \vdash \langle s \mid \text{call} \rangle$ to be well-typed, it must be the case that $s$ is of the form $s' \{e'\}$. Hence we have $\langle s' \{e'\} \mid \text{call} \rangle \mapsto \langle s' \mid e' \rangle$.

case Fix. Then $e = \lambda x . e'$. For $\emptyset \vdash \langle s \mid \lambda x . e' \rangle$ to be well-typed, it must be the case that $s$ is of the form $s' \; v$. Hence we have $\langle s' \; v \mid \lambda x . e' \rangle \mapsto \langle s' \mid [v/x]e' \rangle$.

case Lam. Then $e = \lambda x . e'$. For $\emptyset \vdash \langle s \mid \lambda x . e' \rangle$ to be well-typed, $s$ must be of the form $s' \; v$. Hence we have $\langle s' \; v \mid \lambda x . e' \rangle \mapsto \langle s' \mid [v/x]e' \rangle$.

case Let. Then $e = \text{let } x = e_1 \text{ in } e_2$, and we need no further information about $s$; we simply have $\langle s \mid \text{let } x = e_1 \text{ in } e_2 \rangle \mapsto \langle s \mid [e_1/x]e_2 \rangle$.

Finally, we get the desired result.

**Theorem 7.** (Type Soundness) For closed $e$, if $\emptyset \vdash e : o$ and $\langle e \mid e \rangle \mapsto \langle s \mid e' \rangle$, then either $e' = e$ and $\emptyset \vdash s : o$, or there exists $s'$ and $e''$ such that $\langle s \mid e' \rangle \mapsto \langle s' \mid e'' \rangle$.

**Proof.** The proof falls out of Lemmas 5 and 6 in the usual way. By induction and preservation on $\langle e \mid e \rangle \mapsto \langle s \mid e' \rangle$, we have $\emptyset \vdash \langle s \mid e' \rangle : o$, and by progress we have either that $\langle s \mid e' \rangle$ can take a step, or $e' = e$ and $\emptyset \vdash s : o$. □
TYPE INFERENCE FOR $\lambda_\omega$.

4.1 INFEERENCE ALGORITHM

The type system of the previous section admits a type inference algorithm in the style of Algorithm W [9], defined by Algorithm 1. This algorithm computes the principal type of an expression under some type environment, if such a type exists; otherwise, it fails. In addition to the type environment and the expression, we supply the algorithm with a set of ‘fresh’ type variables $V$, where all variables in $V$ are disjoint. The successful results of the algorithm are a substitution $S$, the inferred type $f$, and the subset of $V$ that was not consumed during inference. The substitution returned by the inference algorithm is used to propagate the results of unification to the enclosing context.

Type inference is dependent on an algorithm for unification of type sequences, which yields a substitution $\Phi$ if the sequences can be unified. We also make use of an auxiliary function $\text{inst}(\sigma, V)$:

$$\text{inst}(\forall (\alpha_i, ..., \alpha_j, ...), f, V) = ([\alpha_i \mapsto \beta_i, ..., \alpha_j \mapsto \beta_j, ...]f, V - \{\beta_i, ..., \beta_j, ...\})$$

where $\beta_i, ..., \beta_j, ... \in V$

That is, $\text{inst}$ substitutes fresh type variables for the quantified type variables in $\sigma$ and returns the instantiated type along with the subset of fresh type variables not used by the instantiation. We explicitly require that quantified individual type variables map to fresh individual type variables, and similarly for sequence variables.

4.2 UNIFICATION ALGORITHM

The unification method used in the inference algorithm is defined in Algorithm 2. This algorithm is split into two parts: a variant of Robinson unification specialized for the individual types $\tau$ of $\lambda_\omega$, and unification of type sequences optionally terminated by a left-most sequence variable. It is known that unification in the presence of sequence variables is unitary when all sequences in both terms have either no sequence variables or a single left-most sequence variable [5]. Our algorithm is tailored for this case. We require two more properties when investigating the proof of algorithmic completeness:

1. $\text{ftv}(\text{Unify}(i, i')) \subseteq \text{ftv}(i) \cup \text{ftv}(i')$

2. $\text{Unify}(i, i') = \Phi$ implies that $\Phi$ is the most general unifier of $i$ and $i'$
Algorithm 1 Inference for $\lambda_0$

\[
\textbf{function Infer}(\Gamma, e, V) \quad \triangleright \text{Infer : } (\Gamma, e, V) \rightarrow (S, f, V)
\]

\[
\begin{align*}
\text{if } e \text{ is } \epsilon & \text{ then } \\
(\emptyset, \overline{\epsilon}, V - \{\overline{\epsilon}\}) \text{ where } \overline{\epsilon} \in V & \\
\text{else if } e \text{ is } e' \text{ n then } & \\
(S, i \rightarrow o, V') \leftarrow \text{Infer}(\Gamma, e', V) & \\
(S, i \rightarrow o \text{ Int, } V') & \\
\text{else if } e \text{ is } e' \text{ b then } & \\
(S, i \rightarrow o, V') \leftarrow \text{Infer}(\Gamma, e', V) & \\
(S, i \rightarrow o \text{ Bool, } V') & \\
\text{else if } e \text{ is } e' + \text{ then } & \\
(S, i \rightarrow o, V') \leftarrow \text{Infer}(\Gamma, e', V) & \\
\Phi \leftarrow \text{Unify}(o, \overline{\epsilon} \text{ Int Int}) \text{ where } \overline{\epsilon} \in V' & \\
(\Phi \circ S, \Phi(i \rightarrow \overline{\epsilon} \text{ Int}), V' - \{\overline{\epsilon}\}) & \\
\text{else if } e \text{ is } e' \text{ if then } & \\
i_2 \rightarrow o_2, V' \leftarrow \text{inst}(\Gamma(x), V) & \\
(S, i_1 \rightarrow o_1, V'') \leftarrow \text{Infer}(\Gamma, e', V') & \\
\Phi \leftarrow \text{Unify}(o_1, S(i_2)) & \\
(\Phi \circ S, \Phi(i_1 \rightarrow S(o_2)), V'') & \\
\text{else if } e \text{ is } e' \text{ call then } & \\
(S, i \rightarrow o, V') \leftarrow \text{Infer}(\Gamma, e', V) & \\
\Phi \leftarrow \text{Unify}(o, \overline{\epsilon} \beta \rightarrow \overline{\beta}) \text{ where } \overline{\epsilon}, \beta \in V' & \\
(\Phi \circ S, \Phi(i \rightarrow \beta), V' - \{\overline{\epsilon}, \beta\}) & \\
\text{else if } e \text{ is } e' \text{ fix then } & \\
(S, i \rightarrow o, V') \leftarrow \text{Infer}(\Gamma, e', V) & \\
\Phi \leftarrow \text{Unify}(o, \overline{\epsilon} \overline{\beta} \rightarrow \overline{\beta}) \text{ where } \overline{\epsilon}, \overline{\beta} \in V' & \\
(\Phi \circ S, \Phi(i \rightarrow \overline{\beta}), V' - \{\overline{\epsilon}, \overline{\beta}\}) & \\
\text{else if } e \text{ is } e_1 \text{ let } x = e_2 \text{ in } e_3 \text{ then } & \\
(S_1, i_1 \rightarrow o_1, V') \leftarrow \text{Infer}(\Gamma, e_1, V) & \\
(S_2, i_2 \rightarrow o_2, V'') \leftarrow \text{Infer}(S_1(\Gamma); x : \forall \overline{\epsilon} \overline{\beta} \rightarrow \overline{\epsilon} \beta, e_2, V' \rightarrow \{\beta\}) \text{ where } \beta \in V' & \\
\Phi \leftarrow \text{Unify}(S_2(o_1), i_2 S_2(\beta)) & \\
(\Phi \circ S_2 \circ S_1, \Phi(S_2(i_1) \rightarrow o_2), V'') & \\
\text{else if } e \text{ is } e_1 \text{ \lambda} x e_2 \text{ then } & \\
(S_1, i_1 \rightarrow o_1, V') \leftarrow \text{Infer}(\Gamma, e_1, V) & \\
(S_2, i_2 \rightarrow o_2, V'') \leftarrow \text{Infer}(S_1(\Gamma); x : S_2(\Gamma)(f_2), e_3, V'') & \\
\Phi \leftarrow \text{Unify}(S_3(S_2(o_1)), i_3) & \\
(\Phi \circ S_3 \circ S_2 \circ S_1, \Phi(S_3(S_2(i_1)) \rightarrow o_3), V'') & \\
\text{end if } & \\
\end{align*}
\]

end function
Algorithm 2 Unification for $\lambda_0$ Types

```
function UNIFY(i_1, i_2) \(\triangleright \) Unify : (i,i) \rightarrow \Phi
    if i_1, i_2 is e, e then []
    else if i_1 is \(\overline{\pi}\) and i_2 is \(\overline{\pi}\) then []
    else if i_1 is \(\overline{\pi}\) and \(\overline{\pi}\) \(\notin\) ftv(i_2) then [\(\overline{\pi}\mapsto i_2\)]
    else if i_2 is \(\overline{\pi}\) and \(\overline{\pi}\) \(\notin\) ftv(i_1) then [\(\overline{\pi}\mapsto i_1\)]
    else if i_1 is \(i'_1\) \(\tau_1\) and i_2 is \(i'_2\) \(\tau_2\) then
        \(\Phi_1 \leftarrow \text{UnifyInd}(\tau_1, \tau_2)\)
        \(\Phi_2 \leftarrow \text{Unify}(\Phi_1 i'_1, \Phi_1 i'_2)\)
        \(\Phi_2 \circ \Phi_1\)
    end if
end function

function UNIFYIND(\(\tau_1, \tau_2\)) \(\triangleright \) UnifyInd : (\(\tau, \tau\)) \rightarrow \Phi
    if \(\tau_1, \tau_2\) is Int, Int then []
    else if \(\tau_1, \tau_2\) is Bool, Bool then []
    else if \(\tau_1\) is \(\alpha\) and \(\alpha\) \(\notin\) ftv(\(\tau_2\)) then [\(\alpha\mapsto \tau_2\)]
    else if \(\tau_2\) is \(\alpha\) and \(\alpha\) \(\notin\) ftv(\(\tau_1\)) then [\(\alpha\mapsto \tau_1\)]
    else if \(\tau_1, \tau_2\) is \(i_1 \rightarrow o_1, i_2 \rightarrow o_2\) then
        \(\Phi_1 \leftarrow \text{Unify}(i_1, i_2)\)
        \(\Phi_2 \leftarrow \text{Unify}(\Phi_1 i_1, \Phi_1 i_2)\)
        \(\Phi_2 \circ \Phi_1\)
    end if
end function
```

4.3 Correctness and Completeness of the Algorithm

The proofs of correctness and completeness are all adapted from Leroy’s proofs for a functional language with a Hindley-Milner type system [6].

Lemma 8. (Typing is Stable under Substitution) For all substitutions \(S\), if \(\Gamma \vdash e : f\) then \(S(\Gamma) \vdash e : S(f)\).

Proof. By structural induction on \(e\), where the structure of \(e\) is left-associative. The proof found in Leroy [6] can be adapted to the syntax and type-system of \(\lambda_0\) with mostly cosmetic changes.

Theorem 9. (Inference is Correct) Let \(e\) be an expression, \(\Gamma\) a type environment, and \(V\) a set of fresh type variables. If \(\text{Infer}(\Gamma, e, V) = (S, f, V')\), then \(S(\Gamma) \vdash e : f\).

Proof. We proceed by induction over \(e\), taking the structure of \(e\) here to be left-associative. Throughout the proof we use the fact that our unification algorithm produces a most-general unifier [5].

Case \(e = e\). We have \(\text{Infer}(\Gamma, e, V) = (\emptyset, \overline{\pi} \rightarrow \overline{\pi}, V')\). By \text{Empty} we can derive \(\Gamma \vdash e : \overline{\pi} \rightarrow \overline{\pi}\), and we have \(\emptyset(\Gamma) = \Gamma\) and \(\emptyset(\overline{\pi} \rightarrow \overline{\pi}) = \overline{\pi} \rightarrow \overline{\pi}\) as required.

Case \(e = e' n\). We apply the induction hypothesis to the recursive call of \(\text{Infer}\), yielding \(S(\Gamma) \vdash e' : i \rightarrow o\). By \text{Num}, we then have \(S(\Gamma) \vdash e' n : i \rightarrow o \text{Int}\), which is the type returned by the algorithm.
case  \( e = e' \, b \). Similar to the case for numbers.

case  \( e = e_1 \{ e_2 \} \). We apply the induction hypothesis to the recursive calls of \texttt{Infer}, which yields

\[
S_1(\Gamma) \vdash e_1 : i_1 \rightarrow o_1 \\
S_2(S_1(\Gamma)) \vdash e_2 : i_2 \rightarrow o_2
\]

By Lemma 8 we have \( S_2(S_1(\Gamma)) \vdash e_1 : S_2(i_1 \rightarrow o_1) \), and by the definition of substitution \( S_2(i_1 \rightarrow o_1) = S_2(i_1) \rightarrow S_2(o_1) \). Hence by \texttt{Block} we have \( S_2(S_1(\Gamma)) \vdash e_1 \{ e_2 \} : S_2(i_1) \rightarrow S_2(o_1) (i_2 \rightarrow o_2) \), and this is the type returned by the algorithm.

case  \( e = e' + \). Applying the induction hypothesis to the recursive call of \texttt{Infer} yields \( S(\Gamma) \vdash e' : i \rightarrow o \). For the algorithm to succeed, we must have \( \texttt{Unify}(o, \alpha \texttt{Int Int}) = \Phi \). Hence, by Lemma 8 we have \( \Phi(S(\Gamma)) \vdash e' : \Phi(i \rightarrow o) \).

Since \( \Phi \) is a most-general unifier [5], we have \( \Phi(S(\Gamma)) \vdash e' : \Phi(i \rightarrow \alpha \texttt{Int Int}) \). By \texttt{Add} we then have \( \Phi(S(\Gamma)) \vdash e' + : \Phi(i \rightarrow \alpha \texttt{Int}) \), which is the type returned by the algorithm.

case  \( (e = e' \, \text{if}) \) or \( (e = e' \, \text{call}) \) or \( (e = e' \, \text{fix}) \). These cases are similar to the case for addition.

case  \( e = e' \, x \). Applying the induction hypothesis to the recursive call of \texttt{Infer} yields \( S(\Gamma) \vdash e' : i_1 \rightarrow o_1 \). We then have \( \texttt{inst}(S(\Gamma)(x), V') = (i_2 \rightarrow o_2, V'') \). By definition of \( \texttt{inst} \), we have \( S(\Gamma)(x) \subseteq i_2 \rightarrow o_2 \). For the algorithm to succeed, we must have \( \texttt{Unify}(o_1, i_2) = \Phi \). By Lemma 8 we have \( \Phi(S(\Gamma)) \vdash e' : \Phi(i_1 \rightarrow o_1) \). Since \( \Phi(o_1) = \Phi(i_2) \), we have by \( \texttt{Var} \) \( \Phi(S(\Gamma)) \vdash e' \, x : \Phi(i_1 \rightarrow o_2) \), and this is the type returned by the algorithm.

case  \( e = e_1 \, \lambda x . e_2 \). Applying the induction hypothesis to the recursive calls of \texttt{Infer} yields

\[
S_1(\Gamma) \vdash e_1 : i_1 \rightarrow o_1 \\
S_2(S_1(\Gamma)); x : S_2(\forall \alpha. \alpha \rightarrow \beta) \vdash e_2 : i_2 \rightarrow o_2
\]

By Lemma 8 we have \( S_2(S_1(\Gamma)) \vdash e_1 : S_2(i_1 \rightarrow o_1) \). For the algorithm to succeed, we must have \( \texttt{Unify}(o_1, i_2) = \Phi \). Hence, by Lemma 8 again, we have:

\[
\Phi(S_2(S_1(\Gamma))) \vdash e_1 : \Phi(S_2(i_1 \rightarrow o_1)) \\
\Phi(S_2(S_1(\Gamma))); x : \Phi(S_2(\forall \alpha. \alpha \rightarrow \beta)) \vdash e_2 : \Phi(i_2 \rightarrow o_2)
\]

Since \( \Phi \) is a most-general unifier, we have \( \Phi(S_2(o_1)) = \Phi(i_2 \, S_2(\beta)) \), so by \texttt{Lam} we have \( \Phi(S_2(S_1(\Gamma))) \vdash e_1 \, \lambda x . e_2 : \Phi(S_2(i_1 \rightarrow o_2) \), and this is the type returned by the algorithm.
Again, the Leroy proof Theorem variables, defined as: Section Proof.
then type environments with the same domain. If Lemma substitutions and a substitution Proof.
that, since rem 9.

For the algorithm to succeed, we must also have \( \text{Unify}(S_3(S_2(o_1)),i_3) = \Phi \). Then, as \( \Phi(S_3(S_2(o_1))) = \Phi(i_3) \), we have by Let that \( \Phi(S_3(S_2(S_1(\Gamma)))) \vdash e_1 \text{ let } x = e_2 \text{ in } e_3 : \Phi(S_2(S_1(\Gamma))) \rightarrow o_3 \), and this is the type returned by the algorithm.

To make the statement of the completeness property less verbose, we introduce a notation \( S - V \), where \( S \) is a substitution and \( V \) a set of disjoint type variables, defined as:

\[
S - V = [\alpha_i \mapsto \tau_i] \text{ for all } (\alpha_i,\tau_i) \in S \text{ where } \alpha_i \notin V
\]

Lemma 10. (Well-typed Under More-general Environment). Let \( \Gamma \) and \( \Gamma' \) be two type environments with the same domain. If \( \Gamma'(x) \subseteq \Gamma(x) \) for all \( x \in \Gamma \) and \( \Gamma \vdash e : f \), then \( \Gamma' \vdash e : f \).

Proof. By easy structural induction on \( e \), where the structure is left-associative. Again, the Leroy proof [6] is easily adapted to \( \lambda_0 \) with the definition of \( \subseteq \) from Section 3.4.

Theorem 11. (Inference is Complete) Let \( e \) be an expression, \( \Gamma \) a type environment, and \( V \) a set of fresh type variables such that \( V \cap \text{ftv}(\Gamma) = \emptyset \). If there exists a type \( f' \) and a substitution \( S' \) such that \( S'(\Gamma) \vdash e : f' \), then there exist a type \( f \) and substitutions \( S,R \) such that the following hold:

1. \( \text{Infer}(\Gamma,e,V) = (S,f,f') \) for some \( V' \subseteq V \)
2. \( f' = R(f) \)
3. \( (R \circ S) - V = S' \)

Proof. We proceed by structural induction over \( e \). Like for the proof of Theorem 9, we specify here that the structure of \( e \) is left-associative. We also note that, since \( V' \subseteq V \), we have \( V' \cap \text{ftv}(f) = \emptyset \) and \( V' \cap \text{ftv}(S(\Gamma)) = \emptyset \).

Case \( e = e \). In this case we let \( S' = \emptyset \). The typing derivation ends with \( \Gamma \vdash e : i \rightarrow i \). For \( e = e \), the algorithm always succeeds, yielding type \( \overline{\beta} \rightarrow \overline{\beta} \) and substitution \( \emptyset \), where \( \overline{\beta} \in V \). We let \( R = [\overline{\beta} \mapsto i] \), and thus we have

1. \( R(\overline{\beta} \rightarrow \overline{\beta}) = i \rightarrow i \)
2. \( (R \circ \emptyset) - V = R - V = \emptyset \)

as required.
case \( e = e' \) \( n \). The typing derivation ends with
\[
S'(\Gamma) \vdash e': i \rightarrow o \\
\frac{S'(\Gamma) \vdash e \ n: i \rightarrow o \ \\text{Int}}{S'(\Gamma) \vdash e \ n: i \rightarrow o \ \text{Int}}
\]

We apply the inductive hypothesis to \( e', \Gamma, V, i \rightarrow o \) and \( S' \) to get
\[
\text{Infer}(\Gamma, e', V) = (S, i' \rightarrow o', V') \\
R(i' \rightarrow o') = i \rightarrow o \\
(R \circ S) - V = S'
\]

The algorithm succeeds with type \( i' \rightarrow o' \ \text{Int} \) and the same substitution \( S \) from the inductive hypothesis. This allows us to reuse the substitution \( R \) from the inductive hypothesis for the remaining two properties.

case \( e = e' \) \( b \). Similar to the case for numbers.

case \( e = e_1 \{ e_2 \} \). The typing derivation ends with
\[
S'(\Gamma) \vdash e_1: i_1 \rightarrow o_1 \\
S'(\Gamma) \vdash e_2: i_2 \rightarrow o_2 \\
\frac{S'(\Gamma) \vdash e_1 \{ e_2 \}: i_1 \rightarrow o_1 \ (i_2 \rightarrow o_2)}{S'(\Gamma) \vdash e_1 \{ e_2 \}: i_1 \rightarrow o_1 \ (i_2 \rightarrow o_2)}
\]

We apply the inductive hypothesis to \( e_1, \Gamma, V, i_1 \rightarrow o_1 \), and \( S' \) to get
\[
\text{Infer}(\Gamma, e_1, V) = (S_1, i'_1 \rightarrow o'_1, V') \\
R_1(i'_1 \rightarrow o'_1) = i_1 \rightarrow o_1 \\
(R_1 \circ S_1) - V = S'
\]

We then apply the inductive hypothesis to \( e_2, S_1(\Gamma), V', i_2 \rightarrow o_2 \) and \( R_1 \).
\[
\text{Infer}(S_1(\Gamma), e_2, V') = (S_2, i'_2 \rightarrow o'_2, V'') \\
R_2(i'_2 \rightarrow o'_2) = i_2 \rightarrow o_2 \\
(R_2 \circ S_2) - V' = R_1
\]

The remark at the beginning of the proof assures us this is possible. Since \( \text{ftv}(V') \cap \text{ftv}(i'_1 \rightarrow o'_1) \), we have that \( R_1(i'_1 \rightarrow o'_1) = R_2(S_2(i'_1 \rightarrow o'_1)) \). As the algorithm succeeded for both of its recursive calls, it then yields the type \( S_2(i'_1) \rightarrow S_2(o'_1) \ (i'_2 \rightarrow o'_2) \) and the substitution \( S_2 \circ S_1 \). We let \( R = R_2 \), and then we have
\[
R_2(S_2(i'_1) \rightarrow S_2(o'_1) \ (i'_2 \rightarrow o'_2)) = R_1(i'_1) \rightarrow R_1(o'_1) \ (R_2(i'_2) \rightarrow R_2(o'_2)) \\
= i_1 \rightarrow o_1 \ (i_2 \rightarrow o_2) \\
(R_2 \circ S_2 \circ S_1) - V = (R_1 \circ S_1) - V = S'
\]
as required.
We apply the inductive hypothesis to \( e' \), \( \Gamma \), \( V \), \( i \rightarrow \alpha \) \( \text{Int} \) \( \text{Int} \), and \( S' \) to get

\[
\text{Infer}(\Gamma, e', V) = (S, i' \rightarrow \alpha', V') \\
R(i' \rightarrow \alpha') = i \rightarrow \alpha \text{Int} \text{Int} \\
(R \circ S) - V = S'
\]

Let \( \Phi = [\overline{\alpha} \rightarrow o] \circ R \), where \( \overline{\alpha} \in V' \). We then have

\[
\Phi(\alpha') = R(\alpha') = \alpha \text{Int} \text{Int} \\
\Phi(\text{Int} \text{Int}) = \alpha \text{Int} \text{Int}
\]

Thus \( \Phi \) is a unifier of \( \alpha' \) and \( \overline{\alpha} \) \( \text{Int} \) \( \text{Int} \). Because sequence unification with only left-most sequence variables is unitary, these two sequences have a most-general unifier \( \Phi' \). Hence the algorithm is defined, yielding type \( \Phi'(i' \rightarrow \overline{\alpha} \text{Int} \text{Int}) \) and substitution \( \Phi' \circ S \). By the definition of most-general unifier, we have \( \Phi = S_\Phi \circ \Phi' \) for some substitution \( S_\Phi \). We then have

\[
S_\Phi(\Phi'(i' \rightarrow \overline{\alpha} \text{Int} \text{Int})) = \Phi(i' \rightarrow \overline{\alpha} \text{Int} \text{Int}) = R(i) = i \rightarrow o \text{Int} = i \rightarrow o \text{Int} \\
(S_\Phi \circ \Phi' \circ S) - V = (\Phi \circ S) - V = (R \circ S) - V = S'
\]

as required.

**Case** \( e = e' + \). The typing derivation ends with

\[
S'(\Gamma) \vdash e' : i \rightarrow o \text{Int} \text{Int} \\
S'(\Gamma) \vdash e' + : i \rightarrow o \text{Int} \text{Int}
\]

We apply the inductive hypothesis to \( e' \), \( \Gamma \), \( V \), \( i \rightarrow o \text{Int} \text{Int} \), and \( S' \) to get

\[
\text{Infer}(\Gamma, e', V) = (S, i' \rightarrow o', V') \\
R(i' \rightarrow o') = i \rightarrow o \text{Int} \text{Int} \\
(R \circ S) - V = S'
\]

**Case** \( e = e' \) if or \( e = e' \) \text{call} or \( e = e' \) \text{fix}. These are similar to the case for addition.

**Case** \( e = e' \) \text{x}. The typing derivation ends with

\[
x : \sigma \in S'(\Gamma) \\
S'(\Gamma) \vdash e' : i_1 \rightarrow o_1 \\
S'(\Gamma) \vdash e' \text{x} : i_1 \rightarrow o_2
\]

Applying the inductive hypothesis to \( e' \), \( \Gamma \), \( V \), \( i_1 \rightarrow o_1 \), and \( S' \), we get

\[
\text{Infer}(\Gamma, e', V) = (S, i'_1 \rightarrow o'_1, V') \\
R(i'_1 \rightarrow o'_1) = i_1 \rightarrow o_1 \\
(R \circ S) - V = S'
\]

Since \( S'(\Gamma) \vdash e' \text{x} : i_1 \rightarrow o_2 \), we know that \( x \in S'(\Gamma) \) and \( S'(\Gamma)(x) \subseteq i_2 \rightarrow o_2 \). Hence \( x \in \Gamma \), and we have \( i'_2 \rightarrow o'_2 = [\alpha_1 \rightarrow \beta_1, ..., \alpha_n \rightarrow \beta_n][i_x \rightarrow o_x] \) and \( V'' = V' - \{\beta_1, ..., \beta_n\} \) according to the definition of \text{inst}, where \( \Gamma(x) = \forall \alpha_1, ..., \alpha_n, i_x \rightarrow o_x \), where \( \forall i \in \{1, ..., n\} \) we have \( \alpha_i \in V'' \) and thus the \( \alpha_i \) are out of reach for \( S' \) and \( R \).

Then we have \( S'(\Gamma)(x) = \forall \alpha_1, ..., \alpha_n, S'(i_x \rightarrow o_x) \). Let \( Q \) be the substitution over the \( \alpha_i \) such that \( i_2 \rightarrow o_2 = Q(S'(i_x \rightarrow o_x)) \), and let \( R' = Q \circ S' \circ [\beta_1 \rightarrow \)
\(\alpha_1, \ldots, \alpha_n \mapsto \alpha_n\). We then have \(R'(i'_2 \rightarrow o'_2) = Q(S'(i_x \rightarrow o_x)) = i_2 \rightarrow o_2\). As \(o_1 = i_2\), we have \(R(o'_1) = R'(i'_2) = Q(S'(i'_2)) = Q(R(S(i_x))) = R(S(Q(i_x)))\) since the \(\alpha_1\) are out of reach for \(S'\) and \(R\).

\(R\) is a unifier of \(o'_1\) and \(S(Q(i_x))\), so there exists a most-general unifier \(\Phi\) for the two sequences, and there exists a substitution \(R_1\) such that \(R = R_1 \circ \Phi\). Hence, the algorithm succeeds, yielding type \(\Phi(i'_1 \rightarrow S(Q(o_x)))\) and substitution \(\Phi \circ S\). We then have

\[
R_1(\Phi([i'_1 \rightarrow S(Q(o_x))])) = i_1 \rightarrow R(S(Q(o_x))) = i_1 \rightarrow o_2
\]

\((R_1 \circ \Phi \circ S) - V = (R \circ S) - V = S'\)

as required.

**Case**  
\(e = e_1\lambda x.e_2\). The typing derivation ends with

\[
S'(\Gamma) \vdash e_1 : i_1 \rightarrow o_1
\]

\[
S'(\Gamma); x : \forall \alpha.\overline{\alpha} \rightarrow \overline{\alpha} \tau \vdash e_2 : i_2 \rightarrow o_2 \quad o_1 = i_2 \tau
\]

Applying the induction hypothesis to \(e_1, \Gamma, V, i_1 \rightarrow o_1\), and \(S'\), we get

\[
\text{Infer}(\Gamma, e_1, V) = (S_1, i'_1 \rightarrow o'_1, V')
\]

\[
R_1(i'_1 \rightarrow o'_1) = i_1 \rightarrow o_1
\]

\((R_1 \circ S_1) - V = S'\)

Let the environment \(\Gamma_1\) be \(\Gamma; x : \forall \alpha.\overline{\alpha} \rightarrow \overline{\alpha} \beta\) and the substitution \(S'_1\) be \([\beta \mapsto \tau] \circ S'\), where \(\beta\) is taken from \(V'\). We apply the induction hypothesis to \(e_2, S_1(\Gamma_1), V' - \{\beta\}, i_2 \rightarrow o_2\) and \(R_1\) to get

\[
\text{Infer}(S_1(\Gamma), e_2, V' - \{\beta\}) = (S_2, i'_2 \rightarrow o'_2, V'')
\]

\[
R_2(i'_2 \rightarrow o'_2) = i_2 \rightarrow o_2
\]

\((R_2 \circ S_2) - V = R_1\)

Since \(o_1 = i_2 \tau\), we have \(R_1(o'_1) = R_2(S_2(o'_1)) = R_2(i'_2) \tau\). Hence, \(R_2\) is a unifier of \(S_2(o'_1)\) and \(i'_2 \tau\), so there must be a most general unifier \(\Phi\) of the two type sequences, and there exists a substitution \(R_3\) such that \(R_2 = R_3 \circ \Phi\). It follows that the algorithm is well-defined, yielding type \(\Phi(S_2(i'_1) \rightarrow o'_2)\) and substitution \(\Phi \circ S_2 \circ S_1\). Then we have

\[
R_3(\Phi(S_2(i'_1) \rightarrow o'_2)) = R_2(S_2(i'_1)) \rightarrow R_2(o'_2) = i_1 \rightarrow o_2
\]

\((R_3 \circ \Phi \circ S_2 \circ S_1) - V = (R_2 \circ S_2 \circ S_1) - V = (R_1 \circ S_1) - V = S'\)

as required.

**Case**  
\(e = e_1\ let x = e_2\ in e_3\). Then the typing derivation ends with

\[
S'(\Gamma) \vdash e_2 : f \quad S'(\Gamma); x : S'(\Gamma)(f) \vdash e_3 : i_2 \rightarrow o_2
\]

\[
S'(\Gamma) \vdash e_1 : i_1 \rightarrow o_1 \quad o_1 = i_2
\]

\[
S'(\Gamma) \vdash e_1 \ let x = e_2 \ in e_3 : i_1 \rightarrow o_2
\]
Applying the induction hypothesis to $e_1$, $\Gamma$, $V$, $i_1 \rightarrow o_1$, and $S'$ yields

\begin{align*}
\text{Infer}(\Gamma, e_1, V) &= (S_1, i_1' \rightarrow o_1', V') \\
R_1(i_1' \rightarrow o_1') &= i_1 \rightarrow o_1 \\
(R_1 \circ S_1) - V &= S'
\end{align*}

We then apply the induction hypothesis to $e_2$, $S_1(\Gamma)$, $V'$, $f$, and $R_1$ to get

\begin{align*}
\text{Infer}(S_1(\Gamma), e_2, V') &= (S_2, f', V'') \\
R_2(f') &= f \\
(R_2 \circ S_2) - V &= R_1
\end{align*}

We have that $S'(\Gamma) = R_1(S_1(\Gamma)) = R_2(S_2(\Gamma))$. Since we know $S'(\Gamma); x : S'(\Gamma)(f) \vdash e_3 : i_2 \rightarrow o_2$, we have by Lemma 10 that $R_2(S_2(S_1(\Gamma)); x : S_2(S_1(\Gamma))(f')) \vdash e_3 : i_2 \rightarrow o_2$. We now apply the induction hypothesis to $e_3$ with $S_2(S_1(\Gamma)); x : S_2(S_1(\Gamma))(f')$, $V''$, $i_3 \rightarrow o_3$, and $R_2$ to get

\begin{align*}
\text{Infer}(S_2(S_1(\Gamma)); x : S_2(S_1(\Gamma))(f'), e_3, V'') &= (S_3, i_2' \rightarrow o_2', V''') \\
R_3(i_2' \rightarrow o_2') &= i_2 \rightarrow o_2 \\
(R_3 \circ S_3) - V &= R_2
\end{align*}

As $o_1 = i_2$, we have $R_1(o_1') = R_3(S_3(S_2(o_1')))) = R_3(i_2')$. Thus, $R_3$ is a unifier of $S_3(S_2(o_1')) and $i_2'$, so there must be a most general unifier $\Phi$ of the two type sequences, and there exists a substitution $R_4$ such that $R_3 = R_4 \circ \Phi$. It follows that the algorithm is well-defined, yielding type $\Phi(S_3(S_2(i_1')) \rightarrow o_2')$ and substitution $\Phi \circ S_3 \circ S_2 \circ S_1$. Then we have

\begin{align*}
R_4(\Phi(S_3(S_2(i_1')) \rightarrow o_2')) &= R_3(S_3(S_2(i_1')) \rightarrow o_2') = i_1 \rightarrow o_2 \\
(R_4 \circ \Phi \circ S_3 \circ S_2 \circ S_1) - V &= (R_3 \circ S_3 \circ S_2 \circ S_1) - V = ... = S'
\end{align*}

as required.
IMPLEMENTATION

A variant of $\lambda_0$ has been implemented as a \texttt{#lang} in Racket. The variant, named \texttt{wort}, is semantically similar to $\lambda_0$ with a slightly different syntax and two additional primitives for comparison on numeric values. The inference algorithm of the implementation is somewhat simpler than the one presented above, but equivalent; however, the paper algorithm is easier to write proofs about since it lacks mutual recursion. The code is well documented and deliberately simple to improve readability; the whole package is around 700 lines of Racket distributed over eight files. A few small sample programs are included with the package; Figure 6 shows one of the samples in the DrRacket IDE that comes with the Racket installation. Running a \texttt{wort} program in DrRacket will display both the results of evaluation and the inferred type of the expression. The repository can be found online at either the Racket package listings as ‘wort’, or on GitHub at \url{https://github.com/robertkleffner/wort}.

![A wort program in DrRacket](image)

Figure 6: A wort program in DrRacket
CONCLUSION & FUTURE DIRECTIONS

Concatenative languages share many useful theoretical properties with their widely studied functional counterparts. I presented a core concatenative language with variable binding, then formulated and proved sound a polymorphic type system for this language. In addition, I described a type inference algorithm for this language, then proved it sound and complete with respect to the type system.

There are many directions that research on concatenative languages might take. In section 2 we examined a particular semantics that bears some resemblance to call-by-value; it would be interesting to investigate whether concatenative languages might have a call-by-name semantics, and how it would differ from the semantics presented here. \( \lambda_0 \) also has striking parallels with the call-by-push-value paradigm [7] which may warrant further investigation.

In the type syntax of \( \lambda_0 \), we require sequence type variables to appear only at the left-most position of a type sequence. This restriction made unification unitary in the inference algorithm. If we abandon type inference and move to explicit types, the restriction could be lifted. A concatenative calculus with sequence type variables presents interesting challenges, and what a richer type system would look like for concatenative languages is an interesting question; even a simply-typed variant of \( \lambda_0 \) is somewhat tricky to formulate properly. The syntax, semantics, and type system of \( \lambda_0 \) provides a starting point for potential answers to these questions.


