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

Robert Kleffner: *A Foundation for Typed Concatenative Languages,* April 2017

Northeastern University
College of Computer and Information Science

MS Thesis Approval

THESIS TITLE:
A Foundation for Typed Concatenative Languages

AUTHOR: Robert Kleffner

M.S. Thesis Approved as an elective towrds the Master of Science Degree in Computer Science.

Thesis Advisor	*Date* 28 Apr 2017
Thesis Reader	*Date* 1 May 2017
Thesis Reader	*Date*
Thesis Reader	*Date*
Thesis Reader	*Date*

GRADUATE SCHOOL APPROVAL:

Director, Graduate School — *Date* 5/1/17

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Recipient's Signature — *Date* 5/1/17

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.

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.

1 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 passsed 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 essentially 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:

$\texttt{call}_{1,1} : (a\ (a \to z) \to z)$
$\texttt{call}_{2,1} : (a\ b\ (a\ b \to z) \to z)$
$\texttt{call}_{1,2} : (a\ (a \to y\ z) \to y\ z)$
etc...

With this solution, the programmer must redefine higher-order functions at each arity; naturally, they would like to write `map` and `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:

$$\texttt{call} : (s, (s \to s')) \to s'$$

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 λ-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 $\lambda_\circ$, 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 $\lambda_\circ$ built in Racket. This implementation supports both type inference and evaluation of type-checked programs.

2

THE LANGUAGE $\lambda_\circ$

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 $\lambda_\circ$ [1], is a concatenative language with a type system that enforces this distinction syntactically.

2.1 SYNTAX

The syntax for $\lambda_\circ$ is given in Fig. 1. In contrast to languages based on the λ-calculus, $\lambda_\circ$ 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 $\lambda_\circ$ 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 `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 [7].

1 The symbol for composition as a subscript emphasizes the importance our language places on composability as a design concept.

$$
\begin{array}{rcl}
e & := & \overrightarrow{w} \\
w & := & n \mid b \mid x \mid + \mid \texttt{call} \mid \texttt{fix} \mid \texttt{if} \\
 & \mid & \{e\} \mid \lambda x.e \mid \texttt{let } x = e \texttt{ in } e \\
n & := & 0, 1, 2... \\
b & := & \texttt{true} \mid \texttt{false} \\
x & := & \textit{variables...}
\end{array}
$$

Figure 1: $\lambda_\circ$ Syntax

$$
\begin{array}{rcl}
s & := & \overrightarrow{v} \\
v & := & n \mid b \mid \{e\} \\
n & := & 0, 1, 2... \\
b & := & \texttt{true} \mid \texttt{false} \\
m & := & \langle s \mid e \rangle
\end{array}
$$

Figure 2: $\lambda_\circ$ Value Stacks & Machine States

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 ϵ 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

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

Figure 3: Machine Transition Rules

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 λ-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 $\twoheadrightarrow$ be the reflexive and transitive closure of $\mapsto$. We would like $\twoheadrightarrow$ 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 ϵ. This gives our first result.

Proposition 1. $\mapsto$ *can get stuck.*

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

$$
\begin{aligned}
&\langle 3\ \{\{4\}\}\ \{2\}\ \texttt{false} \mid \texttt{if}\ \texttt{call}\ +\rangle \\
&\mapsto \langle 3\ \{\{4\}\} \mid \texttt{call}\ +\rangle \\
&\mapsto \langle 3\ \{4\} \mid +\rangle
\end{aligned}
$$

□

2.3 EXAMPLES

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:

... let $Dup = \lambda x.x\ x$ in e ...
... let $Swap = \lambda x.\lambda y.x\ y$ in e ...
... let $Pop = \lambda x.$ in e ...

We may also write these terms inline:

$1\ 2\ (\lambda x.\lambda y.x\ y)\ 3 \twoheadrightarrow 1\ (\lambda y.2\ y)\ 3 \twoheadrightarrow 2\ 1\ 3$

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

... let $Compose = \lambda f.\lambda g.\{g\ \texttt{call}\ f\ \texttt{call}\}$ in e ...
... let $Partial = \lambda f.\lambda x.\{x\ f\ \texttt{call}\}$ in e ...

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 and are often as expressive and useful:

... let $Add1 = 1\ +$ in e ...
... let $Double = Dup\ +$ in e ...

3

TYPES FOR $\lambda_\circ$

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 τ consists of base types Int and Bool, standard type variables α, 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 $\overline{\alpha}$ on the left-hand side. We assume two disjoint infinite sets of type variables for α and $\overline{\alpha}$. All expressions have a function type f in $\lambda_\circ$, so our type schemes σ need only quantify over function types.

$$
\begin{aligned}
\tau &:= \mathsf{Int} \mid \mathsf{Bool} \mid \alpha \mid \mathsf{f} \\
\mathsf{i}, \mathsf{o} &:= \overrightarrow{\tau} \mid \overline{\alpha}\ \overrightarrow{\tau} \\
\mathsf{f} &:= \mathsf{i} \to \mathsf{o} \\
\sigma &:= \forall \overrightarrow{\alpha}.\mathsf{f}
\end{aligned}
$$

Figure 4: Types for $\lambda_\circ$

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 $\mathsf{i}_1 \cdot \mathsf{i}_2$. Sequence type variables are substituted with sequences of types i rather than individual types τ, 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 $\mathtt{ftv}(\mathsf{f})$ for the set of both individual and sequence type variables that occur free in function types f:

$$
\begin{aligned}
\mathtt{ftv}(\mathsf{Int}) = \mathtt{ftv}(\mathsf{Bool}) &= \emptyset \\
\mathtt{ftv}(\alpha) &= \{\alpha\} \\
\mathtt{ftv}(\overline{\alpha}) &= \{\overline{\alpha}\} \\
\mathtt{ftv}(\mathsf{i} \to \mathsf{o}) &= \mathtt{ftv}(\mathsf{i}) \cup \mathtt{ftv}(\mathsf{o}) \\
\mathtt{ftv}(\tau_1 \ldots \tau_n) &= \mathtt{ftv}(\tau_1) \cup \ldots \cup \mathtt{ftv}(\tau_n)
\end{aligned}
$$

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

$$
\mathtt{ftv}(\forall \overrightarrow{\alpha}.\mathsf{f}) = \mathtt{ftv}(\mathsf{f}) - \overrightarrow{\alpha}
$$

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 τ, whereas sequence type variables are instantiated with a finite sequence of types i. We denote instantiation by $\sigma_2 \sqsubseteq \sigma_1$ with σ_1 more specific than σ_2:

$$\frac{S = [\alpha_i \mapsto \tau_i, ..., \overline{\alpha}_j \mapsto i_j, ...] \qquad \{\alpha_i, ..., \overline{\alpha}_j, ...\} \subset \overrightarrow{\alpha} \qquad \sigma_1 = S(\sigma_2)}{\forall \overrightarrow{\alpha}.\sigma_2 \sqsubseteq \sigma_1}$$

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 λ-based languages.

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

$$\mathtt{ftv}(x_1 : \sigma_1, ..., x_n : \sigma_n) = \mathtt{ftv}(\sigma_1) \cup \ ... \ \cup \mathtt{ftv}(\sigma_n)$$

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

$$\widehat{\Gamma}(f) = \forall \overrightarrow{\alpha}.f \quad \text{where } \overrightarrow{\alpha} = \mathtt{ftv}(f) - \mathtt{ftv}(\Gamma)$$

3.3 SUBSTITUTIONS

Substitutions S (sometimes denoted by R and Φ) are finite mappings from individual type variables to individual type τ, 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 $\lambda_\circ$, 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:

$$S(\forall \overrightarrow{\alpha}.f) = \forall \overrightarrow{\alpha}.(S - \overrightarrow{\alpha})(f)$$

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

$$S \circ S' = S \cup [\alpha \mapsto S(i), \ \forall(\alpha, i) \in S' \text{ such that } \alpha \notin \mathtt{Dom}(S)]$$

$$\textsc{Empty}\ \frac{}{\Gamma \vdash \epsilon : i \to i} \qquad \textsc{Num}\ \frac{\Gamma \vdash e : i \to o}{\Gamma \vdash e\ n : i \to o\ \mathsf{Int}} \qquad \textsc{Bool}\ \frac{\Gamma \vdash e : i \to o}{\Gamma \vdash e\ b : i \to o\ \mathsf{Bool}}$$

$$\textsc{Add}\ \frac{\Gamma \vdash e : i \to o\ \mathsf{Int}\ \mathsf{Int}}{\Gamma \vdash e\ + : i \to o\ \mathsf{Int}} \qquad \textsc{If}\ \frac{\Gamma \vdash e : i \to o\ \tau\ \tau\ \mathsf{Bool}}{\Gamma \vdash e\ \mathtt{if} : i \to o\ \tau} \qquad \textsc{Var}\ \frac{x : \sigma \in \Gamma \qquad \sigma \sqsubseteq i_2 \to o_2 \qquad \Gamma \vdash e : i_1 \to o_1 \qquad o_1 = i_2}{\Gamma \vdash e\ x : i_1 \to o_2}$$

$$\textsc{Lam}\ \frac{\Gamma \vdash e_1 : i_1 \to o_1 \qquad \Gamma; x : \forall \overline{\alpha}.\overline{\alpha} \to \overline{\alpha}\ \tau \vdash e_2 : i_2 \to o_2 \qquad o_1 = i_2\ \tau}{\Gamma \vdash e_1\ \lambda x.e_2 : i_1 \to o_2}$$

$$\textsc{Block}\ \frac{\Gamma \vdash e_1 : i_1 \to o_1 \qquad \Gamma \vdash e_2 : i_2 \to o_2}{\Gamma \vdash e_1\ \{e_2\} : i_1 \to o_1\ (i_2 \to o_2)} \qquad \textsc{Let}\ \frac{\Gamma \vdash e_1 : f \qquad \Gamma; x : \widehat{\Gamma}(f) \vdash e_2 : i_2 \to o_2 \qquad \Gamma \vdash e_3 : i_1 \to o_1 \qquad o_1 = i_2}{\Gamma \vdash e_3\ \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : i_1 \to o_2}$$

$$\textsc{Call}\ \frac{\Gamma \vdash e : i_1 \to o_1\ (i_2 \to o_2) \qquad o_1 = i_2}{\Gamma \vdash e\ \mathtt{call} : i_1 \to o_2} \qquad \textsc{Fix}\ \frac{\Gamma \vdash e : i_1 \to o_1\ (i_2\ (i_2 \to o_2) \to o_2) \qquad o_1 = i_2}{\Gamma \vdash e\ \mathtt{fix} : i_1 \to o_2}$$

$$\textsc{Stack}\ \frac{\Gamma \vdash s : o \qquad \Gamma \vdash v : \tau}{\Gamma \vdash s\ v : o\ \tau} \qquad \textsc{VNum}\ \frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \textsc{VBool}\ \frac{}{\Gamma \vdash b : \mathsf{Bool}} \qquad \textsc{VBlock}\ \frac{\Gamma \vdash e : f}{\Gamma \vdash \{e\} : f}$$

$$\textsc{Machine}\ \frac{\Gamma \vdash s : i \qquad \Gamma \vdash e : i \to o}{\Gamma \vdash \langle s \mid e \rangle : o}$$

Figure 5: Type System for $\lambda_\circ$

3.4 TYPE SYSTEM

The syntax-directed typing rules for $\lambda_\circ$ are given in Fig. 5. The rules define the *typing judgment* $\Gamma \vdash e : f$, read as "in type environment Γ, 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 ϵ, but the type rule for ϵ 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

and output sequences in the type of ϵ, 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 $\lambda_\circ$ 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, **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 `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* $\Gamma \vdash e_1 : i_1 \to o_1$ *and* $\Gamma \vdash e_2 : i_2 \to o_2$ *and* $o_1 = i_2$, *then* $\Gamma \vdash e_1 \cdot e_2 : i_1 \to o_2$.

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

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

Proof. By structural induction on e_2. In the base case, $e_2 = \epsilon$, and so for any i' such that $\Gamma \vdash e_1 : i \to i'$, we can derive $\Gamma \vdash \epsilon : i' \to i'$. In the inductive cases, we have $e_2 = e_2'\ w$ for the word w corresponding to the particular case. Applying the inductive hypothesis, there exists some i' such that $\Gamma \vdash e_1 : i \to i'$ and $\Gamma \vdash e_2' : i' \to o'$, with $\Gamma \vdash w : o' \to o$. We then apply the type rule that gives us $\Gamma \vdash e_2'\ w : i' \to o$ to complete the case. □

Lemma 4. (Substitution) *If* $\Gamma; x : \forall\alpha_1 ... \alpha_n.f \vdash e_1 : f'$ *and* $x \notin \Gamma$ *and* $\Gamma \vdash e_2 : f$ *and* $\{\alpha_1, ..., \alpha_n\} \cap ftv(\Gamma) = \emptyset$, *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* $\langle s \mid e \rangle \mapsto \langle s' \mid e' \rangle$, *then* $\Gamma \vdash \langle s' \mid e' \rangle : o$.

Proof. We proceed by case analysis on the transition rule $\langle s \mid e \rangle \mapsto \langle s' \mid e' \rangle$. 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 $\langle s \mid n\ e \rangle \mapsto \langle s\ n \mid e \rangle$. We have $\Gamma \vdash s : i$ and $\Gamma \vdash n\ e : i \to o$ by **Machine**. By Lemma 3 and **Num** we have $\Gamma \vdash n : i \to i\ \mathsf{Int}$ and $\Gamma \vdash e : i\ \mathsf{Int} \to o$. Then by **VNum** we have $\Gamma \vdash s\ n : i\ \mathsf{Int}$, and by **Machine** we have $\Gamma \vdash \langle s\ n \mid e \rangle : o$ as required.

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

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

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

CASE $\langle s\ v_2\ v_1\ \mathtt{true} \mid \mathtt{if}\ e \rangle \mapsto \langle s\ v_1 \mid e \rangle$. By the type rules we have $\Gamma \vdash s\ v_2\ v_1\ \mathtt{true} : i\ \tau\ \tau\ \mathsf{Bool}$ and $\Gamma \vdash \mathtt{if}\ e : i\ \tau\ \tau\ \mathsf{Bool} \to o$. 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 $\langle s\ v_2\ v_1\ \mathtt{false} \mid \mathtt{if}\ e \rangle \mapsto \langle s\ v_2 \mid e \rangle$. Similar to the case for `if`-expressions on a `true` condition, with v_2 in place of v_1 in the final line.

CASE $\langle s\ \{e_1\} \mid \mathtt{call}\ e_2 \rangle \mapsto \langle s \mid e_1 \cdot e_2 \rangle$. By the type rules we have $\Gamma \vdash s\ \{e_1\} : i\ (i \to i')$ and $\Gamma \vdash \mathtt{call}\ e_2 : i\ (i \to i') \to o$. 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.

CASE $\langle s\ \{e_1\} \mid \texttt{fix}\ e_2\rangle \mapsto \langle s\ \{e_1\ \texttt{fix}\} \mid e_1 \cdot e_2\rangle$. By the type rules we have $\Gamma \vdash s\ \{e_1\} : i\ (i\ (i \to i') \to i') \to o$. By **Stack** and **VBlock**, we have $\Gamma \vdash s : i$ and $\Gamma \vdash e_1 : i\ (i \to i') \to i'$. By Lemma 3 and **Fix** we have $\Gamma \vdash e_2 : i' \to o$. By Lemma 2 we have $\Gamma \vdash e_1 \cdot e_2 : i\ (i \to i') \to o$. Next, by **Block**, **Fix**, and **VBlock** we have $\Gamma \vdash \{\{e_1\}\ \texttt{fix}\} : i\ (i \to i') \to i'$. Hence by **Stack** we have $\Gamma \vdash s\ \{\{e_1\}\ \texttt{fix}\} : i\ (i \to i')$, and by **Machine** we have $\Gamma \vdash \langle s\ \{\{e_1\}\ \texttt{fix}\} \mid e_1 \cdot e_2\rangle : o$ as required.

CASE $\langle s\ v \mid (\lambda x.e_1)\ e_2\rangle \mapsto \langle s \mid [v/x]e_1 \cdot e_2\rangle$. By the type rules we have $\Gamma \vdash s\ v : i\ \tau$ and $\Gamma \vdash (\lambda x.e_1)\ e_2 : i\ \tau \to o$. By Lemma 3 and **Lam** we have $\Gamma; x : \forall\overline{\alpha}.\overline{\alpha} \to \overline{\alpha}\ \tau \vdash e_1 : i \to i'$ where $\overline{\alpha} \notin \texttt{ftv}(\Gamma)$, and $\Gamma \vdash e_2 : i' \to o$. By **Stack** we have $\Gamma \vdash v : \tau$ and $\Gamma \vdash s : i$, and thus by Lemma 4 we have $\Gamma \vdash [v/x]e_1 : i \to i'$. Hence by Lemma 2 we have $\Gamma \vdash [v/x]e_1 \cdot e_2 : i \to o$, and by **Machine** we have $\Gamma \vdash \langle s \mid [v/x]e_1 \cdot e_2\rangle : o$ as required.

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

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

Proof. By induction on the typing derivation of e, and case analysis on the last rule used in the derivation.

CASE **Empty**. Then $e = \epsilon$ and we have our result.

CASE **Num**. Then $e = e'\ n$. By the inductive hypothesis, we have either that $\langle s \mid e'\rangle$ can take a step, or $e' = \epsilon$. Suppose $\langle s \mid e'\rangle \mapsto \langle s' \mid e''\rangle$; then we have $\langle s \mid e'\ n\rangle \mapsto \langle s \mid e''\ n\rangle$ by 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 $e' = \epsilon$. For the present rule, we then have $\langle s \mid n\rangle \mapsto \langle s\ n \mid \epsilon\rangle$, as required.

CASE **Bool,Block**. Similar to the case for **Num**.

CASE **Var**. Vacuous: variables are not well-typed when $\Gamma = \emptyset$.

CASE **Add**. Then $e = +$. For $\emptyset \vdash \langle s \mid +\rangle$ to be well-typed, s must be of the form $s'\ n_2\ n_1$. Hence we have $\langle s'\ n_2\ n_1 \mid +\rangle \mapsto \langle s'\ (n_2 + n_1) \mid \epsilon\rangle$.

CASE **If**. Then $e = \texttt{if}$. For $\emptyset \vdash \langle s \mid \texttt{if} \rangle$ to be well-typed, s must be of the form $s'\ v_2\ v_1\ \texttt{true}$ or $s'\ v_2\ v_1\ \texttt{false}$. Hence we have either $\langle s'\ v_2\ v_1\ \texttt{true} \mid \texttt{if} \rangle \mapsto \langle s'\ v_1 \mid \epsilon \rangle$ or $\langle s'\ v_2\ v_1\ \texttt{false} \mid \texttt{if} \rangle \mapsto \langle s'\ v_2 \mid \epsilon \rangle$.

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

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

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

□

Finally, we get the desired result.

Theorem 7. (Type Soundness) *For closed e, if $\emptyset \vdash e : \epsilon \to o$ and $\langle \epsilon \mid e \rangle \twoheadrightarrow \langle s \mid e' \rangle$, then either $e' = \epsilon$ 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 $\langle \epsilon \mid e \rangle \twoheadrightarrow \langle s \mid e' \rangle$, 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 $e' = \epsilon$ and $\emptyset \vdash s : o$. □

4

TYPE INFERENCE FOR $\lambda_\circ$

4.1 INFERENCE 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 Φ if the sequences can be unified. We also make use of an auxiliary function $\mathtt{inst}(\sigma, V)$:

$$\mathtt{inst}(\forall(\alpha_i, ..., \overline{\alpha}_j, ...).f, V) = ([\alpha_i \mapsto \beta_i, ..., \overline{\alpha}_j \mapsto \overline{\beta}_j, ...]f, V - \{\beta_i, ..., \overline{\beta}_j, ...\})$$
$$\text{where } \beta_i, ..., \overline{\beta}_j, ... \in V$$

That is, `inst` substitutes fresh type variables for the quantified type variables in σ 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 τ of $\lambda_\circ$, and unification of type sequences i 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. $\mathtt{ftv}(\mathtt{Unify}(i, i')) \subseteq \mathtt{ftv}(i) \cup \mathtt{ftv}(i')$
2. $\mathtt{Unify}(i, i') = \Phi$ implies that Φ is the most general unifier of i and i'

Algorithm 1 Inference for $\lambda_\circ$

function INFER(Γ, e, V) $\quad \triangleright$ `Infer` : $\langle \Gamma, e, V\rangle \to \langle S, f, V\rangle$

if e is ϵ **then**
$\langle \emptyset, \overline{\alpha} \to \overline{\alpha}, V - \{\overline{\alpha}\}\rangle$ where $\overline{\alpha} \in V$
else if e is e' n **then**
$\langle S, i \to o, V'\rangle \leftarrow$ `Infer`(Γ, e', V)
$\langle S, i \to o\ \mathsf{Int}, V'\rangle$
else if e is e' b **then**
$\langle S, i \to o, V'\rangle \leftarrow$ `Infer`(Γ, e', V)
$\langle S, i \to o\ \mathsf{Bool}, V'\rangle$
else if e is e' $+$ **then**
$\langle S, i \to o, V'\rangle \leftarrow$ `Infer`(Γ, e', V)
$\Phi \leftarrow$ `Unify`$(o, \overline{\alpha}\ \mathsf{Int}\ \mathsf{Int})$ where $\overline{\alpha} \in V'$
$\langle \Phi \circ S, \Phi(i \to \overline{\alpha}\ \mathsf{Int}), V' - \{\overline{\alpha}\}\rangle$
else if e is e' x and $x \in \Gamma$ **then**
$\langle i_2 \to o_2, V'\rangle \leftarrow$ `inst`$(\Gamma(x), V)$
$\langle S, i_1 \to o_1, V''\rangle \leftarrow$ `Infer`(Γ, e', V')
$\Phi \leftarrow$ `Unify`$(o_1, S(i_2))$
$\langle \Phi \circ S, \Phi(i_1 \to S(o_2)), V''\rangle$
else if e is e' `if` **then**
$\langle S, i \to o, V'\rangle \leftarrow$ `Infer`(Γ, e', V)
$\Phi \leftarrow$ `Unify`$(o, \overline{\alpha}\ \beta\ \beta\ \mathsf{Bool})$ where $\overline{\alpha}, \beta \in V'$
$\langle \Phi \circ S, \Phi(i \to \overline{\alpha}\ \beta), V' - \{\overline{\alpha}, \beta\}\rangle$
else if e is e_1 $\{e_2\}$ **then**
$\langle S_1, i_1 \to o_1, V'\rangle \leftarrow$ `Infer`(Γ, e_1, V)
$\langle S_2, i_2 \to o_2, V''\rangle \leftarrow$ `Infer`$(S_1(\Gamma), e_2, V')$
$\langle S_2 \circ S_1, (S_2(i_1) \to S_2(o_1)\ (i_2 \to o_2)), V''\rangle$
else if e is e' `call` **then**
$\langle S, i \to o, V'\rangle \leftarrow$ `Infer`(Γ, e', V)
$\Phi \leftarrow$ `Unify`$(o, \overline{\alpha}\ (\overline{\alpha} \to \overline{\beta}))$ where $\overline{\alpha}, \overline{\beta} \in V'$
$\langle \Phi \circ S, \Phi(i \to \overline{\beta}), V' - \{\overline{\alpha}, \overline{\beta}\}\rangle$
else if e is e' `fix` **then**
$\langle S, i \to o, V'\rangle \leftarrow$ `Infer`(Γ, e', V)
$\Phi \leftarrow$ `Unify`$(o, \overline{\alpha}\ (\overline{\alpha}\ (\overline{\alpha} \to \overline{\beta}) \to \overline{\beta}))$ where $\overline{\alpha}, \overline{\beta} \in V'$
$\langle \Phi \circ S, \Phi(i \to \overline{\beta}), V' - \{\overline{\alpha}, \overline{\beta}\}\rangle$
else if e is e_1 $\lambda x.e_2$ **then**
$\langle S_1, i_1 \to o_1, V'\rangle \leftarrow$ `Infer`(Γ, e_1, V)
$\langle S_2, i_2 \to o_2, V''\rangle \leftarrow$ `Infer`$(S_1(\Gamma); x : \forall\overline{\alpha}.\overline{\alpha} \to \overline{\alpha}\ \beta, e_2, V' - \{\beta\})$ where $\beta \in V'$
$\Phi \leftarrow$ `Unify`$(S_2(o_1), i_2\ S_2(\beta))$
$\langle \Phi \circ S_2 \circ S_1, \Phi(S_2(i_1) \to o_2), V''\rangle$
else if e is e_1 `let` $x = e_2$ `in` e_3 **then**
$\langle S_1, i_1 \to o_1, V'\rangle \leftarrow$ `Infer`(Γ, e_1, V)
$\langle S_2, f_2, V''\rangle \leftarrow$ `Infer`$(S_1(\Gamma), e_2, V')$
$\langle S_3, i_3 \to o_3, V'''\rangle \leftarrow$ `Infer`$(S_2(S_1(\Gamma)); x : \widehat{S_2(S_1(\Gamma))}(f_2), e_3, V'')$
$\Phi \leftarrow$ `Unify`$(S_3(S_2(o_1)), i_3)$
$\langle \Phi \circ S_3 \circ S_2 \circ S_1, \Phi(S_3(S_2(i_1)) \to o_3), V'''\rangle$
end if
end function

Algorithm 2 Unification for $\lambda_\circ$ Types

function UNIFY(i_1, i_2) ▷ $\texttt{Unify} : \langle i, i\rangle \to \Phi$
  if i_1, i_2 is ϵ, ϵ **then** []
  else if i_1 is $\overline{\alpha}$ and i_2 is $\overline{\alpha}$ **then** []
  else if i_1 is $\overline{\alpha}$ and $\overline{\alpha} \notin \texttt{ftv}(i_2)$ **then** $[\overline{\alpha} \mapsto i_2]$
  else if i_2 is $\overline{\alpha}$ and $\overline{\alpha} \notin \texttt{ftv}(i_1)$ **then** $[\overline{\alpha} \mapsto i_1]$
  else if i_1 is $i_1'\ \tau_1$ and i_2 is $i_2'\ \tau_2$ **then**
    $\Phi_1 \leftarrow \texttt{UnifyInd}(\tau_1, \tau_2)$
    $\Phi_2 \leftarrow \texttt{Unify}(\Phi_1 i_1', \Phi_1 i_2')$
    $\Phi_2 \circ \Phi_1$
  end if
end function

function UNIFYIND(τ_1, τ_2) ▷ $\texttt{UnifyInd} : \langle \tau, \tau\rangle \to \Phi$
  if τ_1, τ_2 is Int, Int **then** []
  else if τ_1, τ_2 is Bool, Bool **then** []
  else if τ_1 is α and $\alpha \notin \texttt{ftv}(\tau_2)$ **then** $[\alpha \mapsto \tau_2]$
  else if τ_2 is α and $\alpha \notin \texttt{ftv}(\tau_1)$ **then** $[\alpha \mapsto \tau_1]$
  else if τ_1, τ_2 is $i_1 \to o_1, i_2 \to o_2$ **then**
    $\Phi_1 \leftarrow \texttt{Unify}(i_1, i_2)$
    $\Phi_2 \leftarrow \texttt{Unify}(\Phi_1 o_1, \Phi_1 o_2)$
    $\Phi_2 \circ \Phi_1$
  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 $\lambda_\circ$ with mostly cosmetic changes. □

Theorem 9. (Inference is Correct) *Let* e *be an expression,* Γ *a type environment, and* V *a set of fresh type variables. If* $\texttt{Infer}(\Gamma, e, V) = (S, f, V')$*, 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 $e = \epsilon$. We have $\texttt{Infer}(\Gamma, \epsilon, V) = (\emptyset, \overline{\alpha} \to \overline{\alpha}, V')$. By **Empty** we can derive $\Gamma \vdash \epsilon : \overline{\alpha} \to \overline{\alpha}$, and we have $\emptyset(\Gamma) = \Gamma$ and $\emptyset(\overline{\alpha} \to \overline{\alpha}) = \overline{\alpha} \to \overline{\alpha}$ as required.

CASE $e = e'\ n$. We apply the induction hypothesis to the recursive call of Infer, yielding $S(\Gamma) \vdash e' : i \to o$. By **Num**, we then have $S(\Gamma) \vdash e'\ n : i \to o$ Int, which is the type returned by the algorithm.

CASE $e = e'\ \mathrm{b}$. Similar to the case for numbers.

CASE $e = e_1\ \{e_2\}$. We apply the induction hypothesis to the recursive calls of `Infer`, which yields

$$S_1(\Gamma) \vdash e_1 : i_1 \to o_1$$
$$S_2(S_1(\Gamma)) \vdash e_2 : i_2 \to o_2$$

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

CASE $e = e'\ +$. Applying the induction hypothesis to the recursive call of `Infer` yields $S(\Gamma) \vdash e' : i \to o$. For the algorithm to succeed, we must have $\mathtt{Unify}(o, \overline{\alpha}\ \mathsf{Int}\ \mathsf{Int}) = \Phi$. Hence, by Lemma 8 we have $\Phi(S(\Gamma)) \vdash e' : \Phi(i \to o)$. Since Φ is a most-general unifier [5], we have $\Phi(S(\Gamma)) \vdash e' : \Phi(i \to \overline{\alpha}\ \mathsf{Int}\ \mathsf{Int})$. By **Add** we then have $\Phi(S(\Gamma)) \vdash e'\ + : \Phi(i \to \overline{\alpha}\ \mathsf{Int})$, which is the type returned by the algorithm.

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

CASE $e = e'\ x$. Applying the induction hypothesis to the recursive call of `Infer` yields $S(\Gamma) \vdash e' : i_1 \to o_1$. We then have $\mathtt{inst}(S(\Gamma)(x), V') = (i_2 \to o_2, V'')$. By definition of `inst`, we have $S(\Gamma)(x) \sqsubseteq i_2 \to o_2$. For the algorithm to succeed, we must have $\mathtt{Unify}(o_1, i_2) = \Phi$. By Lemma 8 we have $\Phi(S(\Gamma)) \vdash e' : \Phi(i_1 \to o_1)$. Since $\Phi(o_1) = \Phi(i_2)$, we have by **(Var)** $\Phi(S(\Gamma)) \vdash e'\ x : \Phi(i_1 \to o_2)$, 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 `Infer` yields

$$S_1(\Gamma) \vdash e_1 : i_1 \to o_1$$
$$S_2(S_1(\Gamma)); x : S_2(\forall\overline{\alpha}.\overline{\alpha} \to \overline{\alpha}\ \beta) \vdash e_2 : i_2 \to o_2$$

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

$$\Phi(S_2(S_1(\Gamma))) \vdash e_1 : \Phi(S_2(i_1 \to o_1))$$
$$\Phi(S_2(S_1(\Gamma))); x : \Phi(S_2(\forall\overline{\alpha}.\overline{\alpha} \to \overline{\alpha}\ \beta)) \vdash e_2 : \Phi(i_2 \to o_2)$$

Since Φ is a most-general unifier, we have $\Phi(S_2(o_1)) = \Phi(i_2\ S_2(\beta))$, so by **Lam** we have $\Phi(S_2(S_1(\Gamma))) \vdash e_1\ \lambda x.e_2 : \Phi(S_2(i_1) \to o_2)$, and this is the type returned by the algorithm.

CASE $e = e_1\ \texttt{let}\ x = e_2\ \texttt{in}\ e_3$. Applying the induction hypothesis to the recursive calls of `Infer` yields

$$S_1(\Gamma) \vdash e_1 : i_1 \to o_1$$
$$S_2(S_1(\Gamma)) \vdash e_2 : f_2$$
$$S_3(S_2(S_1(\Gamma)); x : gen(S_3(S_2(S_1(\Gamma))), f_2) \vdash e_3 : i_3 \to o_3$$

By Lemma 8, we have

$$S_3(S_2(S_1(\Gamma))) \vdash e_1 : S_3(S_2(i_1 \to o_1))$$
$$S_3(S_2(S_1(\Gamma))) \vdash e_2 : S_3(i_2 \to o_2)$$

For the algorithm to succeed, we must also have $\texttt{Unify}(S_3(S_2(o_1)), i_3) = \Phi$. Then, as $\Phi(S_3(S_2(o_1))) = \Phi(i_3)$, we have by **Let** that $\Phi(S_3(S_2(S_1(\Gamma)))) \vdash e_1\ \texttt{let}\ x = e_2\ \texttt{in}\ e_3 : \Phi(S_3(S_2(i_1)) \to o_3)$, 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 Γ and Γ' be two type environments with the same domain. If $\Gamma'(x) \sqsubseteq \Gamma(x)$ 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 $\lambda_\circ$ with the definition of $\sqsubseteq$ from Section 3.4. □

Theorem 11. (Inference is Complete) *Let e be an expression, Γ a type environment, and V a set of fresh type variables such that $V \cap \texttt{ftv}(\Gamma) = \emptyset$. 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. $\texttt{Infer}(\Gamma, e, V) = (S, f, V')$ *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 $V' \cap \texttt{ftv}(f) = \emptyset$ and $V' \cap \texttt{ftv}(S(\Gamma)) = \emptyset$.

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

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

as required.

CASE $e = e'\ n$. The typing derivation ends with

$$\frac{S'(\Gamma) \vdash e' : i \to o}{S'(\Gamma) \vdash e\ n : i \to o\ \mathsf{Int}}$$

We apply the inductive hypothesis to e', Γ, V, $i \to o$ and S' to get

$$\begin{aligned}&\mathtt{Infer}(\Gamma, e', V) = (S, i' \to o', V')\\ &R(i' \to o') = i \to o\\ &(R \circ S) - V = S'\end{aligned}$$

The algorithm succeeds with type $i' \to o'\ \mathsf{Int}$ 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

$$\frac{S'(\Gamma) \vdash e_1 : i_1 \to o_1 \qquad S'(\Gamma) \vdash e_2 : i_2 \to o_2}{S'(\Gamma) \vdash e_1\ \{e_2\} : i_1 \to o_1\ (i_2 \to o_2)}$$

We apply the inductive hypothesis to e_1, Γ, V, $i_1 \to o_1$, and S' to get

$$\begin{aligned}&\mathtt{Infer}(\Gamma, e_1, V) = (S_1, i_1' \to o_1', V')\\ &R_1(i_1' \to o_1') = i_1 \to o_1\\ &(R_1 \circ S_1) - V = S'\end{aligned}$$

We then apply the inductive hypothesis to e_2, $S_1(\Gamma)$, V', $i_2 \to o_2$ and R_1.

$$\begin{aligned}&\mathtt{Infer}(S_1(\Gamma), e_2, V') = (S_2, i_2' \to o_2', V'')\\ &R_2(i_2' \to o_2') = i_2 \to o_2\\ &(R_2 \circ S_2) - V' = R_1\end{aligned}$$

The remark at the beginning of the proof assures us this is possible.
Since $\mathtt{ftv}(V') \cap \mathtt{ftv}(i_1' \to o_1')$, we have that $R_1(i_1' \to o_1') = R_2(S_2(i_1' \to o_1'))$. As the algorithm succeeded for both of its recursive calls, it then yields the type $S_2(i_1') \to S_2(o_1')\ (i_2' \to o_2')$ and the substitution $S_2 \circ S_1$. We let $R = R_2$, and then we have

$$\begin{aligned}&R_2(S_2(i_1') \to S_2(o_1')\ (i_2' \to o_2')) = R_1(i_1') \to R_1(o_1')\ (R_2(i_2') \to R_2(o_2'))\\ &= i_1 \to o_1\ (i_2 \to o_2)\\ &(R_2 \circ S_2 \circ S_1) - V = (R_1 \circ S_1) - V = S'\end{aligned}$$

as required.

CASE $e = e' +$. The typing derivation ends with

$$\frac{S'(\Gamma) \vdash e' : \mathsf{i} \to \mathsf{o}\ \mathsf{Int}\ \mathsf{Int}}{S'(\Gamma) \vdash e' + : \mathsf{i} \to \mathsf{o}\ \mathsf{Int}}$$

We apply the inductive hypothesis to e', Γ, V, $\mathsf{i} \to \mathsf{o}\ \mathsf{Int}\ \mathsf{Int}$, and S' to get

$$\begin{aligned}
&\texttt{Infer}(\Gamma, e', V) = (S, \mathsf{i}' \to \mathsf{o}', V') \\
&R(\mathsf{i}' \to \mathsf{o}') = \mathsf{i} \to \mathsf{o}\ \mathsf{Int}\ \mathsf{Int} \\
&(R \circ S) - V = S'
\end{aligned}$$

Let $\Phi = [\overline{\alpha} \mapsto \mathsf{o}] \circ R$, where $\overline{\alpha} \in V'$. We then have

$$\begin{aligned}
&\Phi(\mathsf{o}') = R(\mathsf{o}') = \mathsf{o}\ \mathsf{Int}\ \mathsf{Int} \\
&\Phi(\overline{\alpha}\ \mathsf{Int}\ \mathsf{Int}) = \mathsf{o}\ \mathsf{Int}\ \mathsf{Int}
\end{aligned}$$

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

$$\begin{aligned}
&S_\Phi(\Phi'(\mathsf{i}' \to \overline{\alpha}\ \mathsf{Int})) = \Phi(\mathsf{i}' \to \overline{\alpha}\ \mathsf{Int}) = R(\mathsf{i}) \to \mathsf{o}\ \mathsf{Int} = \mathsf{i} \to \mathsf{o}\ \mathsf{Int} \\
&(S_\Phi \circ \Phi' \circ S) - V = (\Phi \circ S) - V = (R \circ S) - V = S'
\end{aligned}$$

as required.

CASE $e = e'$ `if` or $e = e'$ `call` or $e = e'$ `fix`. These are similar to the case for addition.

CASE $e = e'\ x$. The typing derivation ends with

$$\frac{x : \sigma \in S'(\Gamma) \qquad \sigma \sqsubseteq \mathsf{i}_2 \to \mathsf{o}_2 \qquad S'(\Gamma) \vdash e' : \mathsf{i}_1 \to \mathsf{o}_1 \qquad \mathsf{o}_1 = \mathsf{i}_2}{S'(\Gamma) \vdash e'\ x : \mathsf{i}_1 \to \mathsf{o}_2}$$

Applying the inductive hypothesis to e', Γ, V, $\mathsf{i}_1 \to \mathsf{o}_1$, and S', we get

$$\begin{aligned}
&\texttt{Infer}(\Gamma, e', V) = (S, \mathsf{i}_1' \to \mathsf{o}_1', V') \\
&R(\mathsf{i}_1' \to \mathsf{o}_1') = \mathsf{i}_1 \to \mathsf{o}_1 \\
&(R \circ S) - V = S'
\end{aligned}$$

Since $S'(\Gamma) \vdash e'\ x : \mathsf{i}_1 \to \mathsf{o}_2$, we know that $x \in S'(\Gamma)$ and $S'(\Gamma)(x) \sqsubseteq \mathsf{i}_2 \to \mathsf{o}_2$. Hence $x \in \Gamma$, and we have $\mathsf{i}_2' \to \mathsf{o}_2' = [\alpha_1 \mapsto \beta_1, ..., \alpha_n \mapsto \beta_n](\mathsf{i}_x \to \mathsf{o}_x)$ and $V'' = V' - \{\beta_1, ..., \beta_n\}$ according to the definition of `inst`, where $\Gamma(x) = \forall \alpha_1, ..., \alpha_n.\mathsf{i}_x \to \mathsf{o}_x$, where $\forall i \in \{1, ..., n\}$ we have $\alpha_i \in V''$ and thus the α_i are out of reach for S' and R.

Then we have $S'(\Gamma)(x) = \forall \alpha_1, ..., \alpha_n.S'(\mathsf{i}_x \to \mathsf{o}_x)$. Let Q be the substitution over the α_i such that $\mathsf{i}_2 \to \mathsf{o}_2 = Q(S'(\mathsf{i}_x \to \mathsf{o}_x))$, and let $R' = Q \circ S' \circ [\beta_1 \mapsto$

$\alpha_1, ..., \beta_n \mapsto \alpha_n]$. We then have $R'(i'_2 \to o'_2) = Q(S'(i_x \to o_x)) = i_2 \to o_2$. As $o_1 = i_2$, we have $R(o'_1) = R'(i'_2) = Q(S'(i'_x)) = Q(R(S(i_x))) = R(S(Q(i_x)))$ since the α_i 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 Φ for the two sequences, and there exists a substitution R_1 such that $R = R_1 \circ \Phi$. Hence, the algorithm succeeds, yielding type $\Phi(i'_1 \to S(Q(o_x)))$ and substitution $\Phi \circ S$. We then have

$$R_1(\Phi((i'_1 \to S(Q(o_x))))) = i_1 \to R(S(Q(o_x))) = i_1 \to o_2$$
$$(R_1 \circ \Phi \circ S) - V = (R \circ S) - V = S'$$

as required.

CASE $e = e_1\ \lambda x.e_2$. The typing derivation ends with

$$\frac{S'(\Gamma) \vdash e_1 : i_1 \to o_1 \qquad S'(\Gamma); x : \forall\overline{\alpha}.\overline{\alpha} \to \overline{\alpha}\ \tau \vdash e_2 : i_2 \to o_2 \qquad o_1 = i_2\ \tau}{S'(\Gamma) \vdash e_1\ \lambda x.e_2 : i_1 \to o_2}$$

Applying the induction hypothesis to e_1, Γ, V, $i_1 \to o_1$, and S', we get

$$\texttt{Infer}(\Gamma, e_1, V) = (S_1, i'_1 \to o'_1, V')$$
$$R_1(i'_1 \to o'_1) = i_1 \to o_1$$
$$(R_1 \circ S_1) - V = S'$$

Let the environment Γ_1 be $\Gamma; x : \forall\overline{\alpha}.\overline{\alpha} \to \overline{\alpha}\ \beta$ and the substitution S'_1 be $[\beta \mapsto \tau] \circ S'$, where β is taken from V'. We apply the induction hypothesis to e_2, $S_1(\Gamma_1)$, $V' - \{\beta\}$, $i_2 \to o_2$ and R_1 to get

$$\texttt{Infer}(S_1(\Gamma), e_2, V' - \{\beta\}) = (S_2, i'_2 \to o'_2, V'')$$
$$R_2(i'_2 \to o'_2) = i_2 \to o_2$$
$$(R_2 \circ S_2) - V = R_1$$

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 Φ 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 $\Phi(S_2(i'_1) \to o'_2)$ and substitution $\Phi \circ S_2 \circ S_1$. Then we have

$$R_3(\Phi(S_2(i'_1) \to o'_2)) = R_2(S_2(i'_1)) \to R_2(o'_2) = i_1 \to o_2$$
$$(R_3 \circ \Phi \circ S_2 \circ S_1) - V = (R_2 \circ S_2 \circ S_1) - V = (R_1 \circ S_1) - V = S'$$

as required.

CASE $e = e_1\ \texttt{let}\ x = e_2\ \texttt{in}\ e_3$. Then the typing derivation ends with

$$\frac{S'(\Gamma) \vdash e_2 : f \qquad S'(\Gamma); x : \widehat{S'(\Gamma)}(f) \vdash e_3 : i_2 \to o_2 \qquad S'(\Gamma) \vdash e_1 : i_1 \to o_1 \qquad o_1 = i_2}{S'(\Gamma) \vdash e_1\ \texttt{let}\ x = e_2\ \texttt{in}\ e_3 : i_1 \to o_2}$$

Applying the induction hypothesis to e_1, Γ, V, $i_1 \to o_1$, and S' yields

$$\begin{aligned}
&\texttt{Infer}(\Gamma, e_1, V) = (S_1, i_1' \to o_1', V') \\
&R_1(i_1' \to o_1') = i_1 \to o_1 \\
&(R_1 \circ S_1) - V = S'
\end{aligned}$$

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

$$\begin{aligned}
&\texttt{Infer}(S_1(\Gamma), e_2, V') = (S_2, f', V'') \\
&R_2(f') = f \\
&(R_2 \circ S_2) - V = R_1
\end{aligned}$$

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

$$\begin{aligned}
&\texttt{Infer}(S_2(S_1(\Gamma)); x : \widehat{S_2(S_1(\Gamma))}(f'), e_3, V'') = (S_3, i_2' \to o_2', V''') \\
&R_3(i_2' \to o_2') = i_2 \to o_2 \\
&(R_3 \circ S_3) - V = R_2
\end{aligned}$$

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 Φ 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 $\Phi(S_3(S_2(i_1')) \to o_2')$ and substitution $\Phi \circ S_3 \circ S_2 \circ S_1$. Then we have

$$\begin{aligned}
&R_4(\Phi(S_3(S_2(i_1')) \to o_2')) = R_3(S_3(S_2(i_1'))) \to R_3(o_2)' = i_1 \to o_2 \\
&(R_4 \circ \Phi \circ S_3 \circ S_2 \circ S_1) - V = (R_3 \circ S_3 \circ S_2 \circ S_1) - V = \ldots = S'
\end{aligned}$$

as required.

□

5 IMPLEMENTATION

A variant of $\lambda_\circ$ has been implemented as a `#lang` in Racket. The variant, named *wort*, is semantically similar to $\lambda_\circ$ 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 *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 https://github.com/robertkleffner/wort.

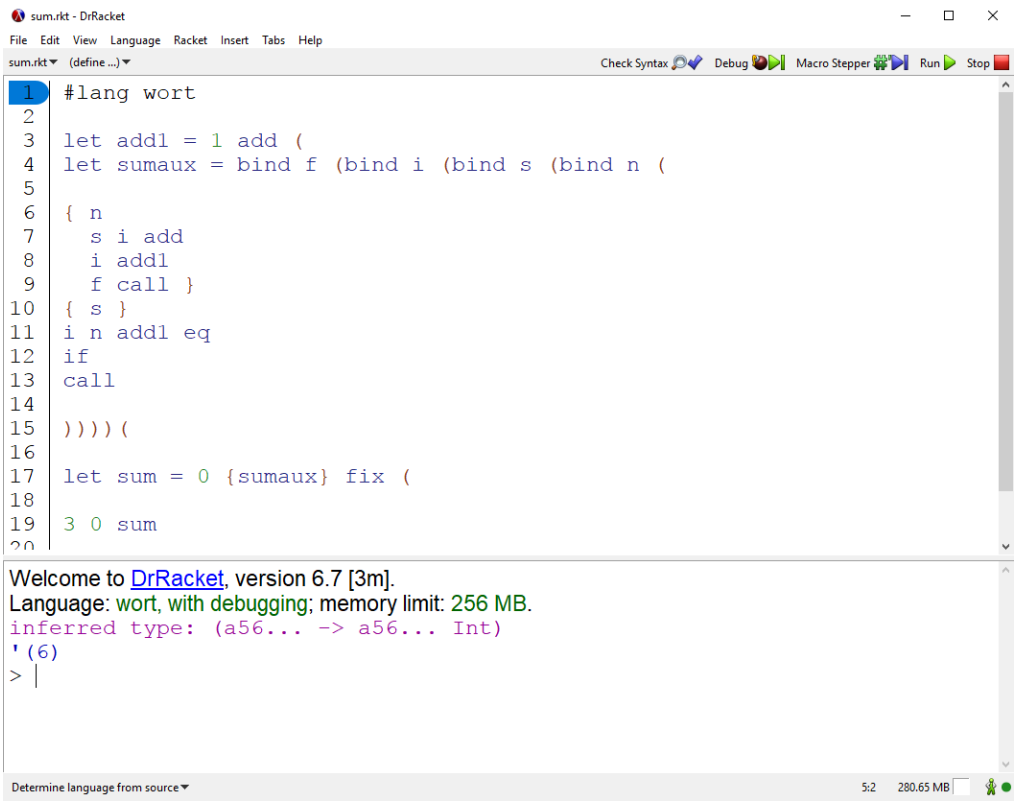

Figure 6: A *wort* program in DrRacket

6 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. $\lambda_\circ$ also has striking parallels with the call-by-push-value paradigm [7] which may warrant further investigation.

In the type syntax of $\lambda_\circ$, 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 $\lambda_\circ$ is somewhat tricky to formulate properly. The syntax, semantics, and type system of $\lambda_\circ$ provides a starting point for potential answers to these questions.

BIBLIOGRAPHY

[1] Christopher Diggins. *Simple Type Inference for Higher-Order Stack-Oriented Languages*. Tech. rep. 2008. DOI: 10.1.1.156.406.

[2] Christopher Diggins. *Typing Functional Stack-Based Languages*. Tech. rep. 2008. URL: https://www.researchgate.net/publication/228985001_Typing_Functional_Stack-Based_Languages.

[3] Sami Hangaslammi. *Concatenative Row-Polymorphic Programming in Haskell*. 2012. URL: https://github.com/leonidas/codeblog/blob/master/2012/2012-02-17-concatenative-haskell.md.

[4] Brent Kerby. *The Theory of Concatenative Combinators*. 2002. URL: http://tunes.org/{~}iepos/joy.html.

[5] Temur Kutsia. *Theorem Proving with Sequence Variables and Flexible Arity Symbols*. Ed. by Matthias Baaz and Andrei Voronkov. Vol. 2514. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. ISBN: 978-3-540-00010-5. DOI: 10.1007/3-540-36078-6.

[6] Xavier Leroy. *Polymorphic typing of an algorithmic language*. Research Report RR-1778. Projet FORMEL. INRIA, 1992. URL: https://hal.inria.fr/inria-00077018.

[7] Paul Blain Levy. "Call-by-push-value: Decomposing Call-by-value and Call-by-name." In: *Higher Order Symbol. Comput.* 19.4 (Dec. 2006), pp. 377–414. ISSN: 1388-3690. DOI: 10.1007/s10990-006-0480-6. URL: http://dx.doi.org/10.1007/s10990-006-0480-6.

[8] Sam Lindley, Conor McBride, and Craig McLaughlin. "Do Be Do Be Do." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, 2017, pp. 500–514. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009897. URL: http://doi.acm.org/10.1145/3009837.3009897.

[9] Robin Milner. "A theory of type polymorphism in programming." In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–375.

[10] Chris Okasaki. "Techniques for embedding postfix languages in Haskell." In: *Proceedings of the ACM SIGPLAN workshop on Haskell - Haskell '02* (2002), pp. 105–113. DOI: 10.1145/581690.581699.

[11] Sviatoslav Pestov, Daniel Ehrenberg, and Joe Groff. "Factor: A Dynamic Stack-based Programming Language." In: *SIGPLAN Not.* 45.12 (Oct. 2010), pp. 43–58. ISSN: 0362-1340. DOI: 10.1145/1899661.1869637.

[12] Jaanus Poial. "Algebraic Specification of Stack Effects for Forth Programs.pdf." In: *1990 FORML Conference Proceedings*. The Forth Interest Group, 1990, pp. 12–14.

[13] Jaanus Poial. "Stack effect calculus with typed wildcards , polymorphism and inheritance." In: *18th EuroForth Conference*. 2002, p. 38.

[14] Jaanus Poial. "Typing Tools for Typeless Stack Languages." In: *23rd EuroForth Conference*. 2006, pp. 40–46.

[15] Jon Purdy. *Why Concatenative Programming Matters*. 2012. URL: https://evincarofautumn.blogspot.com/2012/02/why-concatenative-programming-matters.html.

[16] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. "History of Programming languages—II." In: ed. by Thomas J. Bergin Jr. and Richard G. Gibson Jr. New York, NY, USA: ACM, 1996. Chap. The Evolution of Forth, pp. 625–670. ISBN: 0-201-89502-1. DOI: 10.1145/234286.1057832.

[17] Glenn C. Reid. *Thinking in PostScript*. Addison-Wesley, 1990.

[18] Ando Saabas and Tarmo Uustalu. "Compositional type systems for stack-based low-level languages." In: *Proceedings of the 12th Computing: The Australasian Theroy Symposium*. Vol. 51. 2006, pp. 27–39. ISBN: 1920682333.

[19] Bill Stoddart and Peter J. Knaggs. "Type inference in stack based languages." In: *Formal Aspects of Computing* 5.4 (1993), pp. 289–298. ISSN: 0934-5043. DOI: 10.1007/BF01212404.

[20] Manfred von Thun. "Joy : Forth's Functional Cousin." In: *EuroForth Conference*. 2001. URL: http://www.complang.tuwien.ac.at/anton/euroforth/ef01/thomas01a.pdf.

[21] A.K. Wright and M. Felleisen. "A Syntactic Approach to Type Soundness." In: *Inf. Comput.* 115.1 (Nov. 1994), pp. 38–94. ISSN: 0890-5401. DOI: 10.1006/inco.1994.1093. URL: http://dx.doi.org/10.1006/inco.1994.1093.