What is a Universal Higher-Order Programming Language?

Ramarao Kanneganti and Robert Cartwright *

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract. In this paper, we develop a theory of higher-order computability suitable for comparing the expressiveness of sequential, deterministic programming languages. The theory is based on the construction of a new universal domain $\mathbb{T}$ and corresponding universal language KL. The domain $\mathbb{T}$ is universal for *observably sequential* domains; KL can define all the computable elements of $\mathbb{T}$, including the elements corresponding to computable observably sequential functions. In addition, domain embeddings in $\mathbb{T}$ preserve the maximality of finite elements—preserving the termination behavior of programs over the embedded domains.

1 Background and Motivation

Classic recursion theory [7, 13, 18] asserts that all conventional programming languages are equally expressive because they can define all partial recursive functions over the natural numbers. This statement, however, is misleading because real programming languages support and enforce a more abstract view of data than bitstrings. In particular, most real programming languages support some form of *higher-order* data such as potentially infinite streams (input and output), lazy trees, and functions. In contrast to conventional data objects like numbers, characters, arrays, and lists, *higher-order* data objects do not have finite canonical representations. As a result, computations involving higher-order objects cannot be modeled as ordinary computations over the natural numbers. The integrity of higher-order data abstractions critically depends on the fact that programs obey constraints on how higher-order representations are manipulated. Since classic recursion theory ignores these constraints, it does not address the expressiveness of languages that manipulate higher-order data.

To assess the expressiveness of real languages, we need to develop a theory of computability that includes higher-order data. To be comprehensive, this theory must accommodate all of the data objects that occur in conventional languages, including ordinary finite data objects and higher-order data objects such as lazy trees and functions. In this paper, we present a theory of higher-order computability suitable for sequential, deterministic programming languages. The theory is based on the construction of a new universal domain $\mathbb{T}$ suitable for

* This research was partially supported by the National Science Foundation under the grant CCR-9122518.

embedding all "sequential"[2] domains. Roughly speaking, a Scott-domain (see Sect. 4) is "sequential" if its elements can be represented as lazy trees. The "sequential" domains correspond to "filiform concrete data structures" studied by Curien [9], which are a restriction of "concrete domains" originally introduced by Kahn and Plotkin [11].

We show that the new universal domain $\mathbb{T}$ has two important properties:

1. Every "sequential" domain D can be embedded by a "sequential" retraction as a subdomain D' of $\mathbb{T}$ such that the maximality of finite elements is preserved.
2. There is a simple metalanguage called KL that is *universal* for $\mathbb{T}$: every computable element of $\mathbb{T}$ and every computable "sequential" operation on $\mathbb{T}$ can be expressed in KL.

This two properties imply that KL is a universal "sequential" higher-order programming language. Every computable "sequential" function f over a "sequential" domain D can be defined in KL by first embedding D in $\mathbb{T}$. In addition, the termination behavior of f is preserved in $\mathbb{T}$ because embeddings in $\mathbb{T}$ preserve the finite maximality of elements. Finite maximal elements are the denotations of terminating computations.

2 Previous Work On Universal Languages

The issue of higher-order computation has previously been studied in two contexts: (i) the extension of classic recursion theory to computable functionals [10, 14] and (ii) *domain theory* [15, 17], the mathematical framework underlying denotational semantics. In the classic theory of computable functionals, functions are represented intensionally as strings of symbols. Since programs can inspect the representations of higher order data objects, this model of higher order computation does not capture the linguistic restrictions imposed by real programming languages.

Scott and Plotkin developed domain theory to address this issue. In domain theory, denotational definitions for programming languages are written in a metalanguage *LAMBDA* corresponding to a universal domain U (either $\mathsf{P}\omega$ [17] or T^ω [15]). In essence, a denotational definition of a programming language is a functional program written in *LAMBDA* that maps source program phrases into their meanings, which are data objects in the universal domain. The language *LAMBDA* appears to be a promising candidate for a universal, higher-order programming language for higher-order computation. Unfortunately, the model of computation underlying *LAMBDA* is too abstract to simulate the behavior

[2] The quotation marks indicate that we are using this term informally. Our usage does not exactly match the technical meaning of the term *sequential* in the literature [11]. *Sequential* functions do not provide a satisfactory basis for defining a universal language because the sequential functions and corresponding domains do not form a cartesian closed category [1]. *Sequential algorithms* do form a cartesian closed category but they are intensional.

of real programs faithfully. In particular, *LAMBDA* eliminates the distinction between terminating and non-terminating computations: the evaluation of an expression M in *LAMBDA* simply enumerates all finite approximations to the data object d denoted by M. This enumeration process cannot terminate because no approximation in U is maximal.

Cartwright and Demers [3] have developed a new formulation of domain theory that addresses the issue of program termination. They constructed a new universal domain ∇ (pronounced "Grad") and metalanguage *Cons* that distinguish terminating and non-terminating computations. In contrast to previous universal domains, ∇ includes finite maximal elements. Any Scott-domain (see Sect. 5.2) that is *effectively* presented can be effectively embedded inside ∇ such that the finiteness and maximality of elements are preserved. The evaluation of an expression M in *Cons* terminates iff M denotes a finite maximal element of ∇.

However, their theory has two liabilities. First, the syntax of the universal language *Cons* is not recursive; a programmer must prove that a program written in *Cons* is well-formed. Second, the continuous function space construction does not preserve maximality information. Given effective presentations of the domains A and B, it is impossible in general to construct an effective presentation for the domain $A \rightarrow_c B$ (the continuous functions from A to B).

3 Continuous Functions Are Too General

We can trace the liabilities of ∇ directly to two properties of the continuous function space construction. First, the continuous function space constructor $\rightarrow_c$ is contravariant in its first argument. As a result, the maximality information for a constructed space $A \rightarrow_c B$ cannot in general be computed from the maximality information and effective presentations [15, 17] of A and B. Second, the $\rightarrow_c$ constructor introduces infinitely descending chains of elements in domain constructions. Both of these properties make it difficult to construct a universal domain and programming language based on continuous functions. In the absence of maximality information for a domain D, we cannot preserve the termination behavior of programs over D when D is embedded in the universal domain. The presence of infinite descending chains prevents effective presentations from enumerating finite elements in topologically sorted order, complicating the task of embedding domains in a universal domain. In addition, it is difficult to devise metalanguages for the domains with infinite descending chains because computations climb the approximation ordering, but the inductive definition of the finite elements does not.

We can avoid the pathologies caused by the contravariance of the continuous function space construction by using a more restrictive formulation of functions than the *continuous* functions. We recently developed a subspace of the continuous functions called the observably sequential functions that encompasses all of the functions definable in practical *deterministic* programming languages [5, 6, 12]. The observably sequential functions exclude continuous functions that cannot be evaluated without multiple threads of control. In practice, *non-sequential* functions are unimportant in deterministic computation. We are not aware of any practical deterministic language that can define non-sequential functions like por, the "parallel-or" function.[3] Even pedagogic functional languages like Miranda and Haskell do not support this class of functions because it would encourage insidiously inefficient programming.

The rest of the paper is organized as follows. In Sect. 4, we introduce the class of domains, called OS-domains, suitable for sequential computation. In Sect. 5, we present a domain $\mathbb{T}$ that is universal for OS-domains. We show how to embed any OS-domain in $\mathbb{T}$ while preserving the maximality of finite elements. We also present a language KL for defining operations on the domain $\mathbb{T}$. In Sect. 6, we prove that KL is universal by showing that every computable element of the domain $\mathbb{T}$ is expressible in KL.

4 OS-Domains

In this section, we define the class of observably sequential domains (OS-domains); the construction is spelled out in greater detail in a previous report [12]. These domains have the structure required to support observably sequential computation.

A *Scott-domain* $(D, \sqsubseteq)$ is the ideal completion of a *Scott-basis* $(E, \leq)$, which is a partial order closed under least upper bounds for finite bounded subsets. The *finite elements* of the domain are the principal ideals generated by the elements of the Scott-basis E. As is customary, we will identify all Scott-domains that are order-isomorphic, eliminating the distinction between a Scott-basis and the corresponding finite elements.

OS-domains are Scott-domains with special structure: they can be generated from a subset of finite elements called the *prime basis*. The topological constraints governing OS-domains are formulated as constraints on the prime basis. These notions are formally defined as follows.

Notation: In a partial order $\langle S, \sqsubseteq \rangle$, if $a \sqsubseteq b$, we say that *a approximates b*. An element *a precedes b*, written as $a \prec b$, if $a \sqsubset b$ and there are no elements between a and b in the partial order.

Definition 4.1. (*Prime Basis*) A triple $(P, \leq, C)$ is a *prime basis* iff

[PO] $(P, \leq)$ is a countable partial order.
[UP] Each element of the partial order $(P, \leq)$ has at most one predecessor.
[F] each element of P is approximated by only a finite number of elements.
[C] $C = \{Q_1, Q_2, \ldots\}$ is a partitioning of P into non-empty *conflict* sets (or C-sets) $Q_1, Q_2, \ldots$ such that $p, q \in Q_i$ implies that p and q have the same set of predecessors.

[3] por(x, true) = true; por(true, y)= true; and por(false, false) = false.

A prime basis $(P, \leq, C)$ is *error-rich* iff each C-set $Q \in C$ has two designated elements error_1^Q, error_2^Q that are maximal in $(P, \leq)$. In addition, every C-set Q must contain at least one element other than error_1^Q, error_2^Q. ∎

An OS-domain is the "prime-closure" of an error-rich prime basis.

Definition 4.2. (*Prime-Domain, OS-domain*) A *prime-domain* is a partial order $(D, \sqsubseteq)$ generated from a prime basis $(P, \leq, C)$ where

1. the elements of D are *downward-closed* subsets of P that do not contain conflicting primes (distinct primes from the same C-set).
2. the relation $\sqsubseteq$ is the subset ordering $\subseteq$ on D.

The domain generated by the prime basis $(P, \leq, C)$ is denoted $\mathbb{D}(P, \leq, C)$. If the prime basis $(P, \leq, C)$ is error-rich, then the prime-domain $\mathbb{D}(P, \leq, C)$ is an OS-domain. ∎

Most domains used for the models of deterministic languages are prime-domains. The most notable exceptions are the domains generated by the continuous function construction. Consider the following simple examples.

1. The flat domain $\mathbb{B}_\perp$ of truth values $\{\mathsf{tt}, \mathsf{ff}\}$ is a prime-domain.
2. The flat domain $\mathbb{N}_\perp$ of natural numbers is a prime-domain; every number is a prime element.
3. The continuous function space $\mathbb{B}_\perp \rightarrow_c \mathbb{B}_\perp$ is not a prime-domain; the prime functions (the "one-step" functions) cannot be partitioned into C-sets.

5 A Universal Sequential Domain

To perform arbitrary sequential computations, we need to construct a universal domain U in which any OS-domain can be embedded. Thus, we can reduce sequential computation over an arbitrary domain D to sequential computation over U by embedding D in U.

Our universal domain $\mathbb{T}$ is a domain of lazy binary trees over the natural numbers and error elements. The domain $\mathbb{T}$ is defined by its prime basis as follows.

Definition 5.1. (*Universal Domain* $\mathbb{T}$) The universal domain $\mathbb{T}$ is $\mathbb{D}(P, \leq, C)$ where $\langle P, \leq, C\rangle$ is defined as the following:

1. The set of primes P is given by the inductive definition:
$$P = \mathsf{error}_1 \mid \mathsf{error}_2 \mid n \in \mathbb{N} \mid \langle \perp, \perp \rangle \mid \langle \perp, p \rangle \mid \langle p, \perp \rangle \qquad p \in P$$
2. The ordering relation $\leq$ on P is defined by the rules:
$$\begin{array}{rll} x \leq x & \text{for} & x \in \mathbb{N} \cup \{\mathsf{error}_1, \mathsf{error}_2\} \\ \langle a, b \rangle \leq \langle c, d \rangle & \text{iff} & [(a = \perp) \vee (a \leq c)] \text{ and } [(b = \perp) \vee (b \leq d)] \end{array}$$

3. The *conflict* sets (C-sets) are defined by the equivalence closure of the relation C' over P:

$$\begin{array}{rll} (0, \mathsf{error}_i),\ (0, \langle\bot,\bot\rangle) \in C' & & \\ (0, n) \in C' & \text{for} & n \in \mathbb{N} \\ (\langle a,\bot\rangle, \langle b,\bot\rangle),\ (\langle\bot,a\rangle, \langle\bot,b\rangle) \in C' & \text{for} & (a,b) \in C' \end{array}$$

∎

Every tree in the domain $\mathbb{T}$ corresponds to a distinct set of prime approximations. Each prime approximating the tree t identifies the value of a node in t. If two trees are inconsistent, there is a node where the trees have different values other than $\bot$.

Each C-set in the domain $\mathbb{T}$ identifies a *position* in the tree; the elements of the C-set specify different values for the node identified by the C-set. Hence two trees are inconsistent iff they are approximated by two different prime elements from the same C-set. We refer to C-sets by capital letters such as Q, R and S. A position Q in a tree is *open* iff the tree has $\bot$ at that position. We say that a tree A is *strictly below* a tree B at position Q, written $A \subset_Q B$, iff the tree A has a $\bot$ and B has a node value other than $\bot$ at position Q. Similarly, we say that a tree *a precedes* Q, written $a \prec Q$, iff a is is open at the position Q and in all the positions less than Q, a is defined, *i.e.*, it has non-$\bot$ values at those positions.

The universal OS-domain has the property that any OS-domain D can be embedded in $\mathbb{T}$. Moreover, there is an observably sequential function (OS-function) ρ_D over $\mathbb{T}$ such that $\rho_D(\mathbb{T}) = D'$ where D' is isomorphic to D. To formulate the class of embeddings we must rigorously define the OS-functions.

5.1 OS-Function Space Construction

The observably sequential functions from $\mathbb{T}$ into $\mathbb{T}$ are a subset of continuous functions, $\mathbb{T} \to_c \mathbb{T}$. These functions propagate any error element that they encounter during the evaluation of the input. More precisely, if a function encounters an error element error_i in its input while it is computing the node value for a position Q, it must output the value error_i at position Q of the output element. We formally define this class of functions in the following definitions.

Definition 5.2. (*Sequentiality Index*) For a continuous function $f : \mathbb{T} \to \mathbb{T}$, an open position R in a is a *sequentiality index* for the input a and an open position Q in $f(a)$, iff for any $x \supset a$ such that $f(x) \supset_Q f(a)$, $x \supset_R a$. ∎

If R is a sequentiality index at an input a and an output position Q, then the function f explores the input at position R to generate an output at the position Q. If the function f propagates errors, then it must generate the value error_i at position Q if it encounters the value error_i at R. A function f with such a behavior has a *sequential* strategy to evaluate its arguments. Moreover, the behavior of f over error elements determines that evaluation strategy. Since this information is a part of the graph of f, f is called an *observably* sequential function (OS-function).

Definition 5.3. (*Observably Sequential Function*) A continuous function $f : \mathbb{T} \rightarrow_c \mathbb{T}$ is *observably sequential* if it is:

sequential: for every pair (a, Q) where Q is an open position in $f(a)$ and Q is defined in $f(x)$ for some $x \supset a$, f has a sequentiality index; and

error-sensitive: if R is the sequentiality index of f for the input a and the output position Q, $\mathsf{error}_1^Q \in (f(a \cup \{\mathsf{error}_1^R\}))$ and $\mathsf{error}_2^Q \in f(a \cup \{\mathsf{error}_2^R\})$. ∎

Every OS-function has a unique evaluation strategy manifest in its graph [5]. If an OS-function f has two sequentiality indices R, R' for input a and open position Q in $f(a)$, it is easy to show that f is not continuous.[4] Since the sequentiality indices for an OS-function f are unique, it makes sense to introduce a function si_f, where $si_f(a, Q)$ is the sequentiality index of f for input a and output position Q.

The function space $\mathbb{T} \rightarrow_{os} \mathbb{T}$ is an OS-domain. Its prime basis $(P_\rightarrow, \leq_\rightarrow, C_\rightarrow)$ can be described as follows:

$P_\rightarrow$: The members of P can be divided into two disjoint subsets:

(a) **Output Primes:** An OS-function f in $\mathbb{T} \rightarrow_{os} \mathbb{T}$ is an output prime iff there exists a finite element a and a prime element p in $\mathbb{T}$ such that f is a minimal function satisfying the constraint $f(a) = p$ and for all $a' \subset a$, $f(a') \subset f(a)$. The function f has a C-set *label* $\langle a, Q, F' \rangle$ where Q is the position of p and $F' = \{g \in \mathbb{T} \rightarrow_{os} \mathbb{T} \,|\, g \subset f\}$.

(b) **Schedule Primes:** an OS-function f in $\mathbb{T} \rightarrow_{os} \mathbb{T}$ is a *schedule prime* iff there exists a finite element a and C-sets R, Q in $\mathbb{T}$ where $a \prec R$ such that f is a minimal function with the property $f(a \sqcup \mathsf{error}_i^R) = \mathsf{error}_i^Q$ and $f(a) \subset \mathsf{error}_1^Q$. The function f has the C-set *label* $\langle a, Q, F' \rangle$ where $F' = \{g \in \mathbb{T} \rightarrow_{os} \mathbb{T} \,|\, g \subset f\}$.

$\leq_\rightarrow$: The prime functions are ordered under usual pointwise ordering, *i.e.*, $f \leq g$ iff $\forall x \in \mathbb{T} : f(x) \leq g(x)$.

$C_\rightarrow$: The partitioning of $P_\rightarrow$ determined by the equivalence relation $\sim$ on $P_\rightarrow$ where

$$p \sim q \Leftrightarrow \text{ the C-set } \mathit{label} \text{ of } p = \text{ the C-set } \mathit{label} \text{ of } q.$$

It is easy to verify that $(P_\rightarrow, \leq_\rightarrow, C_\rightarrow)$ forms a prime basis. The two designated error elements of a C-set with label $\langle a, Q, F' \rangle$ are $F' \cup \{(a, \mathsf{error}_1^Q)\}$ and $F' \cup \{(a, \mathsf{error}_2^Q)\}$. We also need to verify that $\mathbb{D}(P_\rightarrow, \leq_\rightarrow, C_\rightarrow)$ is isomorphic to the domain $\mathbb{T} \rightarrow_{os} \mathbb{T}$. The proof breaks down into two parts. First, we can prove that any function in the domain $\mathbb{D}(P_\rightarrow, \leq_\rightarrow, C_\rightarrow)$ is an OS-function. Second, we can prove that any finite function in $\mathbb{T} \rightarrow_{os} \mathbb{T}$ is the least upper bound of its prime approximations in $P_\rightarrow$. Together, these two facts imply that $(P_\rightarrow, \leq_\rightarrow, C_\rightarrow)$ generates the OS-function domain $\mathbb{T} \rightarrow_{os} \mathbb{T}$.

[4] Otherwise, on the input extending a with error_1 at R and error_2 at R' the function has to generate the least upper bound of two inconsistent elements, error_1 and error_2, at position Q [6].

5.2 Embeddings that Preserve Maximality

The universal domain $\mathbb{T}$ is universal for the class of OS-domains. That is, any OS-domain can be embedded in $\mathbb{T}$ by an OS-function such that the maximality of the finite elements is preserved. In addition, any prime-domain can be embedded in $\mathbb{T}$ by enriching it with sufficient number of error elements. Since $\mathbb{T} \to_{os} \mathbb{T}$ is an OS-domain, it can be embedded in $\mathbb{T}$ by an OS-function. This embedding allows functions to be represented as trees.

A computation in a domain D is an enumeration producing finite approximations to the answer with progressively higher accuracy. A computation terminates if and only if the answer is finite maximal element. To preserve the behavior of computations over D, an embedding of D in $\mathbb{T}$ must map the finite elements in terminating computations to finite elements of $\mathbb{T}$ and their answers to finite maximal elements of $\mathbb{T}$.

To satisfy this property, embeddings must preserve more structure than the embeddings used in classical domain theory [15, 17] which do not preserve finite maximality. To describe the embedding process and make it *effective*, we need to define what we mean by an *effective presentation* of a domain. Since our domains are constructed from prime elements and our embedding must preserve finite maximality, our definition differs from others in the literature:

Definition 5.4. (*Effective Presentation*) An effective presentation $\mathcal{P}$ of the domain $\mathbb{D}(P, \leq, C)$ is a tuple $\langle \mathbf{E}, \mathbf{c}, \mathbf{pr}, \mathbf{arity}, \mathbf{size} \rangle$ such that:

1. $\mathbf{E} : P \to \mathbb{N}$ is an injective function that enumerates the prime elements of the domain excluding errors in topologically sorted order. That is, if an element a approximates b, a is enumerated before b. The error elements are not enumerated because their presence is implied by the fact that $\mathbb{D}(P, \leq, C)$ is an OS-domain. We use the notation p_i for q when $\mathbf{E}(q) = i$.
2. $\mathbf{c} : \mathit{range}(\mathbf{E}) \to \mathbb{N}$ is a function that maps the index of the prime element to its C-set code. That is, $\forall i, j > 0 : \mathbf{c}(i) = \mathbf{c}(j)$ iff $p_i, p_j \in Q$ for $Q \in C$. The range of the function $\mathbf{c}$ is the set of C-set numbers.
3. $\mathbf{pr}$ is a recursive function over $\mathbb{N}$ defined by $\mathbf{pr}(i) = j$ iff $p_i \prec p_j$.
4. $\mathbf{arity}$ is a recursive function that maps the index of a prime element to the number in $\mathbb{N} \cup \{\omega\}$ of C-sets above that element;[5] and
5. $\mathbf{size}$ is a recursive function that maps the index of a C-set to its cardinality in $\mathbb{N} \cup \{\omega\}$.

∎

All flat domains such as numbers, booleans, and non-lazy lists obviously have effective presentations. Moreover the domain constructions like cartesian products, disjoint unions, lifting, OS-function construction preserve the effective presentation. As a result, all the domains typically encountered in deterministic sequential computation can effectively be embedded in $\mathbb{T}$ while preserving the maximality of finite elements.

[5] A prime element may have ω C-sets above it.

In particular, the OS-function construction preserves maximality information. If two domains D and E have effective presentations, the corresponding OS-function space $D \rightarrow_{os} E$ has an effective presentation. The functions **arity** and **size** for the function space can be computed from the functions **arity** and **size** of the domains D and E. The details of the effective presentation of OS-functions is given in appendix I.

A domain D is embedded in $\mathbb{T}$ by a function f_d mapping its prime elements into $\mathbb{T}$. For each prime p, $f_d(p)$ is computed by inserting a subtree into the tree corresponding to predecessor of p. Since all the prime elements are enumerated in topologically sorted order, the embedding can be defined inductively. The details of the embedding are given in appendix II. An arbitrary element of the domain D can be embedded by embedding all its prime approximations.

5.3 A Universal Language for Sequential Computation

To embed an arbitrary domain D, we must define a retraction (an idempotent function) f_D over $\mathbb{T}$ such that its range $f_D(\mathbb{T})$ is isomorphic to D. Then, sequential computations over D can be performed in the universal domain $\mathbb{T}$. To express sequential computations over arbitrary OS-domains, we must design a language that can express all computable OS-functions over $\mathbb{T}$. In this section, we introduce such a *universal* language KL (Kleene's Language) for the domain $\mathbb{T}$.

The language KL includes a binary operator $Apply$: $\mathbb{T} \times \mathbb{T} \rightarrow_{os} \mathbb{T}$ and a set of constants: $\{0, \mathsf{error}_i, 1^+, 1^-, \mathsf{pair?}, \mathsf{left}, \mathsf{right}, \mathsf{cons}, \mathsf{if0}\}$. These primitives construct trees, select subtrees, and perform arithmetic operations on natural numbers. All the constants except 0 and error_i are elements of $\mathbb{T} \rightarrow_{os} \mathbb{T}$ (embedded in $\mathbb{T}$); they are interpreted as functions by $Apply$. In Fig. 1, we present the constants in $\mathbb{T} \rightarrow_{os} \mathbb{T}$ as uncurried functions instead of the explicit tree representations for clarity.

In the tradition of Kleene, programs in KL are sets of recursive definitions over the primitive operations using $Apply$. Thus, a function is defined by composition and application of other functions and recursion. The denotation of a function defined by a recursive definition is the least fixed point of corresponding functional [13, 2].

All the primitive operations and $Apply$ over $\mathbb{T}$ are observably sequential. Since composition, application, and least fixed points preserve observable sequentiality, KL can only express the OS-functions.

Figure 2 defines the operational semantics of KL, where $f(x)$ abbreviates $Apply(f, x)$. The operational semantics defined by these rewrite rules exactly match the mathematical semantics. In other words, the evaluation of an expression M yields the potentially infinite canonical form for the tree d denoted by M. The evaluation of M terminates iff d is finite and maximal.

Compatibility with λ-Notation: It is easy to define an alternate syntax for KL based on λ-notation. In that notation, λ-abstractions are abbreviations for terms constructed by the combinators S and K, and the primitive operations

Nullary Operations:

$$0 = \ulcorner 0 \urcorner \qquad\qquad \mathsf{error}_i = \mathsf{error}_i$$

Unary Operations:

$$\begin{aligned}
1^+ &= \{n \mapsto n+1\} \cup \{\mathsf{error}_i \mapsto \mathsf{error}_i\}\\
1^- &= \{0 \mapsto 0\} \cup \{n+1 \mapsto n\} \cup \{\mathsf{error}_i \mapsto \mathsf{error}_i\}\\
\mathsf{pair?} &= \{\langle \bot, \bot \rangle \mapsto 0\} \cup \{n \mapsto 1\} \cup \{\mathsf{error}_i \mapsto \mathsf{error}_i\}\\
\mathsf{left} &= \{\langle x, \bot \rangle \mapsto x\} \cup \{\mathsf{error}_i \mapsto \mathsf{error}_i\}\\
\mathsf{right} &= \{\langle \bot, y \rangle \mapsto y\} \cup \{\mathsf{error}_i \mapsto \mathsf{error}_i\}
\end{aligned}$$

Binary Operations:

$$\mathsf{cons} = \{x \times y \mapsto \langle x, y \rangle\}$$

Ternary Operations:

$$\begin{aligned}
\mathsf{if0} = \{0 \times x \times \bot \mapsto x\} &\cup \{1 \times \bot \times y \mapsto y\}\\
&\cup \{\mathsf{error}_i \times \bot \times \bot \mapsto \mathsf{error}_i\}
\end{aligned}$$

Fig. 1. Primitive Operators of KL

$$\begin{array}{rclrcl}
1^+(n) &\longrightarrow& n+1 & \mathsf{cons}(a)(b) &\longrightarrow& \langle a, b \rangle\\
1^+(\mathsf{error}_i) &\longrightarrow& \mathsf{error}_i & \mathsf{if0}(0)(a)(b) &\longrightarrow& a\\
1^-(0) &\longrightarrow& 0 & \mathsf{if0}(n+1)(a)(b) &\longrightarrow& b\\
1^-(n+1) &\longrightarrow& n & \mathsf{if0}(\mathsf{error}_i)(a)(b) &\longrightarrow& \mathsf{error}_i\\
1^-(\mathsf{error}_i) &\longrightarrow& \mathsf{error}_i & \mathsf{pair?}(\mathsf{error}_i) &\longrightarrow& \mathsf{error}_i\\
\mathsf{pair?}(\langle a, b \rangle) &\longrightarrow& 0 & \mathsf{pair?}(n) &\longrightarrow& 1\\
\mathsf{left}(\langle a, b \rangle) &\longrightarrow& a & \mathsf{right}(\langle a, b \rangle) &\longrightarrow& b\\
\mathsf{left}(\mathsf{error}_i) &\longrightarrow& \mathsf{error}_i & \mathsf{right}(\mathsf{error}_i) &\longrightarrow& \mathsf{error}_i
\end{array}$$

Fig. 2. Rewriting Rules for KL

$\{1^+, 1^-, \mathsf{left}, \mathsf{right}, \mathsf{cons}, \mathsf{pair?}, \mathsf{if0}\}$. The combinators S and K are definable by recursive equations in KL, but the definitions are a bit tedious [4]. Since the set of OS-domains and OS-functions form a cartesian closed category [8], we know that S and K are OS-functions. For the proofs in this paper, it is more convenient to write programs as recursion equations.

6 Computability and Universality

KL can be used to perform arbitrary sequential computations over an OS-domain if it can express all computable OS-functions. The proof for the expressiveness of KL has two parts: First we must show that any partial recursive function over the natural numbers can be defined in KL. Second, we must show that any element or OS-function that is computable on the domain $\mathbb{T}$ can be defined in KL.

We show that the language KL is partially recursive by encoding an arbitrary partial recursive function in the language. It is well known [16] that any partial recursive function can be expressed by the operations: $0, 1^+, 1^-$, selection, composition, primitive recursion, and minimization. All these operations can be expressed in KL over the natural numbers.

A programming language $\mathcal{L}$ is universal for a domain D iff the language can express all the computable elements of D and all the computable functions over D. If we encode the prime elements of D as natural numbers, an element in D is computable if the prime approximations are recursively enumerable. An *OS*-function is computable if its graph, *i.e.*, the set $\{(\mathbf{b}, f(\mathbf{b}))\}$ where $\mathbf{b}$ are finite elements, is recursively enumerable. Since the *OS*-functions can be embedded in $\mathbb{T}$ and interpreted by *Apply*, it suffices to show that KL can define all the computable elements of $\mathbb{T}$ to prove the universality of KL.

We can encode the prime elements of the Universal Domain $\mathbb{T}$ as follows. We can map numbers into the possible values at the end of the node with a function *val* defined as $val(0) = \mathsf{cons}$, $val(1) = \mathsf{error}_1$, $val(2) = \mathsf{error}_2$, and for all other numbers $val(m) = m - 3$. Each path of the tree can be encoded as a binary number where the leading 1 represents the beginning of the tree and every subsequent 0 or 1 represents a left branch or right branch, respectively. For example 1001 is the binary encoding of the path *left, left, right.*

If the prime approximations to a computable element d are r.e., then there is a partial recursive function, called the characteristic function, f_d such that $f_d(n) = m$ iff the element of the node at the end of the path represented by n is $val(m)$. Using such an encoding, the language KL can express any computable element d in $\mathbb{T}$. Since the element d is computable, it has a characteristic function f_d that is partially recursive. Since any partial recursive function can be written in the language KL, some function $\mathbf{f}$ in KL can express f_d. Given such a function $\mathbf{f}$, we can recursively construct the element d as follows:

$$\begin{aligned}
2times(n) &= \mathsf{if0}(n, n, 1^+(1^+(2times(1^-(n))))) \\
g(n) &= \mathsf{if0}(\mathbf{f}(n), \mathsf{cons}(g(2times(n)), g(1^+(2times(n))))), val(n)) \\
\mathbf{d} &= g(1)
\end{aligned}$$

The expression d recursively constructs the tree corresponding to the element d with the help of the function g.

Full abstraction of KL**:** Since every finite element of the domain is computable, every finite element can be defined in KL. From that fact we can prove that the domain $\mathbb{T}$ is a full abstract model for KL as follows. If KL is not fully abstract, then there are two terms M and N that are operationally equivalent but denotationally different. Since their denotations are different, there must be a prime element (a tree node) in one of the denotations, but not the other. Therefore, we can write a simple program P using left and right operations that extracts the differentiating node. Hence P distinguishes M and N contradicting the assumption that they are operationally equivalent. Thus, we conclude that in KL denotational equivalence implies operational equivalence.

7 Conclusions and Acknowledgments

Our investigation of OS-domains provides strong support for the claim that KL is a universal programming language for sequential computation. Any OS-domain D can be embedded in the domain $\mathbb{T}$ by an observably sequential retraction that preserves the maximality of finite elements. Moreover, any computable function f over D is definable in KL over the embedded image of D. Since the maximality of finite elements is preserved by the embedding, the image of f in KL behaves exactly like f—including termination behavior.

We thank Matthias Felleisen and Pierre-Louis Curien for correcting errors in an earlier draft of the paper and making many helpful suggestions on how to present our technical results and intuitive explanations.

References

1. G. Berry and P-L. Curien. Sequential algorithms on concrete data structures. *Theor. Comput. Sci.*, 20:265–321, 1982.
2. R. Cartwright. Recursive programs as definitions in first order logic. *SIAM Journal of Computing*, 13:374–408, 1984.
3. R. Cartwright and A. Demers. The topology of program termination. In *Proc. Symposium on Logic in Computer Science*, pages 296–308, 1988.
4. R. Cartwright and J. Donahue. The semantics of lazy (and industrious) evaluation. In *Proc. 1982 ACM Symposium on Lisp and Functional Programming*, pages 253–264, 1982.
5. Robert Cartwright and Matthias Felleisen. Observable sequentiality and full abstraction. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 328–342, January 1992.
6. R.S. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract models of observably sequential languages. *Information and Computation.* To appear.
7. Alonzo Church. An unsolvable problem of elementary number theory. *Americal Journal of Mathematics*, 58:345–363, 1936.
8. P.-L. Curien. Observable algorithms on concrete data structures. In *Proc 7th Symposium on Logic in Computer Science*, 1992.
9. P.L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical Computer Science. Birkhauser, 1993.
10. Martin Davis. Computable functionals of arbitrary finite type. In A. Heyting, editor, *Constructivity in Mathematics.* North Holland Publishing Company, Amsterdam, 1959.
11. G. Kahn and G. Plotkin. Concrete domains. In *Fertschirift for C. Boehm*, 1993.
12. R. Kanneganti, R. Cartwright, and M. Felleisen. Spcf: its model, calculus, and computational power. In *Proc. REX Workshop on Semantics and Concurrency.* Lecture Notes in Computer Science. Springer Verlag, 1992.
13. Stephen C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112:727–742, 1936.
14. G. Kreisel, D. Lacombe, and J. R. Scoenfield. Partial recursive functionals and effective operations. In A. Heyting, editor, *Constructivity in Mathematics.* North Holland Publishing Company, Amsterdam, 1959.
15. G.D. Plotkin. T^ω as a universal domain. *J. Comput. Sci.*, 17:209–236, 1978.

16. Hartley Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, New York, 1967. McGraw-Hill series in higher mathematics.
17. D.S. Scott. Data types as lattices. *SIAM J. Computing*, 5:522–587, 1976.
18. Alan M. Turing. Computability and λ-definability. *The Jounal of Symbolic Logic*, 2:153–163, 1937.

Appendix I: Effective Presentation of Function Spaces

In this appendix, we show how to construct an effective presentation $\mathcal{P}$ of the function space $D_1 \rightharpoonup_{os} D_2$ given the effective presentations $\mathcal{P}_1$ and $\mathcal{P}_2$ for D_1 and D_2. Since the *finite* elements of D_i are the least upper bounds of sets of primes, we can assign unique codes in $\mathbb{N}$ to the *finite* elements of D_i using standard methods for encoding sequences. Let the indices I, J range over the indices of finite elements and i, j, k range over the indices of prime elements. We refer to the finite element with index I as a_I and the prime element with index i as p_i. We superscript elements with their domains whenever it is necessary to avoid confusion.

The index of each prime function is defined inductively as follows. Let the index of the $\bot \rightharpoonup_{os} \bot$ be 0. Let f be a prime function immediately above f'. Let the index of f' be l. Then,

$$\mathbf{E}(f) = \begin{cases} \langle\langle 0, I, k\rangle, l\rangle & \text{if } f \text{ is an output prime determined by } f', a_I, \text{ and } p_k \\ \langle\langle 1, \langle I, m\rangle, n\rangle, l\rangle & \text{if } f \text{ is a schedule prime with } si_f(a_I, C_m) = C_n \end{cases}$$

where $\langle\cdot,\cdot\rangle$ is the standard bijective pairing function on $\mathbb{N}$. It is clear from the above encoding that the enumeration is topologically sorted.

pr is the function that selects the second component of a tuple $\langle i, j\rangle$ since p_j is the immediate predecessor of the prime element $p_(i, j\rangle$.

The function **c** is defined by the equations:

$$\begin{aligned} \mathbf{c}(\langle\langle 0, I, k\rangle, l\rangle) &= \langle\langle I, \mathbf{c}_2(k)\rangle, l\rangle \\ \mathbf{c}(\langle\langle 1, \langle I, m\rangle, n\rangle, l\rangle) &= \langle\langle I, m\rangle, l\rangle \end{aligned}$$

From the above encoding, it is easy to show that elements in the same C-set are mapped to same number.

The function **arity** is defined by the equations:

$$\begin{aligned} \mathbf{arity}(\langle\langle 0, I, k\rangle, l\rangle) &= \mathbf{arity}_2(k) \\ \mathbf{arity}(\langle\langle 1, \langle I, m\rangle, n\rangle, l\rangle) &= \mathbf{size}_1(n) \end{aligned}$$

The number of C-sets above an output prime f depends on the number of C-sets above the final output p_k of f. The function f may generate incremental outputs in all the directions above p_k; therefore $\mathbf{arity}(f) = \mathbf{arity}_2(k)$. If f is a schedule prime with $si_f(a_I, C_m) = C_n$, f may generate a C-set for each element in the input C-set C_n; therefore $\mathbf{arity}(f) = \mathbf{size}_1(n)$.

The function **size** is defined as follows. Let the index of the C-set Q be $\langle\langle I, m\rangle, l\rangle$. Let S_I be the set of indices primes on the "fringe" of a_I, *i.e.*, if $i \in S_I$ then $p_i \in a_I$ and there is no $p_j \in a_I$ such that $\mathbf{pr}(j) = i$. Then,

$$\mathbf{size}(\langle\langle I, m\rangle, l\rangle) = \mathbf{size}_2(m) + \sum_{i \in S_I} \mathbf{arity}_1(i)$$

where the addition operator $+$ has been extended to the domain $\mathbb{N} \cup \{\omega\}$. We can easily verify that the function **size** gives the cardinality of the C-set in $D_1 \rightarrow_{os} D_2$. It is clear that the number of output primes in the C-set Q is exactly the number of primes in the output C-set C_m. The C-set Q has a schedule prime for each direction above a_I. Thus, the preceding definition of **size** computes the number of elements in the C-set.

Appendix II: Embedding in the Universal Domain

In this appendix, we show how to embed an effectively presented domain D within $\mathbb{T}$. We define the embedding function f mapping each prime element p_i into $\mathbb{T}$ by induction on i. At each stage in the construction of f, we prove the following hypotheses.

1. f maps the prime element p_i to a tree formed by substituting a subtree s_i for a $\perp$ leaf in the tree corresponding to its predecessor. The subtree s_i has same number of $\perp$'s as the number of C-sets above p_i.
2. If $p, q \in C_i$ (the same set in the partitioning), then the trees $f(p)$ and $f(q)$ are inconsistent.
3. If $p \leq q$, then $f(p) \sqsubseteq f(q)$.
4. If a finite element a represented by the set A of prime elements is maximal, the tree $\bigsqcup\{f(p) \mid p \in A\}$ is maximal.

Base Case: f maps p_0, which must be $\perp$, to $\langle n, \perp^n\rangle$ where n is the number of C-sets above $\perp$.

Induction Step: Assume that we have embedded all the primes in the enumeration from p_0 through p_{i-1}. We define $f(p_i)$ as follows:

Let k be the cardinality of the set of the already enumerated elements that are in conflict with p_i. Let s_i, the subtree for p_i be $\langle\langle\langle \mathbf{size}(\mathbf{c}(p_i)), k\rangle, \mathbf{arity}(p_i)\rangle, \perp^l\rangle$, where l is the number of C-sets above p_i. Note that l can be ω. Let q be the element immediately below p_i. By induction, q has been embedded in $\mathbb{T}$ as a tree $f(q)$. The element q was embedded in $\mathbb{T}$ by adding a subtree s_q to the image of q's predecessor. (In the special case where $q = \perp$, let s_q be $f(\perp)$.) In the subtree s_q, insert s_i (the subtree corresponding to p_i) at position $g(p_i)$, where the function $g(p_i)$ is defined as follows. If a prime p' belonging to the same C-set as p_i has been enumerated, then $g(p_i) = g(p')$. Otherwise, $g(p_i)$ is the number of distinct C-sets above q that are already enumerated. By the induction hypothesis, the position $g(p_i)$ has $\perp$ in the subtree s_q.

It is easy to show that by the preceding construction satisfies the hypotheses.