

On the Expressiveness of Affine Programs with Non-local Control: The Elimination of Nesting in SPCF

J. Laird*

Department of Informatics,
University of Sussex, United Kingdom
e-mail: jml@sussex.ac.uk

Abstract. We use a denotational semantics to show that every term in SPCF (a typed functional language with simple non-local control operators) is contextually equivalent to one which is typable in an affine typing system. Nested function calls and recursive definitions are *not* affinely typable, and so our result entails that they can be eliminated from SPCF without losing expressiveness.

Our proof is based on the observation of Longley that every type of SPCF is a retract of a first-order type. We describe retractions of this kind in *bistable biorder* models of SPCF which are definable in the affine fragment. This allows us to transform an arbitrary SPCF term into an affine one by mapping it to a first-order term, obtaining an (affine) “normal form”, and then projecting back to the original type.

We show the flexibility of our approach by considering two variants of SPCF, a finitary, call-by-name version and a call-by-value version over the natural numbers. In the infinitary case, (in which we establish in addition that all instances of recursive definition may be replaced with iteration) our proof is based on an analysis of the relationship between SPCF definable functions and *strategies* for computing them sequentially.

1. Introduction

One of the important sources of expressive power in functional languages is the *nesting* of function calls (both explicitly, and in recursive definitions) and, more generally, the sharing of free variables between procedures and their arguments. With this expressive power comes subtle and complex behaviour, which complicates the implementation and analysis of functional programs. In this paper, we will show that simple non-local control operators can be used to *eliminate* such behaviour: we show that every program

*Supported by UK EPSRC grant GR/S72181.

of SPCF (a prototypical functional language with control) is observationally equivalent to a term which satisfies affine typing rules which prevent function-nesting, and in which recursion is eschewed in favour of iteration.

Formally, we shall show that every term of SPCF is contextually equivalent to one which is typable in a simple affine typing system. The latter requires that in the application of M to N , the terms M and N should not share any free variables. Sharing is permitted between terms which are executed sequentially, for example, between M and N_1 and N_2 in `If M then N_1 else N_2` . Roughly speaking, this affine typing is not a requirement to use each resource at most once, but to use at most one copy of each resource at a time. It is also the basis of Reynolds' *Syntactic Control of Interference* typing system for functional languages with state [22], where it prevents covert interference between procedures which use the store.

To highlight the key role that non-local control plays in this result, we observe that a simple example suffices to establish that nesting cannot be eliminated in the purely functional language PCF: there is no affinely typed term of (call-by-name) PCF equivalent to $G(f) = ((f \text{ tt}) ((f \text{ ff}) \text{ tt})) : \text{bool}$ (Lemma 2.1). However, in SPCF we may use control operators to define tests such as `strict(g)`, which returns `tt` if $g : \text{bool} \Rightarrow \text{bool}$ evaluates its argument, and `ff` otherwise, and so define a term equivalent to $G(f)$ without nesting, for example:

$$\text{If } \text{strict}(f \text{ tt}) \text{ then (If } ((f \text{ ff}) \text{ tt}) \text{ then } ((f \text{ tt}) \text{ tt}) \text{ else } ((f \text{ tt}) \text{ ff}) \text{ else } ((f \text{ tt}) \text{ tt})$$

We consider two versions of SPCF, a call-by-name version over bounded datatypes and a call-by-value version over the natural numbers. In the latter case, affine typing excludes recursive definition (by fixpoint combinator) as well as sharing of variables between procedures and their arguments.

The proofs of our results proceed via analysis of the fully abstract denotational semantics of SPCF. This can be described in two styles: intensional and extensional. In the intensional presentation [7], types are interpreted as sequential data structures or games, and programs as sequential algorithms or strategies. In the extensional presentation [13], types are interpreted as “bistable biorders” and programs as bistable functions. In this paper, we work with the latter, using bistable functions as a simple representation of observational equivalence classes of SPCF programs. This has the advantage of allowing some simple algebraic proofs of equivalence between program fragments, although there is no reason *in principle* why the same results could not be obtained using the sequential algorithms model. In fact, our proof inevitably requires an analysis of the intensional properties of bistable functions, and leads to a self-contained account of the relationship between bistable functions and evaluation strategies for a small fragment of SPCF (general accounts appears in [16, 13]).

We use our models to show that every SPCF type-object is a *retract* of a product of atomic type-objects (in the unbounded, call-by-value case this tuple may be viewed as a first-order function-space). That is, for any type-object A there is an embedding from A into such a tuple, and a projection back into A which compose to give the identity on A . Moreover, this embedding-projection pair is the denotation of a pair of *affinely typable* terms of SPCF. This gives a faithful, internally definable representation of higher-order functionals as lists of numerals or first-order functions, which can be thought of as a *compilation* of a functional program down into a low-level representation as a series of instructions (or, in the unbounded case, a recursive procedure for enumerating such a sequence). Since finite tuples and (computable) first order functions are all definable in the affine fragment of SPCF, we can obtain an affine form for any term by extracting a low-level representation (by embedding) and then applying the affine projection back into the original type.

1.1. Related Work

This paper is a revised and expanded version of a conference paper [15]. The existence of SPCF-definable retractions of higher-order type-objects into a universal first-order function-space in the sequential algorithms semantics was first described by John Longley [19, 20]. Their use to prove full abstraction for bistable models of SPCF is studied in [12, 13].

We may contrast our results with research into type-theories for functional languages and λ -calculi which are used to guarantee polynomial-time normalization of all typable terms [10, 3, 11]. The use of higher-order procedures appears in some senses more constrained in our affine type theory, but it is fully expressive at all types because we allow unlimited sequential uses of procedures (in particular, we can define all partial recursive first-order functions). Indeed, our results suggest that the implicit computational complexity of SPCF programs at higher types could be characterized in terms of the complexity of the first-order functions to which they are reduced by definable retractions.

More generally, transformation of SPCF programs into an equivalent affine form may be compared with *refactoring* — the manipulation of programs to improve performance without altering observable behaviour. For example, it is well known that any instances of tail-recursion may be eliminated in favour of iteration. Our transformation is wholesale (although it could be applied to program fragments) and rather than improving performance, converts a potentially very complicated program (which may be difficult to implement or reason about) to a much larger, but simpler form. It may, however, suggest instances in which a less radical refactoring to a smaller normal form is possible.

2. Bounded SPCF

SPCF, or “observably sequential” PCF [6] is an applied simply-typed λ -calculus (PCF) with a basic form of non-local *control operator* (and possibly some unrecoverable errors, which are omitted here). We first introduce a bounded version of SPCF with product types (including the empty product, I) in which the datatypes are finitary, taking the form \underline{n} (the type of natural numbers less than n). So the types of bounded SPCF are given by the following grammar:

$$S, T ::= I \mid B \mid S \Rightarrow T \mid S \times T$$

where $B = \{\underline{n} \mid n \in \mathbb{N}\}$. We assume a suitable encoding using binary products of the n -fold product T^n of copies of T (with $T^0 = I$) and the projections $\pi_i : T^n \Rightarrow T$.

Terms are formed from the simply-typed λ -calculus (with products) over these types, extended with the following constants:

Numerals $\mathbf{i} : \underline{n}$ for each $i < n$.

Conditional $\mathbf{case} : \underline{n} \times T^n \Rightarrow T$

Divergence $\perp : T$.

Control A control operator $\mathbf{catch}_n : (T^n \Rightarrow \underline{0}) \Rightarrow \underline{n}$, introduced by Cartwright and Felleisen [6]. This evaluates to the numeral \mathbf{i} if its argument is strict in its i th argument, (since its argument is a (sequential) function into the empty type, the only other possibility is that it diverges).

$E[(\lambda x.M) N]$	\longrightarrow	$E[M[N/x]]$
$E[\pi_i \langle M_1, M_2 \rangle]$	\longrightarrow	$E[M_i]$
$E[\text{case } \langle n, M \rangle]$	\longrightarrow	$E[\pi_n M]$
$E[\text{catch}_n \lambda k.E'_k[\pi_n k]]$	\longrightarrow	$E[m]$

Table 1. “Small-step” operational semantics for SPCF programs.

A “small-step” call-by-name operational semantics for bounded SPCF programs (closed terms of basic type) is given in Table 1. It is based on *evaluation contexts*, which are given by the grammar:

$$E[_] ::= [_] \mid E[_] M \mid \text{catch}_n E[_] \mid \lambda k.E[_] \mid \pi_i E[_] \mid \mid \text{case } E[_] \mid \text{case} \langle E[_], M \rangle$$

We write $E_k[_]$ for an evaluation context which does not bind the variable k , and assume that all substitutions are capture-avoiding. For a program M we write $M \Downarrow$ if there exists a value v (a numeral) such that $M \longrightarrow^* v$ (thus all programs of the form $E[_]$ are considered to diverge). We adopt standard definitions of observational approximation and equivalence. Given terms $M, N : T$:

- $M \lesssim N$ if for all compatible program contexts $C[_]$, $C[M] \Downarrow$ implies $C[N] \Downarrow$,
- $M \simeq N$ if $M \lesssim N$ and $N \lesssim M$.

2.1. Affine Typing

A sublanguage, affine SPCF or ASPCF, is obtained by restricting to the terms of SPCF which are derivable in the affine λ -calculus with (additive) pairing, the rules of which are given in Table 2. We will show that this fragment of SPCF is fully expressive: for every SPCF term M there is an ASPCF term M' such that $M \simeq M'$. But what does it mean, operationally, for a term to be affinely typable?

First, observe that affine typing means that the arguments to a term of type $A \Rightarrow (B \Rightarrow C)$ cannot share variables — i.e. if $(M N) N'$ is affinely typable then N, N' contain no free variables in common. Thus there is no (Currying/uncurrying) isomorphism between the types $(A \times B) \Rightarrow C$ and $A \Rightarrow (B \Rightarrow C)$. It would be possible to add a multiplicative product, with sharing not permitted between conjuncts, with respect to which currying would hold (although it is not clear how to interpret such an extension to SPCF whilst retaining full abstraction). Typing the `case` constructor using the additive product allows sharing of variables between the term which is tested and (all of) the branches. This contrasts with the multiplicative rules for the conditional in accounts of linear PCF such as [5], and suggests that in the current case affine typing does not correspond to the single use of each computational resource, but to the fact that a single “copy” of each resource may be in use at any time. The same typing rules are proposed by Reynolds in a functional-imperative setting to enforce *Syntactic Control of Interference* (SCI) [22] — the prevention of covert interference between procedures arising from shared variables. An immediate consequence is to exclude nesting of function variables, but affineness also excludes terms such as $\lambda f.\lambda x.((f x) x)$, where x is a variable of ground type. There are typing systems which eliminate function-nesting but permit “first-order” sharing of this kind (e.g. Serially Re-entrant Idealized Algol [2], and Syntactic Control of Concurrency [9]). Of course, our result establishes that imposing these

$$\begin{array}{c}
\frac{}{\Gamma, x:S \vdash x:S} \quad \frac{\Gamma, x:S \vdash M:T}{\Gamma \vdash \lambda x.M : S \Rightarrow T} \quad \frac{\Gamma \vdash M : S \Rightarrow T \quad \Delta \vdash N : S}{\Gamma, \Delta \vdash M N : T} \\
\frac{}{\Gamma \vdash \langle \rangle : I} \quad \frac{\Gamma \vdash M : S \quad \Gamma \vdash N : T}{\Gamma \vdash \langle M, N \rangle : S \times T} \quad \frac{\Gamma \vdash M : S_1 \times S_2}{\Gamma \vdash \pi_i M : S_i} \quad i \in \{1, 2\}
\end{array}$$

Table 2. Typing judgements for the affine λ -calculus

weaker typing restrictions on SPCF does not reduce its expressivity. By excluding all sharing between procedures and their arguments, we exercise tighter control over (for example) the points from which an argument may be called.

We may show that nesting is not eliminable in finitary PCF (demonstrating the importance of non-local control operators) by proving the claim made in the introduction — that there is no affinely-typed term of finitary PCF (i.e. the simply-typed λ -calculus over the type `bool` with divergence, boolean values and conditional) which is equivalent to $G(f) = ((f \text{ tt}) ((f \text{ ff}) \text{ tt}))$.

Proposition 2.1. Any affinely typed term F of bounded PCF is not (PCF) observationally equivalent to G .

Proof:

It is straightforward to show that bounded PCF is strongly normalising with respect to the following reductions, which are sound with respect to PCF-equivalence and preserve affine typing: $(\lambda x.M) N \longrightarrow M[N/x]$, $\pi_i \langle M_1, M_2 \rangle \longrightarrow M_i$ and $\text{If } \text{tt} \text{ then } M_1 \text{ else } M_2 \longrightarrow M_1$, $\text{If } \text{ff} \text{ then } M_1 \text{ else } M_2 \longrightarrow M_2$.

So we may assume that F is in normal form, and therefore has the form $\lambda f.\text{tt}$, $\lambda f.\text{ff}$, $\lambda f.E[\perp]$ or $\lambda f.E[(f M_1) M_2]$, where $E[_]$ is a context given by the grammar:

$E[_] ::= [_] \mid \text{If } E[_] \text{ then } M_1 \text{ else } M_2$

Clearly, tt and ff are not equivalent to $G(f)$, and nor is $E[\perp]$ (which is equivalent to \perp). So we may assume that $F = \lambda f.E[(f M_1) M_2]$. Since M_1 and M_2 may not contain f , they are closed terms, and we may therefore assume that they are either tt , ff or \perp . We show by case analysis that there is always a term $N : \underline{2} \Rightarrow \underline{2} \Rightarrow \underline{2}$ such that $G N \rightarrow \text{tt}$ and $F N \longrightarrow E[\perp]$.

- If $M_1 = \text{ff}$ or $M_1 = \perp$, then let $N = \lambda xy.\text{If } x \text{ then } \text{tt} \text{ else } \perp$.
- If $M_1 = \text{tt}$ and $M_2 = \text{tt}$, then let $N = \lambda xy.\text{If } x \text{ then } (\text{If } y \text{ then } \perp \text{ else } \text{tt}) \text{ else } \text{ff}$.
- If $M_1 = \text{tt}$ and $M_2 = \text{ff}$, then let $N = \lambda xy.\text{If } x \text{ then } (\text{If } y \text{ then } \text{tt} \text{ else } \perp) \text{ else } \text{tt}$.

□

Note that this proof would apply to any typing system which prevents nesting of function variables and possesses subject reduction with respect to β -reduction.

2.2. Denotational Semantics

We will now briefly describe the denotational semantics of bounded SPCF [12, 13] in the category of bistable biorders and monotone and bistable functions. A bistable biorder may be presented as a partial

order with an equivalence relation (*bistable coherence*); two functions are equivalent essentially if they explore their arguments in the same way, but may fail (either through divergence, or by producing a \top “error” element) in different ways. (So, for example, if Σ is the two-point linear order \perp, \top , with $\top \uparrow \perp$, the order $(\Sigma \Rightarrow \Sigma) \Rightarrow \Sigma$ is the four-point linear-order $\perp, \lambda f.f(\perp), \lambda f.f(\top), \top$, with $\perp \uparrow \top$ and $\lambda f.f(\perp) \uparrow \lambda f.f(\top)$.)

Definition 2.1. A *bistable biorder* consists of a partial order (D, \sqsubseteq) and an equivalence relation \uparrow on D such that for any element a , the equivalence class $[a]$ of a with respect to \uparrow , ordered by \sqsubseteq , is a *distributive lattice*¹, and the inclusion of $([a], \sqsubseteq)$ in (D, \sqsubseteq) preserves binary greatest lower bounds and least upper bounds. A bistable biorder is *pointed* if (D, \sqsubseteq) has least and greatest elements \perp, \top , such that $\perp \uparrow \top$.

For each set X we have a “flat” biorder \tilde{X} consisting of the elements of X with the trivial order and coherence relation — i.e. $x \sqsubseteq y$ iff $x = y$ iff $x \uparrow y$.

Definition 2.2. A function $f : D \rightarrow E$ is *bistable* if its restriction to each \uparrow -equivalence class is a lattice homomorphism — i.e. if $x \uparrow y$, then $f(x) \uparrow f(y)$, $f(x \wedge y) = f(x) \wedge f(y)$ and $f(x \vee y) = f(x) \vee f(y)$. If D, E are pointed, then f is *strict* if $f(\top) = \top$ and $f(\perp) = \perp$.

Proposition 2.2. Bistable biorders and bistable and \sqsubseteq -monotone functions form a (bi)Cartesian closed category.

Proof:

The product and coproduct are defined pointwise on the underlying sets. The exponential $A \Rightarrow B$ consists of the bistable and continuous functions from A to B , with the extensional (Scott) ordering, and bistable coherence defined by $f \uparrow g$ if $x \uparrow y$ implies $f(x) \uparrow g(y)$ and $f(x) \wedge g(y) = f(y) \wedge g(x)$ and $f(x) \vee g(y) = f(y) \vee g(x)$. This is pointed if B is pointed.

For the proof that this does in fact define a bistable biorder, we refer to [13], but to prove that \uparrow is transitive, for example, suppose $f \uparrow g$ and $g \uparrow h$. Suppose $x \uparrow y$. Then:

$$\begin{aligned} f(x) \wedge h(y) &= f(x) \wedge h(y) \wedge (f(x) \vee g(y)) = f(x) \wedge h(y) \wedge (f(y) \vee g(x)) \\ &= (f(x) \wedge h(y) \wedge f(y)) \vee (f(x) \wedge h(y) \wedge g(x)) \end{aligned}$$

$$\sqsubseteq f(y) \vee (f(x) \wedge g(y) \wedge h(x)) = f(y) \vee (f(y) \wedge g(x) \wedge h(x)) = f(y).$$

Similarly, $f(x) \wedge h(y) \sqsubseteq h(x)$, $f(y) \wedge h(x) \sqsubseteq f(y)$ and $f(y) \wedge h(x) \sqsubseteq h(x)$. So $f(x) \wedge h(y) = f(y) \wedge h(x)$. By duality, $f(x) \vee h(y) = f(y) \vee h(x)$ and so $f \uparrow h$ as required. \square

Thus we may interpret bounded SPCF in the CCC of bistable biorders by fixing the denotations of the the ground types and constants. We interpret the type \underline{n} as the *bilifting* of the flat order $\lceil n \rceil = \{i \mid 0 \leq i < n\}$, where the bilifting operation $(-)\perp$ adds (bistably coherent) elements \top and \perp .

Definition 2.3. For any bistable biorder B , we define $B\perp$ to be the pointed bistable biorder with $|B\perp| = (|B| + \emptyset) \cup \{\top, \perp\} = |B| \times \{1\} \cup \{\perp, \perp\}$; $x \sqsubseteq y$ if $x = \perp$ or $y = \top$ or $x = \text{in}_1(x'), y = \text{in}_1(y')$ and $x' \sqsubseteq y'$; $x \uparrow y$ if $x, y \in \{\top, \perp\}$ or $x = \text{in}_1(x'), y = \text{in}_1(y')$ and $x' \uparrow y'$. We write Σ for $\emptyset\perp$.

So we define $\llbracket n \rrbracket = \lceil n \rceil\perp$, and interpret the case construct as the uncurrying of the strict function which sends i to the projection π_i . The interpretation of the control operator *catch* is based on the observation that bistable and monotone/continuous functions are *sequential* in the following sense.

¹Taking a lattice to be a partial order with meets and joins of finite *non-empty* sets

Definition 2.4. A function $f : A_0 \times \dots \times A_n \rightarrow B$ (for pointed A_1, \dots, A_n) is i -strict if $\pi_i(e) = \top$ implies $f(e) = \top$ and $\pi_i(e) = \perp$ implies $f(e) = \perp$.

Lemma 2.1. Given pointed bistable biorders A_1, \dots, A_n , every strict, monotone and bistable function $f : A_1 \times \dots \times A_n \rightarrow \Sigma$ is i -strict for some $i \leq n$.

Proof:

Given $j \leq n$, let $\perp[\top]_j = \langle x_i \mid i \leq n \rangle$, where $x_i = \top$ if $i = j$, and $x_i = \perp$ otherwise. Similarly $\top[\perp]_j = \langle x_i \mid i \leq n \rangle$, where $x_i = \perp$ if $i = j$, and $x_i = \top$ otherwise.

If $\pi_i(x) = \perp$ then $x \sqsubseteq \top[\perp]_i$, and if $\pi_i(x) = \top$, $\perp[\top]_i \sqsubseteq x$. Thus f is i -strict if $f(\top[\perp]_i) = \perp$ and $f(\perp[\top]_i) = \top$. Since $\perp \leq^B \top$, we have $\top[\perp]_j \downarrow \top[\perp]_k$ for all $j, k \leq n$, and $\bigwedge_{i \leq n} \top[\perp]_i = \perp$. Hence $\bigwedge_{i \leq n} f(\top[\perp]_i) = f(\bigwedge_{i \leq n} \top[\perp]_i) = f(\perp) = \perp$, and so for some i , $f(\top[\perp]_i) = \perp$. Similarly $f(\bigvee_{i \in I} (\perp[\top]_i)) = \top$, and so $f(\perp[\top]_j) = \top$ for some j . Moreover, if $i \neq j$, then $\perp[\top]_j \sqsubseteq \top[\perp]_i$, and so either $\perp[\top]_j = \top[\perp]_i$ — in which case each A_k is the one-point order — or else $i = j$ as required, and hence i is unique — i.e. bisquential functions are *strongly sequential*. \square

So for any biorder A and integer n , we have a strict bistable function $\text{catch} : A^n \Rightarrow \Sigma$ to $\llbracket n \rrbracket$ which sends each i -strict function to the value i , and with which we interpret catch_n . (In particular, note that $\Sigma^n \Rightarrow \Sigma$ is isomorphic to $\llbracket n \rrbracket$.) We may show that for any evaluation context $E[_]$, if $\Gamma, k : \underline{0}^n \vdash E[k] : T$ is a well-formed term then $\llbracket E[k] \rrbracket : \llbracket \Gamma \rrbracket \times \Sigma^k \rightarrow \llbracket T \rrbracket$ is a right-strict function, and hence prove soundness of the reduction rule for catch_n . Using standard arguments we then obtain the following result (see [12, 13]).

Proposition 2.3. For any closed $M : \underline{n}, M \longrightarrow n$ if and only if $\llbracket M \rrbracket = n$.

Corollary 2.1. (Inequational Soundness)

$\llbracket M \rrbracket \sqsubseteq \llbracket N \rrbracket$ implies $M \lesssim N$.

3. Expressiveness of the affine fragment

Our proof that every term of bounded SPCF is contextually equivalent to one which is affinely typable is based on the notion of *definable retraction*.

Definition 3.1. Given types S, T , an ASPCF-definable retraction from S to T is a pair of closed ASPCF terms: $\text{inj} : S \Rightarrow T$ and $\text{proj} : T \Rightarrow S$ which denote a retraction in our bistable model — i.e. $\llbracket \lambda x. \text{proj}(\text{inj}(x)) \rrbracket = \llbracket \lambda x. x \rrbracket$. We write $S \leq T$ if there is a definable retraction from S to T ($S \cong T$ if it is an isomorphism).

For example, the isomorphism between $\Sigma^n \Rightarrow \Sigma$ and $\llbracket n \rrbracket$ is ASPCF-definable as the pair of terms $(\text{catch}_n, \lambda x. \lambda y. \text{case}(x, y))$.

Lemma 3.1. For any m there is an ASPCF-definable isomorphism: $\underline{0}^m \Rightarrow \underline{n} \cong \underline{m+n}$.

Proof:

We have $\Sigma^m \Rightarrow [n]_{\perp}^{\top} \cong \Sigma^m \Rightarrow \Sigma^n \Rightarrow \Sigma \cong \Sigma^{m+n} \Rightarrow \Sigma \cong [m+n]_{\perp}^{\top}$. The defining terms for the isomorphism are $\lambda f. \text{catch}_{m+n} \lambda k. (\text{case} \langle f \langle \pi_i(k) \mid i < m \rangle, \langle \pi_{m+j}(k) \mid j < n \rangle \rangle$ and $\lambda x. \lambda y. \text{catch} \lambda k. \text{case} \langle x, \langle \pi_0(y), \dots, \pi_{m-1}(y), \pi_0(k), \dots, \pi_{n-1}(k) \rangle \rangle$. (In general isomorphisms based on currying and uncurrying are not definable in ASPCF but this case is an exception.) \square

If $S \trianglelefteq T$, then we may represent every program of type S as a program of type T in a way which preserves and reflects denotational equivalence. We will show that every type of bounded SPCF is an ASPCF-definable retract of a type of the form \underline{n}^m . Since any element of such a type is a tuple of values and \top and \perp elements, we demonstrate that we may represent any bounded SPCF program as such a tuple (which one might regard as an assembly language representation), from which it may be recovered with the term proj .

The key to our proof is to show that “first-order” function types are retracts of types of the form \underline{n}^m , since we can then show that each higher-order type is a retract of one at lower order, using the fact that the relation \trianglelefteq is a precongruence on types.

Lemma 3.2. If $T_1 \trianglelefteq T_2$ and $S_1 \trianglelefteq S_2$, then $S_1 \times T_1 \trianglelefteq S_2 \times T_2$ and $S_1 \Rightarrow T_1 \trianglelefteq S_2 \Rightarrow T_2$.

Proof:

For example, the terms $\lambda f x. \text{inj}_T (f (\text{proj}_S x))$ and $\lambda f x. \text{proj}_T (f (\text{inj}_S x))$ define a retraction from $S_1 \Rightarrow T_1$ to $S_2 \Rightarrow T_2$. \square

As an example, consider the type $\underline{2} \Rightarrow \underline{2}$. We may completely characterize $f \in [\underline{2} \Rightarrow \underline{2}]$ by determining (in order) the answers to three questions: Is f strict? What is the value of $f(0)$? What is the value of $f(1)$? Thus we have a retraction from $\underline{2} \rightarrow \underline{2}$ to $\underline{2} \times \underline{2} \times \underline{2}$ which is definable as the terms $\lambda f. \langle \text{strict } f, f 0, f 1 \rangle$ and $\lambda x. \lambda y. \text{case} \langle \pi_0 x, \langle \text{case} \langle y, \langle \pi_1 x, \pi_2 x \rangle \rangle, \pi_2 x \rangle \rangle$, where $\text{strict } f = \text{catch} \lambda x. \text{case} \langle f (\pi_0 x), \langle \pi_1 x, \pi_1 x \rangle \rangle$.

In the general case, we proceed by showing that for any m, n , $\underline{n}^{m+1} \Rightarrow \underline{0} \trianglelefteq \underline{m+1} \times (\underline{n}^m \Rightarrow \underline{0})^n$. Informally, the retraction uses the fact that any bistable function $f : [\underline{n}]^{m+1} \rightarrow \Sigma$ may be represented as a pair consisting of a unique “strictness index” i , together with a tuple of “continuation” functions $f' : [\underline{n}]^m \rightarrow \Sigma$ for each possible value at i .

Definition 3.2. Given a tuple $t = \langle t_0, \dots, t_{n-1} \rangle$ and $i \leq n$, let $t[a]_i$ denote the *insertion* of a at position i in t — i.e. $\langle t_0, \dots, t_{i-1}, a, t_i, \dots, t_{n-1} \rangle$, and (for $j < n$) let $[e]_j$ denote the *removal* of the j th element of t — i.e. $\langle t_0, \dots, t_{j-1}, t_{j+1}, \dots, t_{n-1} \rangle$.

We now define functions $\text{inj} : [\underline{n}^{m+1} \Rightarrow \underline{0}] \rightarrow [\underline{m+1} \times (\underline{n}^m \Rightarrow \underline{0})^n]$:

- $\text{inj}(f) = \top$, if $f = \top$,
- $\text{inj}(f) = \perp$, if $f = \perp$,
- $\text{inj}(f) = \langle i, \langle g_j \mid j < n \rangle \rangle$, where $g_j(e) = f(e[j]_i)$, if $\text{catch}_{m+1}(f) = i$.

and $\text{proj} : [\underline{m+1} \times (\underline{n}^m \Rightarrow \underline{0})^n] \rightarrow [\underline{n}^{m+1} \Rightarrow \underline{0}]$:

- $\text{proj}(\langle a, e \rangle)(d) = a$, if $a \in \{\top, \perp\}$

- $\text{proj}(\langle i, d \rangle)(e) = \top$ if $\pi_i(e) = \top$, or $\pi_i(e) = j$ and $\pi_j(d)(\lceil e \rceil_i) = \top$,
- $\text{proj}(\langle i, d \rangle)(e) = \perp$, otherwise.

Lemma 3.3. For all f , $\text{inj}(\text{proj}(f)) = f$.

Proof:

If $f \in \{\top, \perp\}$ then $\text{inj}(\text{proj}(f)) = f$ as both inj and proj are strict. If f is i -strict then $\text{inj}(f) = \langle i, \langle g_j \mid j < n \rangle \rangle$, where $g_j(e) = f(e \lfloor j \rfloor_i)$. So for any $e \in \underline{n}^{m+1}$, if $\text{proj}(\text{inj}(f))(e) = \top$ then by definition of $\text{proj}(\langle i, \langle g_j \mid j < n \rangle \rangle)(e)$, either $\pi_i(e) = \top$ (and hence $f(e) = \top$ by i -strictness) or $\pi_i(e) = j$ and $f(\lceil e \rceil_i \lfloor j \rfloor_i) = f(e) = \top$. Similarly, if $\text{proj}(\text{inj}(f))(e) = \perp$ then $f(e) = \perp$ and so $\text{proj}(\text{inj}(f)) = f$ as required. \square

Lemma 3.4. For any $n > 0$, $\underline{n}^{m+1} \Rightarrow \underline{0} \trianglelefteq \underline{m+1} \times (\underline{n}^m \Rightarrow \underline{0})^n$.

Proof:

inj is definable as the term

$$\lambda f. \text{case} \langle \text{catch}_{m+1} f, \langle \langle j, \langle \lambda x. f(\langle \pi_0(x), \dots, \pi_{m-1}(x) \rangle \lfloor j \rfloor_i) \mid i \leq m \rangle \mid j < n \rangle \rangle$$

and proj is definable as

$$\lambda xy. \text{case} \langle \pi_1(x), \text{case} \langle \pi_i(y), \langle \pi_j(\pi_2(x)) \lceil \langle \pi_0(y), \dots, \pi_m(y) \rangle \rfloor_j \mid j < n \mid i \leq m \rangle \rangle \rangle$$

\square

Proposition 3.1. For all $n > 0$, there is a definable retraction $\underline{n}^m \Rightarrow \underline{0} \trianglelefteq \underline{m}^{n^m \cdot (m+1)}$.

Proof:

By induction on m : for the base case, $\underline{n}^m \Rightarrow \underline{0} = I \Rightarrow \underline{0} \cong \underline{0} = \underline{0}^{n^0 \cdot 1}$. For the induction case:

$$\begin{aligned} \underline{n}^{m+1} \Rightarrow \underline{0} &\trianglelefteq \underline{m+1} \times (\underline{n}^m \Rightarrow \underline{0})^n \text{ by Lemma 3.4,} \\ &\trianglelefteq \underline{m+1} \times (\underline{m}^{(n^m) \cdot (m+1)})^n \text{ by induction hypothesis,} \\ &\trianglelefteq \underline{m+1} \times \underline{m+1}^{(n^{m+1}) \cdot (m+1)} \\ &\trianglelefteq \underline{m+1}^{n^{m+1} \cdot (m+2)} \text{ as required.} \end{aligned}$$

\square

Proposition 3.2. For every type T there exist integers $n(T), m(T)$ such that $T \trianglelefteq \underline{n(T)}^{m(T)}$.

Proof:

By induction on type-structure: for example, if $T = R \Rightarrow S$, then $R \Rightarrow S \trianglelefteq \underline{n(R)}^{m(R)} \Rightarrow \underline{n(S)}^{m(S)}$

$$\begin{aligned} &\cong \underline{n(R)}^{m(R)} \Rightarrow (\underline{0}^{n(S)} \Rightarrow \underline{0})^{m(S)} \\ &\cong (\underline{0}^{n(S)} \Rightarrow \underline{n(R)}^{m(R)} \Rightarrow \underline{0})^{m(S)} \\ &\trianglelefteq (\underline{0}^{n(S)} \Rightarrow \underline{m(R)}^{n(R)^{m(R)} \cdot (m(R)+1)})^{m(S)} \\ &\cong (\underline{n(S)} + \underline{m(R)})^{n(R)^{m(R)} \cdot (m(R)+1) \cdot m(S)}. \end{aligned}$$

\square

Theorem 3.1. Every term of bounded SPCF is observationally equivalent to a term of ASPCF.

Proof:

Given any (closed) SPCF term $M : T$, by Proposition 3.2 there exist n, m such that $T \trianglelefteq \underline{n}^m$. For each $i < m$, $\pi_i(\text{inj } M)$ either diverges or reduces to a numeral, and hence by soundness and adequacy, $\pi_i(\llbracket \text{inj } M \rrbracket)$ is either a number or \perp . So $\llbracket \text{inj } M \rrbracket$ is definable as a term $N : \underline{n}^m$ of ASPCF (a tuple composed of numerals and instances of \perp). Hence $\llbracket \text{proj } N \rrbracket = \llbracket \text{inj } (\text{proj } M) \rrbracket = \llbracket M \rrbracket$, and so by soundness of the bistable model, M is observationally equivalent to the ASPCF term $\text{proj } N$. \square

Note that we have not used the fact that our model is fully abstract — in fact, inequational completeness (and hence full abstraction) is a straightforward consequence of Proposition 3.2.

Proposition 3.3. $M \lesssim N$ implies $\llbracket M \rrbracket \sqsubseteq \llbracket N \rrbracket$.

Proof:

Suppose $\llbracket M \rrbracket \not\sqsubseteq \llbracket N \rrbracket$, then $\llbracket \text{inj } M \rrbracket \not\sqsubseteq \llbracket \text{inj } N \rrbracket$. So there exists $i < m$ such that $\pi_i(\llbracket \text{inj } M \rrbracket) \not\sqsubseteq \pi_i(\llbracket \text{inj } N \rrbracket)$. So by soundness and computational adequacy of the model, there is some value v such that $\pi_i(\text{inj } M) \Downarrow v$ and $\pi_i(\text{inj } N) \not\Downarrow v$. \square

4. Unbounded SPCF: Recursion Versus Iteration

We now extend our results to a version of SPCF over the natural numbers with recursively defined functions. In addition to eliminating explicit instances of sharing which violate the affine typing discipline, we now need to remove those which are introduced by recursive definition, and replace them with *iteration*. We shall present these results in a call-by-value setting (a proof for unbounded call-by-name SPCF was given in [15]), showing the flexibility of our approach vis-a-vis evaluation order and bringing it closer to languages such as ML and Scheme; alternatively one could work in a language unifying call-by-name and call-by-value, such as call-by-push-value [18], or a target language for linear CPS translation [4] (Affine SPCF may be CPS-translated into an affine calculus allowing contraction on the “answer type” only). One complication arising in the unbounded case is the lack of infinitary products in SPCF, specifically nat -indexed products of the type T , which is equivalent to the type of strict functions from nat into T . In the call-by-name case [15], it was necessary to add such a type to the language; in the call-by-value case, the function-type $\text{nat} \rightarrow T$ is interpreted as a strict function space, but we need an additively typed “sequential composition” operator to capture the correspondence with additive products.²

We form unbounded, call-by-value SPCF, or SPCF_v , by extending the simply-typed λ -calculus (without products) over types generated from $\underline{0}$ (empty) and nat :

$$S, T ::= \underline{0} \mid \text{nat} \mid S \Rightarrow_v T$$

with the following constants and operations:

Conditional From $M : \text{nat}$, $N_1, N_2 : T$, form $\text{If0 } M \text{ then } N_1 \text{ else } N_2 : T$

²We have omitted products from SPCF_v because it is possible to do so and thus keep the language as simple as possible — including products presents no problem for the proof that $\text{nat} \Rightarrow_v \text{nat}$ is a universal type, since we have a retraction $(\text{nat} \Rightarrow_v \text{nat}) \times (\text{nat} \Rightarrow_v \text{nat}) \trianglelefteq \text{nat} \Rightarrow_v (\text{nat} \Rightarrow_v \text{nat}) \trianglelefteq \text{nat} \Rightarrow_v \text{nat}$.

Recursion Fixpoint combinators $Y : (T \Rightarrow_v T) \Rightarrow_v T$ for each function-type T .

Numerals Constants $0 : \text{nat}$ and $\text{succ} : \text{nat} \Rightarrow_v \text{nat}$.

Control A control operator $\text{catch} : (\text{nat} \Rightarrow_v \underline{0}) \Rightarrow_v \underline{0} \Rightarrow_v \underline{\text{nat}}$ — with which to declare labels to which a program may subsequently jump back: having declared a label k in M with $\text{label } \lambda k.M$, whenever we call k with a value v we exit M with the value v . We may think of this as a “first-order” version of Felleisen’s *idealized call/cc* [8] $\mathcal{C} : ((T \Rightarrow_v \underline{0}) \Rightarrow_v \underline{0}) \Rightarrow_v T$.

We also add the following operations and constants, which are macro-expressible in SPCF_v , but not in the affine version of the language.

Sequential Composition From $M : \text{nat}$, and $N : \text{nat} \Rightarrow_v T$, form $M; N : T$. (this is equivalent to $(\lambda x.N x) M$, but the latter is not always affinely typable).

Iterated Sequential Composition A constant $\text{it} : (\text{nat} \Rightarrow_v \text{nat}) \Rightarrow_v \text{nat} \Rightarrow_v \underline{0}$. We shall write M^* for $\text{it } M : \text{nat} \Rightarrow \underline{0}$, which is the program which evaluates its argument to a value, applies M to the result, evaluates to a value, applies M to the result and so on. (Thus $M \text{ n}$ may be typed with the empty type as it never returns a value.) it is expressible in SPCF_v as $\lambda f.Y\lambda g.\lambda x.(f x); g$.

Injective pairing An injective pairing function $\text{pair} : \text{nat} \Rightarrow_v \text{nat} \Rightarrow_v \text{nat}$ and projections $\text{fst}, \text{snd} : \text{nat} \Rightarrow_v \text{nat}$. In SPCF pair is expressible as the term computing the partial function $\text{pair}(m)(n) = 2^m \cdot 2n + 1$, and fst and snd as the terms computing the (partial) functions fst, snd such that $\text{fst}(2^m \cdot 2n + 1) = m$ and $\text{snd}(2^m \cdot 2n + 1) = n$.

The small-step operational semantics for SPCF_v programs (closed terms of type nat) is given in Table 3. $E[_]$ ranges over the following evaluation contexts:

$$E[_] ::= [_] \mid E[_] M \mid V E[_] \mid \text{If0 } E[_] \text{ then } M_1 \text{ else } M_2 \mid E[_]; N \mid \text{label } \lambda k.E[_]$$

where values V are variables, numerals, constants and λ -abstractions — i.e. given by the grammar:

$$V ::= x \mid C \mid \text{succ } V \mid \text{pair } n \mid \lambda x.M$$

where C ranges over all constants.

4.1. Denotational Semantics

We interpret SPCF_v in the category of *bistable bicpos* and bistable and continuous functions.

Definition 4.1. Let D be a bistable biorder. Given \sqsubseteq -directed sets $X, Y \subseteq D$, we say that $X \uparrow Y$ if for all $x \in X$ and $y \in Y$ there exists $x' \in X$ and $y' \in Y$ such that $x \sqsubseteq x', y \sqsubseteq y'$ and $x' \uparrow y'$.

D is a *bistable bicpo* if $(|D|, \sqsubseteq)$ is a cpo and if $X \uparrow Y$ then $\bigsqcup X \uparrow \bigsqcup Y$ and $\bigsqcup X \wedge \bigsqcup Y = \bigsqcup \{x \wedge y \mid x \in X \wedge y \in Y \wedge x \uparrow y\}$

Proposition 4.1. Bistable bicpos and bistable and continuous functions form a cpo-enriched biCCC, \mathcal{B} .

Proof:

This follows the proof of Cartesian closure for the category of bistable biorders (for details, we refer to [13, 12]). We define the least upper bounds of directed sets of functions pointwise — i.e. $(\bigsqcup F)(x) = \bigsqcup \{f(x) \mid f \in F\}$ — and use the property that $X \uparrow Y$ implies $\bigsqcup X \uparrow \bigsqcup Y$ and $\bigsqcup X \wedge \bigsqcup Y = \bigsqcup \{x \wedge y \mid x \in X \wedge y \in Y \wedge x \uparrow y\}$ to prove that this defines a bistable function. \square

$E[(\lambda x.M) V]$	\longrightarrow	$E[M[V/x]]$
$E[\text{If } 0 \text{ then } M \text{ else } N]$	\longrightarrow	$E[M]$
$E[\text{If } 0 \text{ (succ } n \text{) then } M \text{ else } N]$	\longrightarrow	$E[N]$
$E[\text{label } \lambda k.E'_k[k \ n]]$	\longrightarrow	$E[n]$
$E[Y V]$	\longrightarrow	$E[V \ \lambda x.(Y V) \ x]$
$E[n; N]$	\longrightarrow	$E[N \ n]$
$E[(\text{pair } m) \ n]$	\longrightarrow	$E[2^m \cdot 2n + 1]$
$E[\text{fst } 2^m \cdot 2n + 1]$	\longrightarrow	$E[m]$
$E[\text{snd } 2^m \cdot 2n + 1]$	\longrightarrow	$E[n]$
$E[\text{it } V]$	\longrightarrow	$E[\lambda x.(V \ x); (\text{it } V)]$

Table 3. “Small-step” operational semantics for SPCF_v programs.

To interpret the call-by-value function-types, we use the bilifting operation, which acts as a monad on \mathcal{B} . Let \mathcal{B}_S be the category of pointed bistable bicpos and strict bistable and continuous functions.

Proposition 4.2. The inclusion of \mathcal{B}_S into \mathcal{B} has a left adjoint, $(-)_\perp^\top$.

Proof:

The unit of the adjunction $\eta : A \rightarrow A_\perp^\top$ sends x to $\text{in}_1(x)$. For any $f : A \rightarrow B$, where B is pointed, $\bar{f} : A_\perp^\top \rightarrow B$ is the strict function such that $f(\text{in}_1(x)) = f(x)$. \square

Thus we give a semantics of the call-by-value λ -calculus in \mathcal{B} by interpreting function types $S \Rightarrow_v T$ as $\llbracket S \rrbracket \Rightarrow \llbracket T \rrbracket_\perp^\top$ and the ground types $\mathbb{0}, \text{nat}$ as the corresponding flat bistable bicpos ($\tilde{\mathcal{O}}$ and $\tilde{\mathbb{N}}$).

As in the bounded case, we define the interpretation of the control operator `label` via the observation that bistable functions are strongly sequential. Noting that $\tilde{\mathbb{N}} \Rightarrow D \cong \prod_{i \in \mathbb{N}} D$, if D, E are pointed we say that a function $f : (\tilde{\mathbb{N}} \Rightarrow D) \rightarrow E$ is i -strict if $g(i) = \perp$ implies $f(g) = \perp$ and $g(i) = \top$ implies $f(g) = \top$.

Lemma 4.1. If A is pointed then every strict, continuous and bistable function $f : (\tilde{\mathbb{N}} \Rightarrow A) \rightarrow \Sigma$ is i -strict for some i .

Proof:

Given $i \in \mathbb{N}$, $a \in A$, and $e \in \tilde{\mathbb{N}} \Rightarrow A$, let $e[i, a] \in \tilde{\mathbb{N}} \Rightarrow A$ denote the function defined by:

- $e[i, a](i) = a$,
- $e[i, a](n) = e(n)$, if $n \neq i$.

Then $\perp[i, \top] \uparrow \perp[j, \top]$ for all i, j , and so by continuity and bistability, $f(\bigvee \{\perp[i, \top] \mid i \in \mathbb{N}\}) = \bigvee \{f(\perp[i, \top]) \mid i \in \mathbb{N}\}$, and so $f(\perp[i, \top]) = \top$ for some i , and $e(i) = \top$ implies $f(e) \sqsupseteq f(\perp[i, \top]) = \top$. $\top[i, \perp] \uparrow \perp[i, \top]$, and so $f(\top[i, \perp]) \wedge f(\perp[i, \top]) = f(\top[i, \perp] \wedge \perp[i, \top]) = f(\perp) = \perp$, and so $f(\top[i, \perp]) = \perp$, and $e(i) = \perp$ implies $f(e) \sqsubseteq f(\top[i, \perp]) = \perp$. \square

$\overline{\Gamma, x:S \vdash x:S}$	$\overline{\Gamma \vdash C:T}$ C a constant of type T	$\frac{\Gamma \vdash M:S \Rightarrow_v T \quad \Delta \vdash N:S}{\Gamma, \Delta \vdash M N:T}$
$\frac{\Gamma, x:S \vdash M:T}{\Gamma \vdash \lambda x.M:S \Rightarrow_v T}$	$\frac{\Gamma \vdash M:\text{nat} \quad \Gamma \vdash N_1:T \quad \Gamma \vdash N_2:T}{\Gamma \vdash \text{If } 0 \ M \ \text{then } N_1 \ \text{else } N_2:T}$	$\frac{\Gamma \vdash M:\text{nat} \quad \Gamma \vdash N:\text{nat} \Rightarrow_v T}{\Gamma \vdash M;N:T}$

Table 4. Typing judgements for ASPCF_v

Hence we may interpret the constant $\text{label} : ((\text{nat} \Rightarrow_v T) \Rightarrow_v \perp) \Rightarrow_v \text{nat}$ as the strict function catch_ω sending each i -strict function to the value i . We also note that an instance of catch_ω is an inverse to the strict function from $\tilde{\mathbb{N}}_\perp^\top$ to $(\tilde{\mathbb{N}} \Rightarrow \Sigma) \Rightarrow \Sigma$ sending n to $\lambda f.f(n)$, which is therefore an isomorphism.

Corollary 4.1. $\tilde{\mathbb{N}}_\perp^\top \cong (\tilde{\mathbb{N}} \Rightarrow \Sigma) \Rightarrow \Sigma$.

The interpretation of the numerals, conditionals and fixpoints is standard for call-by-value PCF, yielding sound interpretations of the additional constants. We thus prove inequational soundness of the model by establishing soundness of the reduction rule for label (as in the bounded case), and giving a standard proof of computational adequacy.

Proposition 4.3. If $\llbracket M \rrbracket \sqsubseteq \llbracket N \rrbracket$ then $M \lesssim N$.

4.2. Affine SPCF_v

As in the call-by-name case, we eliminate interleaved threads of computation using an affine typing system. Since we do not include products, we give explicit typing rules for the conditional and sequential composition operations which allow sharing of variables between the component terms. The Y combinator is not included in ASPCF_v — in general, evaluating it creates nested function calls and violates subject reduction. Various possibilities exist for allowing limited forms of recursion which avoid this problem — we could require that Y is applied only to *closed* terms, for example (although Y still effectively creates nested calls to its argument), or describe a tail-recursion scheme for SPCF . Either of these can be used to express iterated sequential composition, and so we simply restrict recursion to the latter form. Thus the terms of ASPCF_v are those terms of $\text{SPCF}_v - \{Y\}$ which are well-typed according to the rules in Table 4. We write $M * N$ for $N; (\text{pair } M)$, allowing the sharing of variables between M and N .

5. Expressiveness of ASPCF_v

We first note that iteration, jumps, and the injective pairing operation $_ * _$ allow us to define all partial recursive functions in ASPCF_v .

Lemma 5.1. Every partial recursive function $f : \mathbb{N}^k \rightarrow \mathbb{N}$ is ASPCF_v -definable — i.e. there is a term $x_1 : \text{nat}, \dots, x_k : \text{nat} \vdash M_f : \text{nat}$ such that $\llbracket M_f \rrbracket(\vec{n}) = \perp$ if $f(\vec{n}) \uparrow$ and $\llbracket M_f \rrbracket(\vec{n}) = \text{in}_1(f(\vec{n}))$, otherwise.

Proof:

Projection, composition and successor functions are clearly definable.

Suppose $f = \mu x.g(x, \vec{y}) = 0$. Then $M_f = \text{label } \lambda k.0; (\lambda x.\text{If } 0 M_g \text{ then } k x \text{ else succ } x)^*$.

Suppose $f(\vec{y}, n) = g(\vec{y})$ and $f(\vec{y}, n+1) = h(f(\vec{y}, n), \vec{y}, n)$.

Then $M_f = \text{label } \lambda k.((0 * M_g); \lambda x.\text{If } \text{fst}(x) = n \text{ then } (k \text{ snd}(x)) \text{ else } \text{succ}(x) * (((\lambda uv.M_h) \text{fst}(x)) \text{snd}(x)))^*$. \square

As in the bounded case, we show that every term is denotationally equivalent to one definable in ASPCF_v by proving that every type is an ASPCF_v -definable retract of a type of low order. In this case this is the single type $\text{nat} \Rightarrow_v \text{nat}$, which is therefore a *universal* type for ASPCF_v as it is for SPCF_v [20] (we have observed that its denotation $\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}$ is equivalent to an infinite cartesian product of copies of $\tilde{\mathbb{N}}_{\perp}^{\top}$).

The proof of universality is based on a definable retraction $(\text{nat} \Rightarrow_v \text{nat}) \Rightarrow_v \underline{0} \trianglelefteq \text{nat} \Rightarrow_v \text{nat}$. In SPCF , such a retraction can be defined as the limit of a series of approximants defined by extracting strictness indices as in Lemma 3.4. However, to define this retraction (in this way) in SPCF_v requires the use of the Y combinator. To show that $(\text{nat} \Rightarrow_v \text{nat}) \Rightarrow_v \underline{0} \trianglelefteq \text{nat} \Rightarrow_v \text{nat}$ using only iteration, we use the fact that each observably sequential (i.e bistable) function f from $\tilde{\mathbb{N}} \Rightarrow (\tilde{\mathbb{N}}_{\perp}^{\top})$ to Σ may be represented as a *strategy* for a two-player game in which players alternately choose natural numbers (or terminate the game by choosing \top or \perp). The strategy of f describes how the application of f to an argument in $g \in \tilde{\mathbb{N}} \Rightarrow (\tilde{\mathbb{N}}_{\perp}^{\top})$ is sequentially evaluated — first the strategy chooses a number n , then the other player replies with $g(n)$, and so on until either player plays \top or \perp , representing the value of $f(g)$. If an infinite sequence of values is generated then $f(g) = \perp$.

Definition 5.1. A strategy is a partial function $\sigma : (\widetilde{\mathbb{N} \times \mathbb{N}})^* \rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}$ (giving the response of “Player one” to each finite sequence of pairs of moves).

Note that a strategy may contain responses for positions which are not reachable in the evaluation of any argument. (The representation will be used to define a retraction rather than an isomorphism.)

Definition 5.2. Given a finite sequence $s \in (\mathbb{N} \times \mathbb{N})^*$ and $e : \tilde{\mathbb{N}} \rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}$, we define $e[s] \in \tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}$: $e[\varepsilon] = e$ and $e[s(i, j)] = e[s][i, j]$ (the operation $_{-}[i, j]$ is defined in the proof of Lemma 4.1).

Proof of the following lemma by induction on sequence length is straightforward.

Lemma 5.2. For any $e, e' \in \tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}$ and $s \in (\widetilde{\mathbb{N} \times \mathbb{N}})^*$, $e \sqsubseteq e'$ implies $e[s] \sqsubseteq e'[s]$, and if $e \uparrow e'$, then $e[s] \uparrow e'[s]$, $e'[s] \wedge e[s] = (e \wedge e')[s]$ and $e[s] \vee e'[s] = (e \vee e')[s]$.

Hence if $f : (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}) \rightarrow \Sigma$ is continuous and bistable, then for any $s \in (\mathbb{N} \times \mathbb{N})^*$, $\lambda x.f(x[s])$ is continuous and bistable, and we may define a strategy for f : $\text{strat}(f)(s) = \text{catch}_{\omega}(\lambda e.f(e[s]))$.

From each strategy κ , we may recover a function $\text{fun}(\kappa) : (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}) \rightarrow \Sigma$ by reconstructing a series of *sequentializations* for each argument.

Definition 5.3. Given a strategy κ and an element $e \in \tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}$, we define a sequence of elements $\text{seq}_i(\kappa, e) \in ((\mathbb{N} \times \mathbb{N})^*)_{\perp}^{\top}$ by $\text{seq}_0(\kappa, e) = \varepsilon$ and:

- $\text{seq}_{n+1}(\kappa, e) = \top$, if $\text{seq}_n(\kappa, e) = \top$ or $(\text{seq}_n(\kappa, e) = s \text{ and } \kappa(s) = \top)$ or $(\kappa(s) = i \text{ and } e(i) = \top)$,
- $\text{seq}_{n+1}(\kappa, e) = s(i, j)$ if $\text{seq}_n(\kappa, e) = s$, $\kappa(s) = i$ and $e(i) = j$,
- $\text{seq}_{n+1}(\kappa, e) = \perp$, otherwise.

Thus we define $\text{fun}(\kappa) : (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}) \rightarrow \Sigma$:

- $\text{fun}(\kappa)(e) = \top$ if there exists n such that $\text{seq}_n(\kappa, e) = \top$,
- $\text{fun}(\kappa)(e) = \perp$, otherwise.

We may now prove that $\text{fun}(\text{strat}(f)) = f$ for any f .

Lemma 5.3. If $\text{seq}_n(\kappa, e) = s$ then $e[s] = e$.

Proof:

By induction on n . If $\text{seq}_{n+1}(\kappa, e) = s(i, j)$ then $\text{seq}_n(\kappa, e) = s$ and $e(i) = j$. Hence $e[s(i, j)] = e[s][i, j] = e[i, j] = e$. \square

Lemma 5.4. If $f[s](i) \neq g[s](i)$ then $f[s](i) = f(i)$ and $f(i) \neq g(i)$.

Proof:

By induction on the length of s . \square

Lemma 5.5. If $\lambda x.f(x[s])$ is i -strict for some f then $\perp[s](i) = \perp$.

Proof:

If $\lambda x.f(x[s])$ is i -strict then $f(\perp[s]) = \perp$ and $f(\perp[i, \top][s]) = \top$. Hence $\perp[s] \neq \perp[i, \top][s]$ — i.e. there exists j such that $\perp[s](j) \neq \perp[i, \top][s](j)$. So by Lemma 5.4, $\perp[s](j) = \perp(j) = \perp$, and $\perp(j) \neq \perp[i, \top](j)$, and so $i = j$ as required. \square

We will also use the fact that $\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}$ is algebraic, and every compact element is finite (i.e. dominates finitely many elements).

Lemma 5.6. Every element of $\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}$ is the least upper bound of a directed set of finite elements.

Proof:

For any $f \in \tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}$, define $f_i(n) = f(n)$ if $n \leq i$, and $f_i(n) = \perp$, otherwise. Then each f_i dominates finitely many elements, and $f = \bigsqcup_{i \in \omega} f_i$ as required. \square

Lemma 5.7. If $e \in \tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}$ is finite then there exists $n \in \mathbb{N}$ such that $\text{seq}_n(\text{strat}(f), e) \in \{\top, \perp\}$.

Proof:

Suppose $\text{seq}_n(\text{strat}(f), e) \notin \{\top, \perp\}$, for all n . Then for all n , $\text{seq}_{n+1}(\text{strat}(f), e) = s(i, j)$, for some s, i, j such that $\text{seq}_n(\text{strat}(f), e) = s$ and $\text{strat}(f)(s) = i$. So $\lambda x.f(x[s])$ is i -strict, and so by Lemma 5.5 $\perp[s](i) = \perp$. Hence $\perp[s] \sqsubset \perp[s][i, j] = \perp[\text{seq}_{n+1}(\text{strat}(f), e)]$. Thus $\langle \perp[\text{seq}_n(\text{strat}(f), e)] \rangle_n$ forms a strictly increasing infinite sequence of elements bounded above by e , contradicting finiteness of e . \square

Lemma 5.8. For any f , $\text{fun}(\text{strat}(f)) = f$.

Proof:

Suppose $\text{fun}(\text{strat}(f))(e) = \top$. There exists a smallest n such that $\text{seq}_{n+1}(\text{strat}(f), e) = \top$. Then $\text{seq}_n(\text{strat}(f), e) = s$ and either $\text{strat}(f)(s) = \top$ or $\text{strat}(f)(s) = i$ and $\pi_i(e) = \top$. In the former case, we have $\top = (\lambda x.f(x[s])) \perp = f(\perp[s]) \sqsubseteq f(e[s]) = f(e)$. In the latter case $\lambda x.f(x[s])$ is i -strict, and $e(i) = \top$, and so $\top = (\lambda x.f(x[s])) e = f(e[s]) = f(e)$.

To prove the converse, suppose $f(e) = \top$. Then by continuity and algebraicity there exists a finite element $e' \sqsubseteq e$ such that $f(e') = \top$. By Lemma 5.7, there exists (a smallest) n such that $\text{seq}_{n+1}(\text{strat}(f), e') = \top$ or $\text{seq}_{n+1}(\text{strat}(f), e) = \perp$. In the former case, $\text{fun}(\text{strat}(f))(e) = \top$ as required. In the latter case, $\text{seq}_n(\text{strat}(f), e') = s$ and either $\text{strat}(f)(s) = \perp$ (so $(\lambda x.f(x[s])) \top = \perp$) — but this contradicts $\top = f(e') = f(e'[s]) \sqsubseteq f(\top[s])$ — or else $\text{strat}(f)(s) = i$ (so $\lambda x.f(x[s])$ is i -strict) and $e'(\perp) = \perp$ — but then $e'[s](i) = e'(i) = \perp$ and so $f(e') = \perp$, which is also a contradiction. \square

We may prove directly that strat and fun are bistable functions (a more general relationship between strategies or sequential algorithms and bistable functions is studied in [16] and [13]). Here, we combine them with an encoding of finite lists in \mathbb{N} , using our injective pairing operation, to define a retraction from $(\text{nat} \Rightarrow_v \text{nat}) \Rightarrow_v \underline{0}$ to $\text{nat} \Rightarrow_v \text{nat}$ in ASPCF_v . We define the function $\phi : (\mathbb{N} \times \mathbb{N})^* \rightarrow \mathbb{N}$:

- $\phi(\varepsilon) = 0$, and
- $\phi(s(i, j)) = \text{pair}(\phi(s), \text{pair}(i, j))$.

and the right-inverse to ϕ :

- $\phi^{-1}(0) = \varepsilon$
- $\phi^{-1}(n) = \phi^{-1}(\text{fst}(n))(\text{fst}(\text{snd}(n)), \text{snd}(\text{snd}(n)))$

Let $\text{eq} : \text{nat} \Rightarrow_v \text{nat} \Rightarrow_v \text{nat}$ be an ASPCF_v term such that $\llbracket (\text{eq } m) m \rrbracket = 0$ and $\llbracket \llbracket (\text{eq } m) n \rrbracket \rrbracket = 1$ if $m \neq n$.

Definition 5.4. Define the ASPCF_v term: $\text{sub}(g : \text{nat} \Rightarrow_v \text{nat}, k : \text{nat} \Rightarrow_v \underline{0}, x : \text{nat}) : \text{nat} \Rightarrow_v \underline{0} :$
 $\text{sub}(g, k, x) = \lambda v. \text{If } 0 \ v \ \text{then } k(g \ x) \ \text{else } \text{If } 0 \ (\text{eq } x) \ (\text{fst } v) \ \text{then } k \ \text{fst}(\text{snd } v) \ \text{else } \text{snd } v.$

Lemma 5.9. If $\phi(s) = n$ then $\llbracket \text{sub}^* \ n \rrbracket(g, k, m) = k(g[s](m))$

Proof:

By induction on the length of s . If $s = \varepsilon$ then $\phi(s) = 0$ and $\llbracket \text{sub}^* \ 0 \rrbracket(g, k, m) = \overline{\llbracket \text{sub}^* \rrbracket}(\llbracket \text{sub } 0 \rrbracket(g, k, m)) = \overline{\llbracket \text{sub} \rrbracket}(k(g(m))) = k(g[s](m))$.

Suppose $s = s'(i, j)$.

If $m = i$ then $\llbracket \text{sub } n \rrbracket(g, k, m) = k(j) = k(g[s](m))$ and $\llbracket \text{sub}^* \ n \rrbracket = \overline{\llbracket \text{sub}^* \rrbracket}(\llbracket \text{sub } n \rrbracket(g, k, m)) = k(g[s](m))$.

If $m \neq i$ then $\llbracket \text{sub } n \rrbracket(g, k, m) = l$ for some $l \in \mathbb{N}$ such that $l = \phi(s')$. So $\llbracket \text{sub}^* \ n \rrbracket(g, k, m) = \overline{\llbracket \text{sub}^* \rrbracket}(\llbracket \text{sub } n \rrbracket(g, k, m))(g, k, m) = \llbracket \text{sub}^* \ 1 \rrbracket(g, k, m) = k(g[s'](m)) = k(g[s](m))$. \square

Proposition 5.1. The map from $(\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}) \Rightarrow \Sigma$ to $\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}$ sending f to $\phi^{-1}; \text{strat}(f)$ is definable in ASPCF_v .

Proof:

Let $\text{strat} = \lambda f. \lambda y. \text{label } \lambda g. f (\lambda x. \text{label } \lambda k. (\text{sub}^* y))$. Then $\llbracket \text{strat} \rrbracket(f)(n) = \text{catch}_{\omega}(\lambda g. f \lambda x. g[\phi^{-1}(n)](x)) = \text{catch}_{\omega}(\lambda g. f g[\phi^{-1}(n)]) = \text{strat}(f)(\phi^{-1}(n))$ as required. \square

Definition 5.5. For each $n \in \omega$, we define terms $\text{seq}_n(x : \text{nat} \Rightarrow_v \text{nat}, y : \text{nat} \Rightarrow_v \text{nat}) : \text{nat}$ by $\text{seq}_0 = 0$ and $\text{seq}_{n+1} = \text{seq}_n; (\lambda u. u * ((x u) * (y (x u))))$

Lemma 5.10. For all n , $\llbracket \text{seq}_n \rrbracket(f, e) = \phi_{\perp}^{\top}(\text{seq}_n(\phi; f, e))$.

Proof:

By induction on n : $\llbracket \text{seq}_0 \rrbracket(f, e) = 0 = \phi(\varepsilon) = \phi_{\perp}^{\top}(\text{seq}_0(\phi; f, e))$.

If $\text{seq}_{n+1}(\phi; f, e) = s(i, j)$, then $\text{seq}_n(\phi; f, e) = s$, $f(\phi(s)) = i$ and $e(i) = j$. By induction hypothesis, $\llbracket \text{seq}_n \rrbracket = \phi(s)$, and so $\llbracket \text{seq}_n \rrbracket(f, e) = \text{pair}(\phi(s), \text{pair}(f(\phi(s)), e(f(\phi(s)))))) = \text{pair}(\phi(s), \text{pair}(i, j)) = \phi(s(i, j)) = \phi_{\perp}^{\top}(\text{seq}_{n+1}(\phi; f, e))$.

If $\text{seq}_{n+1}(\phi; f, e) = \top$ then either $\text{seq}_n(\phi; f, e) = \top$ — hence $\llbracket \text{seq}_n \rrbracket(f, e) = \top$ and so $\llbracket \text{seq}_{n+1} \rrbracket(f, e) = \top$ — or else $\text{seq}_n(\phi; f, e) = s$ and $f(\phi(s)) = \top$ — and so $\llbracket \text{seq}_n \rrbracket(f, e) = \phi(s)$ and $\llbracket \text{seq}_{n+1} \rrbracket(f, e) = \top$ — or $\text{seq}_n(\phi; f, e) = s$, $f(\phi(s)) = i$ and $e(i) = \top$ — so $\llbracket \text{seq}_n \rrbracket(f, e) = \phi(s)$ and $\llbracket \text{seq}_{n+1} \rrbracket(f, e) = \top$. Similarly, if $\text{seq}_{n+1}(\phi; f, e) = \perp$ then $\llbracket \text{seq}_{n+1} \rrbracket(f, e) = \perp$. \square

Proposition 5.2. The map from $\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}$ to $(\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}) \Rightarrow \Sigma$ sending f to $\text{fun}(\phi; f)$ is definable in ASPCF_v .

Proof:

Let $\text{fun} = \lambda xy. 0; (\lambda u. u * (x u) * (y (x u)))^*$. Then $\llbracket \text{fun} \rrbracket(\phi^{-1}; \kappa)(e) = \top$ if and only if there exists n such that $\llbracket \text{seq}_n \rrbracket(\phi^{-1}; \kappa, e) = \top$ if and only if there exists n such that $\text{seq}_n(\kappa, e) = \top$ if and only if $\text{fun}(\kappa)(e) = \top$. \square

Proposition 5.3. There is an ASPCF_v definable retraction $(\text{nat} \Rightarrow_v \text{nat}) \Rightarrow_v \underline{0} \trianglelefteq \text{nat} \Rightarrow_v \text{nat}$.

Proof:

For any $f \in (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}) \Rightarrow \Sigma$, $\llbracket \lambda x. \text{fun}(\text{strat } x) \rrbracket(f) = \text{fun}(\phi; \phi^{-1}; \text{strat}(f)) = \text{fun}(\text{strat}(f)) = f$. \square

Corollary 5.1. There is an ASPCF_v definable retraction $(\text{nat} \Rightarrow_v \text{nat}) \Rightarrow_v \text{nat} \trianglelefteq \text{nat} \Rightarrow_v \text{nat}$.

Proof:

$\llbracket (\text{nat} \Rightarrow_v \text{nat}) \Rightarrow_v \text{nat} \rrbracket = (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}) \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}$
 $\cong (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}) \Rightarrow (\tilde{\mathbb{N}} \rightarrow \Sigma) \Rightarrow \Sigma \cong (\tilde{\mathbb{N}} \Rightarrow \Sigma) \Rightarrow (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}) \Rightarrow \Sigma$
 $\trianglelefteq (\tilde{\mathbb{N}} \Rightarrow \Sigma) \Rightarrow (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top})$ by Proposition 5.3
 $\cong \tilde{\mathbb{N}} \Rightarrow (\tilde{\mathbb{N}} \Rightarrow \Sigma) \Rightarrow (\tilde{\mathbb{N}} \Rightarrow \Sigma) \Rightarrow \Sigma$
 $\trianglelefteq \tilde{\mathbb{N}} \Rightarrow (\tilde{\mathbb{N}} \Rightarrow \Sigma) \Rightarrow \Sigma \cong \tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top} = \llbracket \text{nat} \Rightarrow_v \text{nat} \rrbracket$.

The defining terms are thus: $\text{inj} = \lambda f. \lambda x. \text{label } \lambda k. k((\text{strat}(\lambda y. k((f y) * 1)) x) * 0)$,
 $\text{proj} = \lambda g. \lambda h. \text{label } \lambda k. \text{fun}(\lambda x. \text{If } 0(\text{snd}(x)) \text{ then } (k \text{fst}(x)) \text{ else } g \text{fst}(x))$. \square

To complete our proof that $\text{nat} \Rightarrow_v \text{nat}$ is a universal type, we require some further simple retractions, to deal with the liftings used to interpret call-by-value function types.

Lemma 5.11. $(\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top})_{\perp}^{\top}$ is a retract of $(\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top})$ (in \mathcal{B}).

Proof:

We define (bistable) maps $\text{inj} : (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top})_{\perp}^{\top} \rightarrow (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top})$ and $\text{proj} : (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}) \rightarrow (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top})_{\perp}^{\top}$:

- $\text{inj}(e) = e$, if $e \in \{\top, \perp\}$,
- $\text{inj}(\text{in}(g)) = g'$, where $g'(0) = 0$ and $g'(n+1) = g(n)$.
- $\text{proj}(f) = \text{in}(f')$, where $f'(n) = f(n+1)$, if $f(0) \in \tilde{\mathbb{N}}$,
- $\text{proj}(f) = f(0)$, otherwise.

□

We use this retraction to obtain the following definable retractions.

Lemma 5.12. For any type T there is an ASPCF_v definable retraction $T \Rightarrow_v (\text{nat} \Rightarrow_v \text{nat}) \trianglelefteq \text{nat} \Rightarrow_v (T \Rightarrow_v \text{nat})$.

Proof:

$$\begin{aligned} \llbracket T \Rightarrow_v (\text{nat} \Rightarrow_v \text{nat}) \rrbracket &= \llbracket T \rrbracket \Rightarrow (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top})_{\perp}^{\top} \\ &\trianglelefteq \llbracket T \rrbracket \Rightarrow \tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top} \text{ by Lemma 5.11,} \\ &\cong \tilde{\mathbb{N}} \Rightarrow \llbracket T \rrbracket \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top} \trianglelefteq \tilde{\mathbb{N}} \Rightarrow (\llbracket T \rrbracket \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top})_{\perp}^{\top} = \llbracket \text{nat} \Rightarrow_v (T \Rightarrow_v \text{nat}) \rrbracket. \end{aligned}$$

The defining terms are $\text{inj} = \lambda f. \lambda x. \lambda y. \text{If } 0 \ x \ \text{then } (f \ \text{fst}(y)); 0 \ \text{else } ((f \ y) \ (\text{pred } x))$ and $\text{proj} = \lambda g. \lambda a. ((g \ 0) \ a); \lambda b. ((g \ (\text{succ } b)) \ a)$. □

Lemma 5.13. There is an ASPCF_v definable retraction $\text{nat} \Rightarrow_v \text{nat} \Rightarrow_v \text{nat} \trianglelefteq \text{nat} \Rightarrow_v \text{nat}$.

Proof:

$$\begin{aligned} \llbracket \text{nat} \Rightarrow_v (\text{nat} \Rightarrow_v \text{nat}) \rrbracket &= \tilde{\mathbb{N}} \Rightarrow (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top})_{\perp}^{\top} \\ &\trianglelefteq \tilde{\mathbb{N}} \Rightarrow (\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}) \text{ by Lemma 5.11,} \\ &\trianglelefteq \tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}, \text{ since } \tilde{\mathbb{N}} \times \tilde{\mathbb{N}} \trianglelefteq \tilde{\mathbb{N}}. \end{aligned}$$

The defining terms are $\text{inj} = \lambda f. \lambda x. \text{If } 0 \ \text{snd}(x) \ \text{then } (f \ \text{fst}(x)); 0 \ \text{else } ((f \ \text{fst}(x)) \ \text{pred}(\text{snd}(x)))$ and $\text{proj} = \lambda g. \lambda y. (g \ (y * 0)); \lambda z. g \ (y * z)$. □

Proposition 5.4. Every type of SPCF_v is an ASPCF_v -definable retract of $\text{nat} \Rightarrow_v \text{nat}$.

Proof:

By induction on type-structure. For function-types $S \Rightarrow_v T$ we have:
 $S \Rightarrow_v T \trianglelefteq (\text{nat} \Rightarrow_v \text{nat}) \Rightarrow_v (\text{nat} \Rightarrow_v \text{nat})$ by induction hypothesis,
 $\trianglelefteq \text{nat} \Rightarrow_v (\text{nat} \Rightarrow_v \text{nat}) \Rightarrow_v \text{nat}$ by Lemma 5.12,
 $\trianglelefteq \text{nat} \Rightarrow_v \text{nat} \Rightarrow_v \text{nat}$ by Corollary 5.1,
 $\trianglelefteq \text{nat} \Rightarrow_v \text{nat}$ by Lemma 5.13. □

Theorem 5.1. Every SPCF_v term $M : T$ is observationally equivalent to a term of ASPCF_v .

Proof:

As in the finite case, we apply the definable retraction, to obtain an element $\text{inj}(\llbracket M \rrbracket)$ of $\tilde{\mathbb{N}} \Rightarrow \tilde{\mathbb{N}}_{\perp}^{\top}$ — a function from $\tilde{\mathbb{N}}$ to $\tilde{\mathbb{N}}_{\perp}^{\top}$. Since $\text{inj}(\llbracket M \rrbracket)(n) = \llbracket (\text{inj } M) \mathbf{n} \rrbracket$ and $(\text{inj } M) \mathbf{n}$ either diverges or conevrgees to a numeral, $\text{inj}(\llbracket M \rrbracket)(n) \not\approx \top$ for all n by soundness and adequacy — i.e. $\text{inj}(\llbracket M \rrbracket)$ corresponds to a partial function from $\tilde{\mathbb{N}}$ to $\tilde{\mathbb{N}}$.

Informally, it is clear that $\text{inj}(\llbracket M \rrbracket)$ corresponds to a partial *recursive* function, since it is effectively computable (by the associated term $\text{inj } M$ of SPCF). More formally, Kanneganti and Cartwright [1] have defined a notion of computable observably sequential functional which correspond to the functionals computable in SPCF , and, at type $\text{nat} \Rightarrow_v \text{nat}$, are simply partial recursive functions.

Hence by Lemma 5.1, $\text{inj}(\llbracket M \rrbracket)$ is the denotation of a term $N : \text{nat} \Rightarrow_v \text{nat}$ of ASPCF_v . Hence $\llbracket M \rrbracket = \llbracket \text{proj } N \rrbracket$, and by soundness of the bistable model, M is observationally equivalent to the ASPCF_v term $\text{proj } N$. \square

As in the bounded case, full abstraction of our denotational semantics follows from the existence of a definable retraction into $\text{nat} \Rightarrow_v \text{nat}$.

Proposition 5.5. The bistable bicpo semantics of SPCF_v is fully abstract.

Proof:

As for Proposition 3.3. \square

6. Conclusions and Further Directions

In this paper, we have described an example of the use of a denotational model to eliminate nesting in an applied λ -calculus with control operators. It remains to be seen whether this result has any application in the design, implementation or analysis of functional programming languages. In general, elimination of nesting will reduce the space required for program evaluation in a stack-based implementation, since the size of the allocation stack for an ASPCF program can be bounded in the size of the types of the variables. However, this comes at the cost of increase in the size of the program itself (unless the original program is unnecessarily large). Similarly, with regard to time-efficiency, we may observe that optimal reduction techniques such as graph reduction are much easier to apply to terms without nesting, but this must be set against their (sometimes vastly) increased size. One possibility would be to apply de-nesting locally, to eliminate particular nested calls or recursive definitions, as these may only occur in a small fragment of a program.

If the priority is reasoning correctly about programs, rather than evaluating them efficiently, then ASPCF does appear to offer potential advantages. For example, *unification* of affinely typed terms is much simpler than the general case.

We have discussed the the expressiveness of affinely typed PCF and SPCF ; a natural question concerns how other side effects affect these results. (Or, in other words, is SPCF essentially unique in allowing this kind of de-nesting?) Preliminary results suggest that whilst our proofs rely on some properties are unusual, it is possible to build up bidomain models of languages with features such as recursive types

and non-determinism [17], with definable retractions into first-order types which are affinely typable, meaning that the results described here would apply.

The case of affine PCF extended with (integer) store (or affine Idealized Algol) is of independent interest, since this is the core of Reynolds' *Syntactic Control of Interference* language, in which the affine type system has a strong justification in terms of the prevention of covert interference between procedures. In the presence of effects, such as state, which allow nested function calls to be observed, nesting-elimination is not possible in the strong sense described here (up to observational equivalence). We also note that decidability results for observational equivalence in Idealized Algol and Syntactic Control of Interference expose their different levels of expressive power: the former is undecidable at order 4 and above [21], the latter at all orders [14]. Similar results apply when the languages are extended with catch-style control operators [14].

On the other hand, basic SCI with control may be considered to be more expressive than SPCF, since it is a proper extension of affine SPCF. As well as having good algorithmic properties (decidability of equivalence at all orders) this language retains the advantage of using state in an interference-controlled form — access to state allows many functions to be written more efficiently in an affine type system — for example, the retractions defined in this paper.

Acknowledgements

Thanks to the referees for several useful comments, in particular to Paul B. Levy for a new proof of the transitivity of \Downarrow .

References

- [1] , R. Cartwright, R. K.: What is a Universal Higher-Order Programming Language, *Proceedings of the International Conference on Automata, Languages and Programming, 1993*, 1993.
- [2] Abramsky, S.: Beyond Full Abstraction: Model Checking For Algol-like languages, 2001, Lecture notes from the Marktoberdorf summer school.
- [3] Asperti, A.: Light Affine Logic, *Proceedings of LICS '98*, IEEE press, 1998.
- [4] Berdine, J., O'Hearn, P., Reddy, U., Thielecke, H.: Linear Continuation-Passing, *Higher Order Symbolic Computation*, **15**(2/3), September 2002, 181–208.
- [5] Bräuner, T.: *An axiomatic approach to adequacy*, Ph.D. Thesis, BRICS, 1996.
- [6] Cartwright, R., Felleisen, M.: Observable sequentiality and full abstraction, *Proceedings of POPL '92*, 1992.
- [7] Cartwright, P.-L. Curien and M. Felleisen, R.: Fully abstract semantics for observably sequential languages, *Information and Computation*, 1994.
- [8] Felleisen, M., Friedman, D. P., Kohlbecker, E. E., Duba, B.: A syntactic theory of sequential control, *Theoretical Computer Science*, **52**, 1987, 205 – 207.
- [9] Ghica, D., Murawaki, A. S., Ong, C.-H. L.: Syntactic Control of Concurrency, *Proceedings of ICALP '04*, number 3142 in LNCS, Springer, 2004.
- [10] J.-Y. Girard, A. S., Scott, P.: Bounded Linear Logic: A Modular Approach to Polynomial-time computability, *Theoretical Computer Science*, **97**, 1992, 1–66.

- [11] Lafont, Y.: Soft Linear Logic and Polynomial time, *Theoretical Computer Science*, 2004, To appear.
- [12] Laird, J.: Bistability: an extensional characterization of sequentiality, *Proceedings of CSL '03*, number 2803 in LNCS, Springer, 2003.
- [13] Laird, J.: Bistability: A sequential domain theory, 2004, Available from <http://www.cogs.susx.ac.uk/users/jiml>.
- [14] Laird, J.: Decidability in Syntactic Control of Interference, *Proceedings of ICALP '05*, number 3580 in LNCS, Springer, 2005.
- [15] Laird, J.: The Elimination of Nesting in SPCF, *Proceedings of TLCA '05*, number 3461 in LNCS, Springer, 2005.
- [16] Laird, J.: Locally Boolean Domains, *Theoretical Computer Science*, **342**, 2005, 132–148.
- [17] Laird, J.: Bidomains and Full abstraction for countable non-determinism, *Proceedings of FoSSaCS'06*, number 3921 in LNCS, Springer, 2006.
- [18] Levy, P. B.: *Call-By-Push-Value*, Semantic Structures in Computation, Kluwer, 2004.
- [19] Longley, J.: The sequentially realizable functionals, *Annals of Pure and Applied Logic*, 1998.
- [20] Longley, J.: Universal Types and What They Are Good For, *Domain Theory, Logic and Computation: Proceedings of the 2nd International Symposium on Domain Theory*, Kluwer, 2004.
- [21] Murawski, A.: On Program Equivalence in Languages with Ground-type References, *Proceedings of LICS '03*, IEEE Press, 2003.
- [22] Reynolds, J.: Syntactic Control of Interference, *Conf. Record 5th ACM Symposium on Principles of Programming Languages*, 1978.