J. Functional Programming **7** (4): 357–394, July 1997. Printed in the United Kingdom

A syntactic method for finding least fixed points of higher-order functions over finite domains†

TYNG-RUEY CHUANG

Institute of Information Science, Academia Sinica, Nankang, Taipei 11529, Taiwan

BENJAMIN GOLDBERG

Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012, USA

Abstract

This paper describes a method for finding the least fixed points of higher-order functions over finite domains using symbolic manipulation. Fixed point finding is an essential component in the calculation of abstract semantics of functional programs, providing the foundation for program analyses based on abstract interpretation. Previous methods for fixed point finding have primarily used semantic approaches, which often must traverse large portions of the semantic domain even for simple programs. This paper provides the theoretical framework for a syntax-based analysis that is potentially very fast. The proposed syntactic method is based on an augmented simply typed lambda calculus where the symbolic representation of each function produced in the fixed point iteration is transformed to a syntactic normal form. Normal forms resulting from successive iterations are then compared syntactically to determine their ordering in the semantic domain, and to decide whether a fixed point has been reached. We show the method to be sound, complete and compositional. Examples are presented to show how this method can be used to perform strictness analysis for higher-order functions over non-flat domains. Our method is compositional in the sense that the strictness property of an expression can be easily calculated from those of its sub-expressions. This is contrary to most strictness analysers, where the strictness property of an expression has to be computed anew whenever one of its subexpressions changes. We also compare our approach with recent developments in strictness analysis.

Capsule Review

In recent years abstract interpretation has become a valuable tool for functional program analysis. For it to be practical, however, sound and efficient algorithms for computing

† A preliminary version of this paper was presented at the 1992 *ACM Conference on Lisp and Functional Programming*, June 1992, San Francisco, CA, USA, under the title 'A syntactic approach to fixed point computation on finite domains'. This research has been supported, in part, by the National Science Foundation (CCR–8909634) and DARPA (DARPA/ONR N00014–91–J1472). A part of this work was performed while Tyng–Ruey Chuang was at Department of Computer Science, New York University.

fixpoints are needed. This becomes especially difficult for higher-order abstractions, where the problem may be intractable. This paper outlines a method for computing fixpoints in such higher-order domains. The key contributions are a way to compute normal forms for the representation of functions, and a way to compare them in the information ordering. All results are shown to be sound, and there is some indication that the method will lead to a practical implementation.

1 Motivation and introduction

Finding the least fixed points of monotonic functions over finite domains is an important task in abstract interpretation. In abstract interpretation, a standard (or non-standard) semantics of a functional program is abstracted to a monotonic function over finite domains, and if the program contains recursive definitions, least fixed point finding is used to calculate the abstract semantics of the program. Much work had been performed to devise elegant and effective methods to calculate least fixed points on finite domains. Most notable is the frontier method developed by Clack and Peyton Jones (1985; 1987), Martin and Hankin (1987; 1989) and Hunt and Hankin (1989; 1991), with emphasis on applications to strictness analysis. Recent progresses in the development of abstract interpretation, not all of them based on least fixed point finding, abound as well in the literature. See, for example, Ferguson and Hughes (1993), Hankin and Hunt (1992), Hankin and Le Métayer (1994), Nocker (1993) and Seward (1993). We will briefly describe their work, and compare our approach to theirs, in section 6.

For now, let us briefly describe how the frontier method works. Take a function's strictness property as an example. For the moment, assume that we are only interested in whether a function application will terminate or not. We can describe the function's strictness property as a monotonic function f from a finite abstract domain D to the two element domain $\mathit{2}$. Domain $\mathit{2}$ contains only elements 0 and 1, with $0 \sqsubseteq_{\mathit{2}} 1$. Element 0 denotes that the function application will not terminate, while 1 denotes that it may or may not terminate. The maximal 0-frontier representation of function f is the smallest subset F_0 of D such that for any element d in D, if d is weaker than any element in F_0, then the result of applying f to d is 0. A similar minimal 1-frontier can also be defined which works equally well.

If the formulation of f is recursive in nature, then f is characterized as the least fixed point of a monotonic functional F from domain $D \to \mathit{2}$ to $D \to \mathit{2}$, where $D \to \mathit{2}$ is the monotonic function space from D to $\mathit{2}$. The least fixed point of F is approximated by using a successive sequence of maximal 0-frontiers. The approximation starts from $\bot_{D \to \mathit{2}}$, the least element in domain $D \to \mathit{2}$, which has the maximal 0-frontier representation $\{\top_D\}$. At each iteration of the approximation, the new frontier is found by moving down from the old one. This is accomplished first by using the old frontier as the function definition of f in the body of F. This yields a new definition of f. Then elements of the old frontier are evaluated according to the new definition of f to see if they are mapped to 0 or 1. If an element in the old frontier is mapped to 1, then the element is replaced by elements

not stronger than it in the new frontier. If an element in the old frontier is mapped to 0, then it is kept in the new frontier. The process then continues with the new frontier replacing the role of the old one. When the frontier does not change between successive iterations, then the fixed point has been found.

Often a function's strictness involves more than just termination status. For example, if the function yields a list, we may want to know if the result is spine-strict or not. In these cases, f is described by a monotonic function from a finite abstract domain D to a finite abstract domain R. Domain R may be more complicated than $\mathit{2}$. A typical case is where R is an abstract domain representing yet another monotonic function space. Again, if the formulation of f is recursive in nature, then f is characterized as the least fixed point of a functional from $D \to R$ to $D \to R$. For the frontier method, multiple frontiers will be needed for each step during the approximation sequence for finding F's least fixed point.

The frontier method is attractive in several ways. The representation is economical in space. It also allows fast function application. We simply check whether the argument is weaker than any of the elements in the maximal 0-frontier. If it is, then the result is 0; otherwise the result is 1.

Though elegant, there are several drawbacks in the frontier method. First, the frontier representations do not compose easily. Suppose that we have the frontier representations of functions f and g, what is the frontier representation of the functional composition $f \circ g$? It seems that we do not have much choice but to calculate it from scratch. Normally this will require the program texts of f and g. A functional program is very likely to be built up from smaller functional components by using the mechanism of abstraction, application and composition. But the frontier method does not provide such a building mechanism, unless, of course, when functions are fully applied to their arguments. We may say that the frontier method is not compositional, and does not fit well in a modular program development environment where program texts for functions may not be exported to one another.

Secondly, the frontier method is carried out mainly on the semantic domains of a program; the method pays little attention to the program text itself. This may cause great inefficiency. Consider a function $f \in \mathit{2}^{11} \to \mathit{2}$, which is defined as the least fixed point of the following functional F,

$$F \equiv \lambda f \,.\, \lambda x_0 \,.\, \lambda x_1 \,.\, \ldots \,.\, \lambda x_{10} \,.\, x_0 \sqcup (f\ x_0\ x_1\ \ldots\ x_{10}), \tag{1}$$

where $x_0, x_1, \ldots, x_{10} \in \mathit{2}$, and $\sqcup$ is the (infix) least upper bound function (i.e. the boolean OR operator) over domain $\mathit{2}$.

By symbolic evaluation, we determine

$$\lambda x_0 \,.\, \lambda x_1 \,.\, \ldots \,.\, \lambda x_{10} \,.\, x_0$$

to be the least fixed point of F. The process begins with the weakest approximation

$$\lambda x_0 \,.\, \lambda x_1 \,.\, \ldots \,.\, \lambda x_{10} \,.\, 0,$$

and takes only two iterations. By the above result, we also know that (the uncurried version of) f has maximal 0-frontier $\{\langle 0,1,1,1,1,1,1,1,1,1,1\rangle\}$.

But how would the frontier method reach this result? The frontier method will approximate the maximal 0-frontier of the least fixed point from the least element in domain $\mathit{2}^{11} \to \mathit{2}$. The frontier method will have to make, step by step, 2^{10} approximations to reach the above maximal 0-frontier, which is right in the middle of the ascending chain from the least element $\lambda x_0 \,.\, \lambda x_1 \,.\, \ldots \,.\, \lambda x_{10} \,.\, 0$ (whose maximal 0-frontier is $\{\langle 1,1,1,1,1,1,1,1,1,1,1\rangle\}$) to the greatest element $\lambda x_0 \,.\, \lambda x_1 \,.\, \ldots \,.\, \lambda x_{10} \,.\, 1$ (whose maximal 0-frontier is $\emptyset$) in the domain $\mathit{2}^{11} \to \mathit{2}$. One can show that, by induction, there exists an ascending chain of length $2^n + 1$ from the least element to the greatest element in domain $\mathit{2}^n \to \mathit{2}$, and the element $\lambda x_0 \,.\, \lambda x_1 \,.\, \ldots \,.\, \lambda x_{n-1} \,.\, x_0$ is right in the middle of this chain. Therefore, there are 2^{n-1} elements below the element $\lambda x_0 \,.\, \lambda x_1 \,.\, \ldots \,.\, \lambda x_{n-1} \,.\, x_0$, and 2^{n-1} elements above it. The frontier method will need 2^{n-1} approximations to reach the least fixed point. See Chuang (1993).

In such 'badly behaved' cases, the frontier method is very inefficient compared to a symbolic evaluation method. This phenomenon has been observed by Clack and Peyton Jones (1985; 1987). Hunt and Hankin (1989; 1991) further suggest that higher-order functional programs are often badly behaved in this way.

A symbolic evaluation method for computing least fixed points is proposed by Martin (1989). However, the method is limited to first-order functions over (the Cartesian products of) domain $\mathit{2}$. In this paper, we will develop a syntactic method suitable for symbolic calculation of least fixed points over the monotonic function spaces generated by domain $\mathit{2}$. Because higher-order functions over domain $\mathit{2}$ are themselves elements in the monotonic function spaces generated by domain $\mathit{2}$, our syntactic approach can be used to computing least fixed points of higher-order functions.

Our method uses a simply typed λ-calculus augmented with four predefined constants – $0, 1, \sqcap$, and $\sqcup$ – and their associated reduction rules. A language Λ is defined to describe elements of semantic domains, and is used to perform calculations upon them. A relation $\preceq$ (pronounced 'syntactically weaker') is defined among Λ terms, aiming to capture the relation $\sqsubseteq$ ('semantically weaker') among the denotation of Λ terms. Several reduction rules in Λ are based on the $\preceq$ relation.

We will show in section 3 that, in a sufficiently expressive sub-language of Λ, the proposed calculus is sound and complete with respect to the semantics. Because functions for computing least fixed points are themselves elements in some monotonic function spaces, we can directly perform the least fixed point computation on Λ by means of symbolic calculation. That is, for a given type σ denoting a finite domain, we will show in section 3 (Theorem 3.9) that there is a fixed point term $Y \in \Lambda_{(\sigma\to\sigma)\to\sigma}$ such that for any term F in the sub-language of $\Lambda_{\sigma\to\sigma}$, not only do YF and $F(YF)$ express the same element in domain D_σ, they also reduce to the same normal form in Λ_σ. The iterative approach, used above to compute the least fixed point of F in (1) by successively calculating more accurate estimates, is also shown to be sound and complete.

What are the advantages of using a syntactic method over a semantic method for computing least fixed points on finite domains? One advantage is that the syntactic method may make fewer iterations than the semantic one, as illustrated by the above example. Another advantage is that we have a more uniform way to cal-

culate the semantics of a functional program, whether it is a standard semantics or an abstract semantics, since these calculations differ only in how the reductions are performed. This is contrast to the frontier method of computing a program's abstract semantics, in which the method used is outside of the syntactic calculus (such as the λ-calculus) used to compute a program's standard semantics. Our syntactic method is compositional as well. The syntactic term for the strictness property of a function can be composed with other terms to derive strictness properties of new functions.

We will mainly address least fixed point finding in the monotonic function spaces generated by the basic domain *2*. We will later show how to relax this restriction and make the method applicable to basic domains other than *2*. Several implementation issues are discussed, and a substantial example is provided to demonstrate how the proposed syntactic method is used to compute the strictness properties of some higher-order functions over non-flat domains. We will also compare our work to other developments in strictness analyses.

2 A language Λ for finite domains

In this section we define a language Λ to describe the semantic elements in a class of finite domains. In addition to the syntax and semantics of Λ, a binary relation $\preceq$ (pronounced *syntactically weaker*) is defined on Λ terms, aiming to capture the relationship $\sqsubseteq$ (*semantically weaker*) among the elements in the domains. Reduction rules for Λ are introduced, several of them based on the relation $\preceq$.

Language Λ can be viewed as a variation of the simply typed λ-calculus. They differ in that Λ is augmented with four predefined constants – $0, 1, \sqcup$, and $\sqcap$ – and their associated reduction rules. They also differ in their interpretations of type expressions. Most of the definitions in this section are standard or intuitive, many of them borrowed from Barendregt (1984).

Definition 2.1
The set Γ of type expressions is inductively defined as follows:

- $2 \in \Gamma$, and
- $(\sigma \to \tau) \in \Gamma$ if $\sigma, \tau \in \Gamma$.

Definition 2.2
The language Λ, along with the sub-language Λ_σ for each type $\sigma \in \Gamma$, is inductively defined as follows.

- $x_\sigma \in \Lambda_\sigma$, where x_σ is a variable of type σ,
- $(MN) \in \Lambda_\tau$ if $M \in \Lambda_{\sigma\to\tau}$ and $N \in \Lambda_\sigma$,
- $(\lambda\, x_\sigma \,.\, M) \in \Lambda_{\sigma\to\tau}$ if $x_\sigma \in \Lambda_\sigma$ and $M \in \Lambda_\tau$,
- $0, 1 \in \Lambda_2$, and
- $(M \sqcup N), (M \sqcap N) \in \Lambda_2$ if $M, N \in \Lambda_2$.

Type 2 is the (only) *ground* type of language Λ. Language Λ can be viewed as the set of the simply typed λ-terms constructed from the ground type 2 and the four predefined constants: $0, 1 \in \Lambda_2$, and $\sqcup, \sqcap \in \Lambda_{2\to2\to2}$. The type constructor $\to$

is right associative. That is, $\sigma_1 \to \sigma_2 \to \ldots \to \sigma_n$ is a shorthand for $(\sigma_1 \to (\sigma_2 \to (\ldots \to \sigma_n)))$. The set Γ can also be defined inductively by the following: $2 \in \Gamma$, and $\sigma_1 \to \sigma_2 \to \ldots \to \sigma_n \to 2 \in \Gamma$ if $\sigma_1, \sigma_2, \ldots \sigma_n \in \Gamma$. We take the liberty to omit some parentheses and type subscripts in a Λ term if it is clear to do so. All type expressions in Γ and all terms in Λ are understood to be of finite length. Note that language Λ does not include a constant Y to express recursion. We will show in section 3 that, for any given type $\sigma \in \Gamma$, there is a term $Y_{(\sigma\to\sigma)\to\sigma}$ definable in language $\Lambda_{(\sigma\to\sigma)\to\sigma}$ and denoting the least fixed point function.

The following two definitions give the interpretation of types in Γ and terms in Λ.

Definition 2.3

- Type *2* denotes the domain $D_{\mathit{2}} = \{0, 1\}$, with the ordering $0 \sqsubseteq_{\mathit{2}} 1$, and
- Type $\sigma \to \tau$ denotes the domain $D_{\sigma\to\tau} = \{f \mid f$ is a total, monotonic function from domain D_σ to domain $D_\tau\}$, with the ordering $f \sqsubseteq_{\sigma\to\tau} g$ iff $f\,x \sqsubseteq_\tau g\,x$ for all $x \in D_\sigma$.

The least element in domain D_2, written as $\bot_2$, is 0. The least element in domain $D_{\sigma\to\tau}$, written as $\bot_{\sigma\to\tau}$, is the function that maps all elements in domain D_σ to $\bot_\tau$, the least element in domain D_τ. When the context is clear, we often drop the subscript σ in $\sqsubseteq_\sigma$ and often use a type expression σ to denote its semantic domain D_σ. For example, we often write *2* for D_2, and write $\mathit{2} \to \mathit{2}$ for $D_{2\to 2}$. It can be shown that for each type $\sigma \in \Gamma$, domain D_σ is a finite and distributive lattice.

Definition 2.4
Let environment ρ be a total function from typed variables to the union of all finite domains, $\bigcup_{\sigma\in\Gamma} D_\sigma$. Let $[\![M]\!]\rho$ be the interpretation of a term $M \in \Lambda$ under the environment ρ, and be defined as follows.

- $[\![x_\sigma]\!]\rho = \rho\, x_\sigma$,
- $[\![MN]\!]\rho = [\![M]\!]\rho\ [\![N]\!]\rho$,
- $[\![\lambda\, x\,.\, M]\!]\rho = \lambda\, y\,.\,([\![M]\!](\rho[x \mapsto y]))$,
- $[\![0]\!]\rho = 0$,
  $[\![1]\!]\rho = 1$, and
- $[\![M \sqcup N]\!]\rho = [\![M]\!]\rho \sqcup [\![N]\!]\rho$,
  $[\![M \sqcap N]\!]\rho = [\![M]\!]\rho \sqcap [\![N]\!]\rho$.

Note that we use the same symbol to denote both a syntactic phrase and its semantic meaning (for example, the symbol 0 in $[\![0]\!]\rho = 0$). We assume that this will not cause confusion. If a term $M \in \Lambda$ is closed, then its interpretation is simply written as $[\![M]\!]$, without referring to any environment, since environments do not affect the interpretation of M.

We now describe how to perform syntactic calculations in Λ. First we define the binary relation $\preceq$ between Λ terms. It is intended that, for $M, N \in \Lambda$, if $M \preceq N$ then $[\![M]\!]\rho \sqsubseteq [\![N]\!]\rho$ for all environments ρ.

Definition 2.5
A binary relation $\mathcal{R}$ on language Λ is *compatible* if the following inference rules apply for all $F, G, H \in \Lambda$ and all $L, M, N \in \Lambda_{\mathit{2}}$.

- (application)

$$\frac{F\ \mathcal{R}\ G}{(FH)\ \mathcal{R}\ (GH)} \qquad \frac{F\ \mathcal{R}\ G}{(HF)\ \mathcal{R}\ (HG)}$$

- (abstraction)

$$\frac{F\ \mathcal{R}\ G}{(\lambda x_\sigma . F)\ \mathcal{R}\ (\lambda x_\sigma . G)}$$

- ($\sqcup$)

$$\frac{M\ \mathcal{R}\ N}{(L \sqcup M)\ \mathcal{R}\ (L \sqcup N)} \qquad \frac{M\ \mathcal{R}\ N}{(M \sqcup L)\ \mathcal{R}\ (N \sqcup L)}$$

- ($\sqcap$)

$$\frac{M\ \mathcal{R}\ N}{(L \sqcap M)\ \mathcal{R}\ (L \sqcap N)} \qquad \frac{M\ \mathcal{R}\ N}{(M \sqcap L)\ \mathcal{R}\ (N \sqcap L)}$$

Definition 2.6
The relation $\preceq$ on language Λ is the compatible, reflexive, and transitive relation induced by the following axioms for all terms $L, M, N \in \Lambda_2$.

- (0 and 1)

$$0 \preceq M, \quad M \preceq 1$$

- ($\sqcup$ and $\sqcap$)

$$M \preceq (M \sqcup N), \quad (M \sqcap N) \preceq M$$

- (idempotence)

$$(M \sqcup M) \preceq M, \quad M \preceq (M \sqcap M)$$

- (commutativity)

$$(M \sqcup N) \preceq (N \sqcup M), \quad (M \sqcap N) \preceq (N \sqcap M)$$

- (associativity)

$$\begin{aligned}
(L \sqcup (M \sqcup N)) &\preceq ((L \sqcup M) \sqcup N) \\
((L \sqcup M) \sqcup N) &\preceq (L \sqcup (M \sqcup N)) \\
(L \sqcap (M \sqcap N)) &\preceq ((L \sqcap M) \sqcap N) \\
((L \sqcap M) \sqcap N) &\preceq (L \sqcap (M \sqcap N))
\end{aligned}$$

Definition 2.7
Let $M, N \in \Lambda$. Then $M \simeq N$ if $M \preceq N$ and $N \preceq M$.

By using compatibility in defining the relation $\preceq$, we have $\preceq$ well-defined for all Λ terms, not just for Λ_2 terms. Note that the definition of $\preceq$ contains some redundancy. For example, not all of the four associativity axioms are needed once we have the commutativity axioms. We include them for clarity, however. Also, by convention, Λ terms that are α-congruent are identified as the same term. As a consequence, we can prove $\lambda x_2 . 0 \preceq \lambda y_2 . y$ (which would be impossible if α-congruent terms were not identified). It is proved by first showing $\lambda x_2 . 0 \preceq \lambda x_2 . x$, then by α-congruence.

Relation $\preceq$ imposes a rather weak theory on Λ. For example, we cannot even

prove that $1 \preceq (\lambda\, x_2\,.\,1)\,1$, although it is certainly true by semantic reasoning. It is clear that some reduction rules, $\boldsymbol{\beta}$-reduction in the above case, will be needed if a more complete theory is desirable. On the other hand, the simplicity of $\preceq$ has its virtue. For example, it is easy to check whether two Λ terms satisfy the $\preceq$ relationship or not, as shown in the following by Lemma 2.8.

We will freely use notation like $\bigsqcup_{i\in I} M_i$ to describe the disjunction of a finite number of terms M_i, where $i \in I$, I a finite index set, without explicitly stating their permutations within the disjunction. Because associativity and commutativity are incorporated in the definition of $\preceq$, we know that all those various disjunctions are equivalent under the $\simeq$ relationship. By convention, we define $\bigsqcup_{i\in\emptyset} M_i \equiv 0$ and $\bigsqcap_{i\in\emptyset} M_i \equiv 1$. It follows that $\bigsqcup\{M\} \equiv M$ and $\bigsqcap\{M\} \equiv M$.

Lemma 2.8
Let $E, F, G, H \in \Lambda$, and let $M_i, N_j \in \Lambda_2$ for $i \in I$ and $j \in J$, where I and J are some finite index sets. Then,

1. If $F \preceq G$, then F and G have the same type.
2. $EF \preceq GH$ iff $E \preceq G$ and $F \preceq H$.
3. $\lambda\, x\,.\,F \preceq \lambda\, x\,.\,G$ iff $F \preceq G$.
4. $\bigsqcup_{i\in I} M_i \preceq \bigsqcup_{j\in J} N_j$ iff for all $i \in I$ there is a $j \in J$ such that $M_i \preceq N_j$.
5. $\bigsqcap_{i\in I} M_i \preceq \bigsqcap_{j\in J} N_j$ iff for all $j \in J$ there is a $i \in I$ such that $M_i \preceq N_j$.

Proof outline
For the first assertion, we notice that all the base cases for $F \preceq G$, as described in Definition 2.6, require F and G both to be of the same type. Furthermore, the compability rules described in Definition 2.5 only introduce identical terms or binding variables of the same type. This concludes that F and G have the same type.

We then prove the other four assertions all by induction. The induction is based on the structure of the proof which leads to the $\preceq$ relationship between Λ terms. A relationship like $F \preceq G$ can only be proved if

- $F \preceq G$ is an axiom described in Definition 2.6, or it is the reflexivity axiom (i.e. $F \equiv G$); or
- some intermediate $\preceq$ relationships between Λ terms are proved first, and then the proof for $F \preceq G$ are obtained from those intermediate results either by using the compatibility inference rules (as described in Definition 2.5) or by using the transitivity inference rule (i.e. $F \preceq G$ because $F \preceq H$ and $H \preceq G$ for some H).

Therefore, the base cases for an induction are those proofs which only consist the use of a single axiom. The induction hypothesis is that the assertion is true for the proof leading to the intermediate results. What remains to be proved is that the assertion is also true for the proof which incorporates the intermediate results by using the applicable inference rules.

For example, suppose we want to prove the $\Longrightarrow$ part of the second assertion in this lemma: $EF \preceq GH$ iff $E \preceq G$ and $F \preceq H$. It suffices to show that if there is a

proof for $EF \preceq GH$, then there are also proofs for both $E \preceq G$ and $F \preceq H$. What would a proof for $EF \preceq GH$ look like? There are two cases:

- $EF \preceq GH$ is simply proved by an axiom. The only applicable axiom in this setting is the reflexivity axiom. That is, $EF \equiv GH$. It follows that $E \equiv G$ and $F \equiv H$. Then, by reflexivity, we have proofs for both $E \preceq G$ and $F \preceq H$.
- $EF \preceq GH$ is the result obtained by using one of those inference rules on some intermediate results. The only applicable rules in this setting is the application rule or the transitivity rule. Suppose that $EF \preceq GH$ follows from the application rule. Then either $E \preceq G$ and $F \equiv H$, or $E \equiv G$ and $F \preceq H$. In either case, by reflexivity, we know that there are proofs for both $E \preceq G$ and $F \preceq H$. On the other hand, suppose that $EF \preceq GH$ follows from the transitivity rule. Then there is a term $L \equiv MN$ such that both $EF \preceq L$ and $L \preceq GH$ can be proved. Since we can prove $EF \preceq MN$, then, by induction hypothesis, there must be proofs for both $E \preceq M$ and $F \preceq N$. Similarly, there must be proofs for both $M \preceq G$ and $N \preceq H$. By transitivity, it follows that we can prove both $E \preceq G$ and $F \preceq H$.

This completes the $\Longrightarrow$ part of the second assertion in this lemma. The proofs for the $\Longrightarrow$ parts of the other assertions in this lemma are similar to the above. The proofs for the $\Longleftarrow$ parts are straightforward. □

A naive algorithm based on the above lemma can determine the $\preceq$ relationship between two terms M and N in $O(|M|\,|N|)$ time. We define in the following four reduction relations for Λ terms, two of them are based on the relation $\preceq$. In the definition below, we use $\prod_{i\in I} J_i$ to denote the Cartesian product of finite sets J_i, and use $p|i$ to denote the ith component of a tuple $p \in \prod_{i\in I} J_i$.

Definition 2.9
The reduction relations $\boldsymbol{\beta}$, $\sqcup$, $\sqcap$ and $\boldsymbol{d}$ on language Λ are defined as follows.

$$\begin{aligned}
\boldsymbol{\beta} &= \{((\lambda x\,.\,M)N,\ M[x := N]) \mid M, N \in \Lambda\}.\\
\sqcup &= \{(\bigsqcup_{i\in I} M_i,\ \bigsqcup_{i\in I-J} M_i) \mid \emptyset \neq J \subset I,\ (\forall j \in J)(\exists i \in I - J)(M_j \preceq M_i)\}.\\
\sqcap &= \{(\bigsqcap_{i\in I} M_i,\ \bigsqcap_{i\in I-J} M_i) \mid \emptyset \neq J \subset I,\ (\forall j \in J)(\exists i \in I - J)(M_i \preceq M_j)\}.\\
\boldsymbol{d} &= \{(\bigsqcap_{i\in I} \bigsqcup_{j\in J_i} M_{i,j},\ \bigsqcup_{p\in\prod_{i\in I} J_i} \bigsqcap_{i\in I} M_{i,p|i}) \mid |I| > 1,\ (\exists i \in I)(|J_i| > 1)\}.
\end{aligned}$$

A reduction relation $\boldsymbol{r}$ is a set consisting of some pairs (M, N), where M, N are Λ term. A $\boldsymbol{r}$ reduction can be used to transforms a term $C[M]$ (read 'M in context C') to $C[N]$ if $(M, N) \in \boldsymbol{r}$. Although we have defined a language Λ to represent elements in higher-order domains, until the definition of the above four reduction relations, we had not described how to perform calculation based on Λ.

Recall that we write $\bigsqcup_{i\in I} M_i$ to describe the disjunction of a finite number of terms M_i, where $i \in I$, I a finite index set. Reduction relation $\sqcup$ is used to eliminate redundant terms in a disjunction, using the syntactically weaker relation $\preceq$. That is, we eliminate term M_j if there is a term $i \in I - \{j\}$ such that $M_j \preceq M_i$.

Likewise, reduction relation $\sqcap$ eliminates redundant terms in a conjunction. Reduction relation $\boldsymbol{d}$ distributes conjunction over disjunction. We aim to keep disjunctive normal forms. There is no *a priori* reason why conjunctive normal forms are not used. An alternative development can certainly prefer conjunctive normal forms over disjunctive normal forms.

Note that reduction relations $\sqcup$, $\sqcap$, and $\boldsymbol{d}$ are modulo associativity and commutativity of $\sqcup$ and $\sqcap$. Permutation of sub–terms in a disjunctive, or conjunctive, term is considered insignificant. For example, we will view both $(L \sqcap M) \sqcup (L \sqcap N)$ and $(N \sqcap L) \sqcup (L \sqcap M)$ denoting the same resulting term from a one-step $\boldsymbol{d}$-reduction on term $L \sqcap (M \sqcup N)$, for $L, M, N \in \Lambda_2$.

Let $\boldsymbol{r}$ be a reduction relation on Λ. We use $\rightarrow_r$ to denote the compatible closure of $\boldsymbol{r}$, and use $\rightarrow_r^*$ to denote the reflexive and transitive closure of $\rightarrow_r$. We also use $\boldsymbol{rs}$ to denote the reduction relation $\boldsymbol{r} \cup \boldsymbol{s}$. The notations $\rightarrow$ and $\rightarrow^*$ are, respectively, shorthands for $\rightarrow_{\beta \sqcup \sqcap d}$ and $\rightarrow^*_{\beta \sqcup \sqcap d}$. The standard definitions of $\boldsymbol{r}$-*redex* and $\boldsymbol{r}$-*normal form* are used. We will use *normal form* as an abbreviation for $\boldsymbol{\beta} \sqcup \sqcap \boldsymbol{d}$-normal form. A term $M \in \Lambda$ is called *strongly normalizable* iff M is $\boldsymbol{\beta} \sqcup \sqcap \boldsymbol{d}$-strongly normalizable; that is, there is no infinite $\boldsymbol{\beta} \sqcup \sqcap \boldsymbol{d}$-reduction sequence starting with M.

Proposition 2.10
Every term $M \in \Lambda$ is strongly normalizable.

Proof
See Appendix A $\square$

Proposition 2.11
$\boldsymbol{\beta} \sqcup \sqcap \boldsymbol{d}$ is Church–Rosser.

Proof
See Appendix B. $\square$

The above two propositions imply that, efficiency considerations aside, we need not worry about reduction strategy when computing the normal form of a Λ term.

Proposition 2.12 (Soundness)
Let $L, M, N \in \Lambda$. Then,

1. $M \preceq N$ implies $[\![M]\!]\rho \sqsubseteq [\![N]\!]\rho$, and
2. $L \rightarrow^* M$ implies $[\![L]\!]\rho = [\![M]\!]\rho$

for all environments ρ.

Proof
By the definitions of $\preceq$ and $\rightarrow^*$ and by a straightforward structural induction on how $M \preceq N$ and $M \rightarrow^* N$ are derived. $\square$

The above proposition means that the syntactically weaker relation $\preceq$ and the reduction relation $\boldsymbol{\beta} \sqcup \sqcap \boldsymbol{d}$ are faithful to the semantics of the language Λ. But is the semantically weaker relationship between two Λ terms completely captured by the syntactic notions of $\preceq$ and $\boldsymbol{\beta} \sqcup \sqcap \boldsymbol{d}$? If it is, then in order to determine the

Table 1. *The valuation table of applying* $M \equiv \lambda f \,.\, \lambda x \,.\, (f\,0) \sqcup ((f\,1) \sqcap x)$ *and* $N \equiv \lambda f \,.\, \lambda x \,.\, f\,x$ *to all elements* $d \in D_{2\to 2}$

$d \in D_{2\to 2}$	$[\![M]\!]\ d$	$[\![N]\!]\ d$
$\lambda x_2 \,.\, 0$	$\lambda x_2 \,.\, 0$	$\lambda x_2 \,.\, 0$
$\lambda x_2 \,.\, x$	$\lambda x_2 \,.\, x$	$\lambda x_2 \,.\, x$
$\lambda x_2 \,.\, 1$	$\lambda x_2 \,.\, 1$	$\lambda x_2 \,.\, 1$

Note. The two sets of results agree on all three elements in domain $D_{2\to 2}$, showing $[\![\lambda f \,.\, \lambda x \,.\, (f\,0) \sqcup ((f\,1) \sqcap x)]\!] = [\![\lambda f \,.\, \lambda x \,.\, f\,x]\!]$.

semantic relationship for any two terms $M, N \in \Lambda$, we can simply calculate the normal forms of M and N and then compare the two normal forms using the $\preceq$ rules. Unfortunately, the answer is no.

Proposition 2.13 (Incompleteness)
There exist normal forms $M, N \in \Lambda$ such that $[\![M]\!]\rho \sqsubseteq [\![N]\!]\rho$ for all environments ρ, but $M \preceq N$ is not provable.

As a proof, let us consider the following two terms in language $\Lambda_{(2\to 2)\to 2\to 2}$,

$$\begin{aligned} M &\equiv \lambda f \,.\, \lambda x \,.\, (f\,0) \sqcup ((f\,1) \sqcap x), \\ N &\equiv \lambda f \,.\, \lambda x \,.\, f\,x. \end{aligned}$$

By Table 1, it is clear that $[\![M]\!] = [\![N]\!]$. However, neither $M \preceq N$ nor $N \preceq M$ is provable according to rules in Definition 2.5 and 2.6.

Since the reduction relation $\boldsymbol{\beta} \sqcup \sqcap\boldsymbol{d}$, along with relation $\preceq$, is incomplete with respect to the semantics of the language Λ, we can either introduce new reduction relations and new inference rules (for syntactic weakness) to achieve completeness in language Λ, or we can define a sub-language of Λ such that $\boldsymbol{\beta} \sqcup \sqcap\boldsymbol{d}$ and $\preceq$ is complete in the sub-language. We define in the next section a sub-language of Λ, called Λ^0, such that $\boldsymbol{\beta} \sqcup \sqcap\boldsymbol{d}$ and $\preceq$ is complete in Λ^0. Furthermore, we will show that language Λ^0 is expressive enough to describe all the semantic elements in finite domains D_σ, where $\sigma \in \Gamma$.

3 A complete sub-language Λ^0 of Λ

We define a restricted language Λ^0 of Λ, with the intention of making the syntactically weaker relationship $\preceq$ among Λ^0 terms complete, with respect to the semantically weaker relationship $\sqsubseteq$ among their denotations. We often write a term T in Λ as $\lambda \vec{x} \,.\, M$ (that is, $\lambda x_1 \,.\, \ldots \,.\, \lambda x_n \,.\, M$), where M is not of the form $\lambda y \,.\, N$. We call $\vec{x}$ the *vector* of T and M the *body* of T.

Definition 3.1
The sub-language Λ^0_σ of Λ_σ is inductively defined for each type $\sigma \in \Gamma$ as follows.

- $0, 1 \in \Lambda^0_2$; and

- $\lambda\,\vec{x}\,.\,M \in \Lambda^0_{\sigma_1\to\ldots\to\sigma_n\to 2}$ if
  - $\vec{x}$ consists of variables of types $\sigma_1,\ldots,\sigma_n$, and M contains no free variable other than those from $\vec{x}$,
  - $M \in \Lambda_2$ and it is in $\boldsymbol{\beta}\sqcup\sqcap\boldsymbol{d}$ normal form $\bigsqcup_{i\in I}\bigsqcap_{j\in J_i} M_{i,j}$, and
  - each term $M_{i,j}$ is either a bound variable of type 2, or an application of the form $(x_i\ e_1\ \ldots\ e_m)$, where variable x_i is in $\vec{x}$ and is of type $\sigma_i = \tau_{i_1}\to\ldots\to\tau_{i_m}\to 2$, and term $e_k \in \Lambda^0_{\tau_{i_k}}$ for each $1\le k\le m$.

The motivation behind the above definition is to make language Λ^0 as simple as possible. Therefore, we let Λ^0 include only those closed terms which are in normal form and whose bodies are the disjunction of conjunctions of function applications, with the function arguments being in Λ^0 again. The bound variables of a Λ^0 term are never used as function arguments in the body; they are only used as function names, applied to other Λ^0 terms. We will later show that Λ^0 is actually not a very restricted language. In fact, it is expressive enough to describe all the elements in the semantic domains $D_\sigma, \sigma\in\Gamma$. On the other hand, the simplicity of Λ^0 helps in showing that semantical weakness implies syntactical weakness in Λ^0. Note that, by induction, it can be shown that for any given type $\sigma\in\Gamma$, there are only finite number of Λ^0_σ terms. We will later use this property to show that there exists no infinite approximation sequence during the least fixed point iteration.

Example 3.2
Assume that variable f has type $2\to 2$ and variable x has type 2. Then the following are the only 10 terms in $\Lambda^0_{(2\to 2)\to(2\to 2)}$ (ignoring the variations introduced by α-congruence and by associativity and commutativity of $\sqcup$ and $\sqcap$):

$$\begin{array}{ll}
\lambda f\,.\,\lambda x\,.\,0, & \lambda f\,.\,\lambda x\,.\,(f\ 0)\sqcap x,\\
\lambda f\,.\,\lambda x\,.\,(f\ 1)\sqcap x, & \lambda f\,.\,\lambda x\,.\,f\ 0,\\
\lambda f\,.\,\lambda x\,.\,x, & \lambda f\,.\,\lambda x\,.\,(f\ 0)\sqcup((f\ 1)\sqcap x),\\
\lambda f\,.\,\lambda x\,.\,(f\ 0)\sqcup x, & \lambda f\,.\,\lambda x\,.\,f\ 1,\\
\lambda f\,.\,\lambda x\,.\,(f\ 1)\sqcup x, & \lambda f\,.\,\lambda x\,.\,1.
\end{array}$$

The following five terms are not in $\Lambda^0_{(2\to 2)\to(2\to 2)}$:

$$\begin{array}{l}
\lambda f\,.\,f,\\
\lambda f\,.\,\lambda x\,.\,f\ x,\\
\lambda f\,.\,\lambda x\,.\,f\ (f\ 0),\\
\lambda f\,.\,\lambda x\,.\,(f\ 1)\sqcap((f\ 0)\sqcup x),\\
\lambda f\,.\,\lambda x\,.\,x\sqcup x.
\end{array}$$

That is because the first term's body is not of type 2, the arguments in the second term's and the third term's function applications are not in language Λ^0_2, and the bodies of last two are not in $\boldsymbol{\beta}\sqcup\sqcap\boldsymbol{d}$ normal form.

Since Λ^0 is a subset of Λ, it inherits the soundness property from Λ. Λ^0 also has the following nice properties.

Proposition 3.3

Let $M, N \in \Lambda^0$. The normal forms for MN, $\lambda x\,.\,M$, $M \sqcup N$, and $M \sqcap N$ are all in Λ^0.

Proof
By Definition 3.1, term $\lambda x\,.\,M$ is in normal form if M is in normal form.

For terms $M \sqcup N$ and $M \sqcap N$, $\sqcup\, \sqcap\, \boldsymbol{d}$-reductions suffice to reduce them to normal forms. For term $M_{\tau\to\gamma}N_\tau$, use an induction on type expressions. □

Proposition 3.4 (Definability)
For every element $f \in D_\sigma, \sigma \in \Gamma$, there is a term $F \in \Lambda^0_\sigma$ such that $[\![F]\!] = f$.

Proof
We perform an induction based on the structure of the type expression σ. The proposition is true for the base case $\sigma = 2$. If $\sigma = \tau \to \gamma$, then, by induction hypotheses, all elements in domains D_τ and D_γ are definable in languages Λ^0_τ and Λ^0_γ respectively. It remains to be proved that all elements in D_σ can be defined in language Λ^0_σ.

The step function $step_{a,b}$ in domain $D_{\tau\to\gamma}$, where $a \in D_\tau$ and $b \in D_\gamma$, is defined by

$$step_{a,b}\ \ x \ \ = \ \ \mathit{if}\ a \sqsubseteq_\tau x\ \mathit{then}\ b\ \mathit{else}\ \bot_\gamma.$$

Furthermore, an element $f \in D_{\tau\to\gamma}$ can be expressed as the least upper bound of a set of step functions in $D_{\tau\to\gamma}$. That is, $f = \bigsqcup_{i\in I} step_{a_i,b_i}$ for some index set I. This standard construction can be found, for example, in Plotkin (1977). Since D_τ and D_γ are finite, the index set I is finite.

Let type $\tau = \tau_1 \to \ldots \tau_n \to 2$ and type $\gamma = \gamma_1 \to \ldots \gamma_m \to 2$. Then the step function can be defined equivalently as

$$\begin{array}{rcll} step_{a,b}\ \ x\ z_1\ \ldots\ z_m & = & \mathit{if} & ((a\ y_1\ \ldots\ y_n) \sqsubseteq_2 (x\ y_1\ \ldots\ y_n) \\ & & & \mathit{for\ all}\ y_1 \in D_{\tau_1}, \ldots, y_n \in D_{\tau_n}) \\ & & \mathit{then} & (b\ z_1\ \ldots\ z_m)\ \mathit{else}\ 0, \end{array}$$

where $z_1 \in D_{\gamma_1}, \ldots, z_m \in D_{\gamma_m}$. This is the same as

$$\begin{array}{l} step_{a,b}\ \ x\ z_1\ \ldots\ z_m \ \ = \\ \quad (\bigsqcap\{(x\,y_1\ \ldots\ y_n) \mid y_1 \in D_{\tau_1}, \ldots, y_n \in D_{\tau_n}, (a\,y_1\ \ldots\ y_n) = 1\}) \sqcap (b\,z_1\ \ldots\ z_m). \end{array}$$

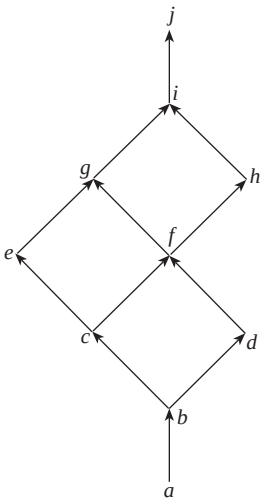
Motivation for the above two reformulations is to bring the *step* function in a form that is close to language Λ^0. By the induction hypotheses, all elements in domains $D_{\tau_1}, \ldots, D_{\tau_n}$, and D_γ, are definable. Hence, the above step function can be defined in language Λ^0_σ as the normal form of the following term

$$\lambda x\,.\,\lambda z_1 \ldots.\,\lambda z_m\,.\,(\bigsqcap\nolimits_{y_1\in D_{\tau_1},\ldots,y_n\in D_{\tau_n},(a\ y_1\ldots y_n)=1} (x\ Y_1 \ldots Y_n)) \sqcap (B\ z_1 \ldots z_m),$$

where $Y_i \in \Lambda^0_{\tau_i}$ with $[\![Y_i]\!] = y_i$, and $B \in \Lambda^0_\gamma$ with $[\![B]\!] = b$.

Because f can be expressed as $\bigsqcup_{i\in I} step_{a_i,b_i}$, and each of the function $step_{a_i,b_i}$ can be defined by a Λ^0_σ term $\lambda x\,.\,\lambda z_1 \ldots.\,\lambda z_m\,.\,M_i$, f can be defined by the following term F,

$$F \equiv \lambda x\,.\,\lambda z_1 \ldots.\,\lambda z_m\,.\,\bigsqcup\nolimits_{i\in I} M_i.$$



j	=	[v * w,	< * w,	w * w],
i	=	[v * <,	< * w,	w * w],
h	=	[v * v,	< * w,	w * w],
g	=	[v * <,	< * <,	w * w],
f	=	[v * v,	< * <,	w * w],
e	=	[v * <,	< * <,	w * <],
d	=	[v * v,	< * v,	w * w],
c	=	[v * v,	< * <,	w * <],
b	=	[v * v,	< * v,	w * <],
a	=	[v * v,	< * v,	w * v].

w	=	[0 * 1,	1 * 1],
<	=	[0 * 0,	1 * 1],
v	=	[0 * 0,	1 * 0].

Fig. 1. The 10 elements in domain $D_{(2\to2)\to(2\to2)}$.

Note. The three elements in domain $D_{2\to2}$ are denoted by $\bot, \|$, and $\top$, with $\bot \sqsubseteq \| \sqsubseteq \top$. The 10 elements in domain $D_{(2\to2)\to(2\to2)}$ are denoted by $a, b, c, d, e, f, g, h, i$, and j. The lower an element in the lattice, the less defined the element is.

The normal form of F is in language Λ^0_σ. □

Example 3.5

There is a function y in domain $D_{((2\to2)\to(2\to2))\to(2\to2)}$ such that for all elements z in domain $D_{(2\to2)\to(2\to2)}$, $(y\ z)$ is the least fixed point of z. Can we find a term Y in language $\Lambda^0_{((2\to2)\to(2\to2))\to(2\to2)}$ such that $[\![Y]\!] = y$?

Before calculating Y, let us first draw a diagram of domain $D_{(2\to2)\to(2\to2)}$. The diagram in Figure 1 illustrates the ordering of the 10 elements in domain $D_{(2\to2)\to(2\to2)}$. The functionalities of the 10 elements are also described as maps from domain $D_{2\to2}$ to $D_{2\to2}$ in the illustration. We write the 3 elements in $D_{2\to2}$ as $\bot, \|$, and $\top$, with the ordering $\bot \sqsubseteq_{2\to2} \| \sqsubseteq_{2\to2} \top$. Note that they are definable in $\Lambda^0_{2\to2}$ by $\bot = [\![\lambda x\,.\,0]\!]$, $\| = [\![\lambda x\,.\,x]\!]$, and $\top = [\![\lambda x\,.\,1]\!]$.

From Figure 1, we observe that the least fixed point function y can be described by

$$y = \bigsqcup\{step_{e,\|},\ step_{i,\top}\}.$$

This describes that, for element $z \in D_{(2\to2)\to(2\to2)}$, if $e \sqsubseteq z$ then the least fixed point of z is greater than $\|$, and if $i \sqsubseteq z$ then the least fixed point of z is greater than $\top$, otherwise the least fixed point of z is $\bot$. By using Proposition 3.4, $step_{e,\|}$ and $step_{i,\top}$ can be defined by

$$\begin{aligned} step_{e,\|} &= [\![\lambda z\,.\,\lambda x\,.\,(z\ (\lambda x\,.\,0)\ 1) \sqcap x]\!], \\ step_{i,\top} &= [\![\lambda z\,.\,\lambda x\,.\,(z\ (\lambda x\,.\,0)\ 1) \sqcap (z\ (\lambda x\,.\,x)\ 0)]\!], \end{aligned}$$

where variable z is of type $(2 \to 2) \to (2 \to 2)$ and x is of type 2.

The following is a step-by-step derivation for the above definition of $step_{e,\|}$. The case for $step_{i,\top}$ is similar.

$$
\begin{aligned}
& step_{e,\|} \\
= \;& \lambda\, z\,.\, \mathit{if}\ e \sqsubseteq_{(2\to 2)\to(2\to 2)} z\ \mathit{then}\ \| \ \mathit{else}\ \bot \\
= \;& \lambda\, z\,.\, \lambda\, x\,.\, \mathit{if}\ ((e\ f\ x) \sqsubseteq_2 (z\ f\ x)\ \ \mathit{for\ all}\ f \in D_{2\to 2}, x \in D_2)\ \mathit{then}\ \| \ x\ \mathit{else}\ \bot\ x \\
= \;& \lambda\, z\,.\, \lambda\, x\,.\, \mathit{if}\ (x \sqsubseteq_2 (z\ f\ x)\ \ \mathit{for\ all}\ f \in D_{2\to 2}, x \in D_2)\ \mathit{then}\ x\ \mathit{else}\ 0 \\
= \;& \lambda\, z\,.\, \lambda\, x\,.\, \mathit{if}\ (1 \sqsubseteq_2 (z\ \bot\ 1)\ \mathit{and}\ 1 \sqsubseteq_2 (z\ \|\ 1)\ \mathit{and}\ 1 \sqsubseteq_2 (z\ \top\ 1))\ \mathit{then}\ x\ \mathit{else}\ 0 \\
= \;& \lambda\, z\,.\, \lambda\, x\,.\, \mathit{if}\ (1 \sqsubseteq_2 z\ \bot\ 1)\ \mathit{then}\ x\ \mathit{else}\ 0 \\
= \;& \lambda\, z\,.\, \lambda\, x\,.\, (z\ \bot\ 1) \sqcap x \\
= \;& [\![\lambda\, z\,.\, \lambda\, x\,.\, (z\ (\lambda\, x\,.\, 0)\ 1) \sqcap x]\!]
\end{aligned}
$$

The least fixed point function y can therefore be represented by a $\Lambda^0_{((2\to 2)\to(2\to 2))\to(2\to 2)}$ term Y, where

$$Y \equiv \lambda\, z\,.\, \lambda\, x\,.\, ((z\ (\lambda\, x\,.\, 0)\ 1) \sqcap x) \sqcup ((z\ (\lambda\, x\,.\, 0)\ 1) \sqcap (z\ (\lambda\, x\,.\, x)\ 0)).$$

As a side note, the 10 elements in domain $D_{(2\to 2)\to(2\to 2)}$ happen to be defined in Example 3.2 by the 10 $\Lambda^0_{(2\to 2)\to(2\to 2)}$ terms.

Proposition 3.6 (Completeness)
Let $M, N \in \Lambda^0$. Then, $[\![M]\!] \sqsubseteq [\![N]\!]$ implies $M \preceq N$.

Proof outline
The idea is to show that if $M \preceq N$ is not provable, then $[\![M]\!] \not\sqsubseteq [\![N]\!]$. Suppose that terms M and N are of type $\sigma = \sigma_1 \to \ldots \to \sigma_n \to 2$. Write M as $\lambda\, \vec{x}\,.\, \bigsqcup_{i\in I} \bigsqcap_{j\in J_i} M_{i,j}$ and N as $\lambda\, \vec{x}\,.\, \bigsqcup_{k\in K} \bigsqcap_{l\in L_k} N_{k,l}$, where $\vec{x}$ is a vector of variables whose types are $\sigma_1, \ldots, \sigma_n$.

We prove the proposition by an induction on type expression σ. The base case is $\sigma = 2$, in which the only two Λ^0_2 term are 0 and 1, and the proposition is true. We want to show that the proposition is true for type $\sigma = \sigma_1 \to \ldots \to \sigma_n \to 2$, given it is true for types $\sigma_1, \ldots, \sigma_n$.

Since $M \preceq N$ is not provable, there exists an index $i \in I$ such that for all indices $k \in K$, $\bigsqcap_{j\in J_i} M_{i,j} \preceq \bigsqcap_{l\in L_k} N_{k,l}$ is not provable. Based on the conjunctive term $\bigsqcap_{j\in J_i} M_{i,j}$, we will construct an environment ρ such that

$$[\![\bigsqcup_{i\in I} \bigsqcap_{j\in J_i} M_{i,j}]\!]\rho = 1 \qquad \text{but} \qquad [\![\bigsqcup_{k\in K} \bigsqcap_{l\in L_k} N_{k,l}]\!]\rho = 0.$$

That is, $[\![M]\!] \not\sqsubseteq [\![N]\!]$. This will complete the proof.

Given such an index $i \in I$, we now describe how to construct the environment ρ. Let x_h be one of the bound variables in $\vec{x}$. Suppose that x_h is of type 2 and $M_{i,j} \equiv x_h$ for some $j \in J_i$, then we map x_h to 1 in environment ρ. If x_h never occurs in the conjunctive term $\bigsqcap_{j\in J_i} M_{i,j}$, then x_h is mapped to 0. If variable x_h is of type $\sigma_h = \tau_{h_1} \to \ldots \tau_{h_m} \to 2$, then we define a set $P \subseteq D_{\tau_{h_1}} \times \ldots \times D_{\tau_{h_m}}$ by

$$P = \{\langle [\![e_1]\!], \ldots, [\![e_m]\!]\rangle \mid \text{for some } j \in J_i, M_{i,j} \equiv x_h\ e_1\ \ldots\ e_m\},$$

and map x_h to the following function in ρ,

$$\lambda\, y_1 \ldots \lambda\, y_m \quad . \quad \textit{if there exists } p \in P \textit{ such that } p \sqsubseteq_{\tau_{h_1} \times \ldots \times \tau_{h_m}} \langle y_1, \ldots, y_m \rangle \textit{ then } 1 \textit{ else } 0.$$

If x_h never occurs in the conjunctive term $\bigsqcap_{j \in J_i} M_{i,j}$, then x_h is mapped to $\lambda\, y_1 \ldots \lambda\, y_m \,.\, 0$.

It is easy to see that $[\![\bigsqcap_{j \in J_i} M_{i,j}]\!]\rho = 1$. Thus $[\![\bigsqcup_{i \in I} \bigsqcap_{j \in J_i} M_{i,j}]\!]\rho = 1$. It remains to be shown that $[\![\bigsqcup_{k \in K} \bigsqcap_{l \in L_k} N_{k,l}]\!]\rho = 0$. Since for the given index $i \in I$, $\bigsqcap_{j \in J_i} M_{i,j} \preceq \bigsqcap_{l \in L_k} N_{k,l}$ is not provable for all indices $k \in K$, then, for each fixed $k \in K$, there exists an index $l \in L_k$ such that for all indices $j \in J_i$, $M_{i,j} \preceq N_{k,l}$ is not provable. There are two cases for $N_{k,l}$.

- $N_{k,l} \equiv x_h$, where x_h is in $\vec{x}$ and of type *2*. Thus x_h must not occur in the conjunctive term $\bigsqcap_{j \in J_i} M_{i,j}$. Otherwise we could have proved $M_{i,j} \preceq N_{k,l}$ for some $j \in J_i$, a contradiction. By the construction of ρ above, x_h is mapped to 0. Hence, $[\![\bigsqcap_{l \in L_k} N_{k,l}]\!]\rho = 0$.
- $N_{k,l} \equiv x_h\ f_1\ \ldots\ f_m$, where x_h is in $\vec{x}$ and of type $\sigma_h = \tau_{h_1} \to \ldots \tau_{h_m} \to 2$, and $f_1 \in \Lambda^0_{\tau_{h_1}}, \ldots, f_m \in \Lambda^0_{\tau_{h_m}}$. If x_h never occurs in $\bigsqcap_{j \in J_i} M_{i,j}$, then by the construction of ρ, $[\![N_{k,l}]\!]\rho = 0$. If x_h does occur in $\bigsqcap_{j \in J_i} M_{i,j}$ as $M_{i,j} \equiv x_h\ e_1\ \ldots\ e_m$ for some $j \in J_i$, where $e_1 \in \Lambda^0_{\tau_{h_1}}, \ldots, e_m \in \Lambda^0_{\tau_{h_m}}$. Then not all of $e_1 \preceq f_1, \ldots, e_m \preceq f_m$ are provable. Otherwise we could have proved $M_{i,j} \preceq N_{k,l}$ for some $j \in J_i$, a contradiction. By the induction hypothesis, either $[\![e_1]\!] \not\sqsubseteq [\![f_1]\!]$, ..., or $[\![e_m]\!] \not\sqsubseteq [\![f_m]\!]$. That is, $\langle [\![e_1]\!], \ldots, [\![e_m]\!] \rangle \not\sqsubseteq \langle [\![f_1]\!], \ldots, [\![f_m]\!] \rangle$. Then, by the construction of ρ above, $[\![N_{k,l}]\!]\rho = 0$. Hence, $[\![\bigsqcap_{l \in L_k} N_{k,l}]\!]\rho = 0$.

In all cases, $[\![\bigsqcap_{l \in L_k} N_{k,l}]\!]\rho = 0$ for every $k \in K$. It follows $[\![\bigsqcup_{k \in K} \bigsqcap_{l \in L_k} N_{k,l}]\!]\rho = 0$. This completes the proof. □

Example 3.7

Suppose we have the following two $\Lambda^0_{((2 \to 2) \to (2 \to 2)) \to (2 \to 2)}$ terms, Y and Z, defined by

$$\begin{aligned} Y &\equiv \lambda\, z \,.\, \lambda\, x \,.\, ((z\ (\lambda\, x \,.\, 0)\ 1) \sqcap x) \sqcup ((z\ (\lambda\, x \,.\, 0)\ 1) \sqcap (z\ (\lambda\, x \,.\, x)\ 0)), \\ Z &\equiv \lambda\, z \,.\, \lambda\, x \,.\, ((z\ (\lambda\, x \,.\, x)\ 1) \sqcap x) \sqcup (z\ (\lambda\, x \,.\, 0)\ 0). \end{aligned}$$

It can be checked that neither $Y \preceq Z$ nor $Z \preceq Y$ is provable. Therefore, we should be able to find elements $z, z' \in D_{(2 \to 2) \to (2 \to 2)}$ and $x, x' \in D_2$ such that

$$[\![Y]\!]\ z\ x\ =\ 1 \quad \text{but} \quad [\![Z]\!]\ z\ x\ =\ 0,$$

and

$$[\![Y]\!]\ z'\ x'\ =\ 0 \quad \text{but} \quad [\![Z]\!]\ z'\ x'\ =\ 1.$$

This will show that both $[\![Y]\!] \not\sqsubseteq [\![Z]\!]$ and $[\![Z]\!] \not\sqsubseteq [\![Y]\!]$.

The conjunctive term $(z\ (\lambda\, x \,.\, 0)\ 1) \sqcap (z\ (\lambda\, x \,.\, x)\ 0)$ in Y's body cannot be proved to be syntactically weaker than either one of the two conjunctive terms in Z's body.

It follows that we can choose

$$\begin{aligned} z &= \lambda f\,.\,\lambda x\,.\,\mathit{if}\ (\langle [\![\lambda x\,.\,0]\!], 1\rangle \sqsubseteq \langle f, x\rangle \text{ or } \langle [\![\lambda x\,.\,x]\!], 0\rangle \sqsubseteq \langle f, x\rangle)\ \mathit{then}\ 1\ \mathit{else}\ 0, \\ x &= 0, \end{aligned}$$

to make $[\![Y]\!]\ z\ x = 1$ but $[\![Z]\!]\ z\ x = 0$. Likewise, the witness of the unprovability of $Z \preceq Y$ is the conjunctive term $(z\ (\lambda x\,.\,x)\ 1) \sqcap x$ in Z's body. We then choose

$$\begin{aligned} z' &= \lambda f\,.\,\lambda x\,.\,\mathit{if}\ \langle [\![\lambda x\,.\,x]\!], 1\rangle \sqsubseteq \langle f, x\rangle\ \mathit{then}\ 1\ \mathit{else}\ 0, \\ x' &= 1, \end{aligned}$$

to make $[\![Y]\!]\ z'\ x' = 0$ but $[\![Z]\!]\ z'\ x' = 1$.

Corollary 3.8
Let $M, N \in \Lambda^0$. $M \simeq N$ implies $[\![M]\!] = [\![N]\!]$, and $[\![M]\!] = [\![N]\!]$ implies $M \simeq N$.

We show in Appendix B that $M \simeq N$ implies that M and N reduce to the same normal form (Lemma B.2). Because terms M and N in Corollary 3.8 are already in normal forms (they are in Λ^0), it follows that not only does $[\![M]\!] = [\![N]\!]$ imply $M \simeq N$, it also implies $M \equiv N$ (modulo associativity and commutativity of $\sqcup$ and $\sqcap$).

Theorem 3.9
Let $\sigma \in \Gamma$. Then there is a least fixed point term $Y \in \Lambda^0_{(\sigma\to\sigma)\to\sigma}$ such that for all terms $F \in \Lambda^0_{\sigma\to\sigma}$,

1. $[\![YF]\!] = \mathit{fix}\ [\![F]\!]$, where $\mathit{fix}\ \in D_{(\sigma\to\sigma)\to\sigma}$ is the least fixed point function, and
2. YF and $F(YF)$ reduce to the same normal form.

Proof

1. Because *fix* is a monotonic function for each type $(\sigma \to \sigma) \to \sigma \in \Gamma$, by Definition 2.3, we have $\mathit{fix} \in D_{(\sigma\to\sigma)\to\sigma}$. The existence of Y such that $[\![Y]\!] = \mathit{fix}$ is proved by Proposition 3.4. It follows that $[\![YF]\!] = [\![Y]\!][\![F]\!] = \mathit{fix}\ [\![F]\!]$.
2. First of all, $[\![YF]\!] = \mathit{fix}\ [\![F]\!] = [\![F]\!](\mathit{fix}\ [\![F]\!]) = [\![F]\!]([\![YF]\!]) = [\![F(YF)]\!]$. On the other hand, YF normalizes to a Λ^0 term M with $[\![YF]\!] = [\![M]\!]$ (Proposition 2.12 and 3.3). Similarly, $F(YF)$ normalizes to a Λ^0 term N with $[\![F(YF)]\!] = [\![N]\!]$. That is, $[\![M]\!] = [\![N]\!]$. It follows that $M \simeq N$ (Corollary 3.8), which implies $M \equiv N$ (Lemma B.2).

□

The above Theorem states that, not only does there exist a term Y whose denotation is the least fixed point function *fix* (clause 1 in Theorem 3.9), but whose *computational* characteristic also serves as a least fixed point term (clause 2 in Theorem 3.9).

The above properties of language Λ^0 enable us to syntactically calculate the least fixed point of a term $M \in \Lambda^0_{\sigma\to\sigma}$. In fact, there are two ways to do it. The first method uses Propositions 3.4 and Theorem 3.9 to find the least fixed point term $Y \in \Lambda^0_{(\sigma\to\sigma)\to\sigma}$ for the given type $\sigma \in \Gamma$, then calculate the normal form of YM. This normal form denotes the least fixed point of $[\![M]\!]$.

The second method uses an approximation sequence starting with $B_\sigma \in \Lambda^0_\sigma$, where $[\![B_\sigma]\!] = \bot_\sigma$. The iteration successively calculates a term $N^{(k)} \in \Lambda^0_\sigma$, where $N^{(0)} = B_\sigma$ and $N^{(k+1)}$ the normal form of $MN^{(k)}$, until it finds $N^{(i+1)} \preceq N^{(i)}$. Then term $N^{(i)}$ is the least fixed point of M. The iteration is guaranteed to be terminated because, for a given type $\sigma \in \Gamma$, there are only finite number of Λ^0_σ terms. Furthermore, the iteration will terminate as soon as possible because, by completeness (Proposition 3.6), $[\![N^{(i+1)}]\!] \sqsubseteq [\![N^{(i)}]\!]$ implies $N^{(i+1)} \preceq N^{(i)}$.

Example 3.10
Suppose we want to calculate the least fixed point of the following $\Lambda^0_{(2\to 2)\to(2\to 2)}$ term M, defined as

$$M \equiv \lambda f \,.\, \lambda x \,.\, (f\ 0) \sqcup ((f\ 1) \sqcap x).$$

By Proposition 3.4, we can find a least fixed point term Y in $\Lambda^0_{((2\to 2)\to(2\to 2))\to(2\to 2)}$. For example,

$$Y \equiv \lambda z \,.\, \lambda x \,.\, ((z\ (\lambda x \,.\, 0)\ 1) \sqcap x) \sqcup ((z\ (\lambda x \,.\, 0)\ 1) \sqcap (z\ (\lambda x \,.\, x)\ 0)),$$

as defined in Example 3.5 is such a term. Furthermore, YM normalizes to $\lambda x \,.\, 0$, which is the least fixed point of M.

We can also use a least fixed point approximation sequence starting with $\lambda x \,.\, 0$, which denotes the least element in $D_{2\to 2}$. Then $\lambda x \,.\, 0$ is the least fixed point of M because $M\ (\lambda x \,.\, 0) \to^* \lambda x \,.\, 0$.

Syntactic approaches to fixed point finding have been attempted before, see for example Clack and Peyton Jones (1985) and Martin (1989), but with no success for higher-order functions. The problem is that, if a naive approach is used, for a given element in the higher-order function space, there would be many syntactic normal forms denoting the element. But these terms are not comparable, syntactically, to one another. This makes impossible an iterative process for finding the fixed point.

Our approach avoids this problem by using the restricted language Λ^0 which is expressive enough (Proposition 3.4) yet with syntactic comparison rules that are complete with respect to the semantics (Proposition 3.6). Furthermore, there are only a finite number of Λ^0 terms for a given type (Definition 3.1), and terms in Λ^0 are closed under the usual syntactic reductions (Proposition 3.3). Hence we are sure that the fixed point iteration will converge, and the convergent term indeed denotes the fixed point value.

4 Implementation issues

In this section, we briefly consider several implementation issues when using a syntactic approach based on language Λ^0 in computing least fixed points of the abstract functions derived from functional programs. One immediate concern is that, although the abstract semantics of a functional program can be easily described in language Λ, it is not necessary so in language Λ^0. In short, we must have a way to translate a closed term in Λ into an equivalent term in Λ^0 before we can compute its least fixed point. We also describe how we can relax language Λ^0 to

a new language Λ^1 to greatly reduce the translation process. Language Λ^1 is no longer complete. However, we show that the syntactic approach can be adapted to compute least fixed points on Λ^1 too. We then consider the problem of embedding other distributive finite domains into domains $D_\sigma, \sigma \in \Gamma$. We also mention an approximation technique which speeds up the least fixed point iteration but computes less accurate results.

4.1 Translation from language Λ to language Λ^0

We describe in the following a systematic way to translate a closed term in Λ into a semantically equivalent term in Λ^0. We first assume that a closed term $N \in \Lambda$ has been reduced to its normal form $\lambda \vec{x} \,.\, M$. Furthermore, by η equality (i.e. $[\![M]\!] = [\![\lambda x_\tau \,.\, Mx]\!]$ for M of type $\tau \to \gamma$) we can assume that N can be written as $\lambda \vec{x} \,.\, \bigsqcup_{i \in I} \bigsqcap_{j \in J_i} M_{i,j}$ where each $M_{i,j}$ is of type $\mathit{2}$. The problem is that for terms $M_{i,j} \equiv x_i \; e_1 \; \ldots \; e_m$, where variable x_i is in $\vec{x}$ and of type $\sigma_i = \tau_{i_1} \to \ldots \to \tau_{i_m} \to \mathit{2}$, terms $e_k, 1 \le k \le m$, are not necessary in $\Lambda^0_{\tau_{i_k}}$.

However, function application itself is definable in language Λ^0, and we can use this definition to turn each term $M_{i,j}$ above into an equivalent term where each e_k is in $\Lambda^0_{\tau_{i_k}}$. The translation proceeds until all the function applications in a term satisfy the requirements of Λ^0. We also perform $\boldsymbol{\beta} \sqcup \sqcap \boldsymbol{d}$-reduction during the translation process.

Example 4.1
Suppose we have a term M in $\Lambda_{(\mathit{2}\to\mathit{2}\to\mathit{2})\to\mathit{2}\to\mathit{2}\to\mathit{2}}$, defined as

$$M \equiv \lambda f \,.\, \lambda x \,.\, \lambda y \,.\, f \;\; (x \sqcap y) \;\; (x \sqcup y).$$

We want to translate M into a term in $\Lambda^0_{(\mathit{2}\to\mathit{2}\to\mathit{2})\to\mathit{2}\to\mathit{2}\to\mathit{2}}$. First, we observe that the function $\mathit{apply} \;\in D_{(\mathit{2}\to\mathit{2})\to\mathit{2}\to\mathit{2}}$, where $\mathit{apply} \; f \; x = f \; x$ for all $f \in D_{\mathit{2}\to\mathit{2}}$ and all $x \in D_{\mathit{2}}$, can be defined in $\Lambda^0_{(\mathit{2}\to\mathit{2})\to\mathit{2}\to\mathit{2}}$ by

$$\lambda f \,.\, \lambda x \,.\, (f \; 0) \sqcup ((f \; 1) \sqcap x).$$

By the same principle, the function $\mathit{Apply} \;\in D_{(\mathit{2}\to\mathit{2}\to\mathit{2})\to\mathit{2}\to\mathit{2}\to\mathit{2}}$ can be defined in Λ^0 by

$$\lambda f \,.\, \lambda x \,.\, \lambda y \,.\, (f \; 0 \; 0) \sqcup ((f \; 1 \; 0) \sqcap x) \sqcup ((f \; 0 \; 1) \sqcap y) \sqcup ((f \; 1 \; 1) \sqcap x \sqcap y).$$

Note that in the above two definitions, the function variable f is applied only to Λ^0 terms but not to variables x or y.

Now, the following term M' has the same semantics as term M,

$$M' \equiv \lambda f \,.\, \lambda x \,.\, \lambda y \,.\, \mathit{Apply} \; f \;\; (x \sqcap y) \;\; (x \sqcup y).$$

After substituting the Λ^0 definition of *Apply* and normalizing, we derive

$$M' \to^* \lambda f \,.\, \lambda x \,.\, \lambda y \,.\, (f \; 0 \; 0) \sqcup ((f \; 1 \; 1) \sqcap x \sqcap y),$$

which is semantically equivalent to M and in $\Lambda^0_{(\mathit{2}\to\mathit{2}\to\mathit{2})\to\mathit{2}\to\mathit{2}\to\mathit{2}}$.

As shown by the above example, Λ^0 terms like *apply* and *Apply* are used to make function variables like f accept only Λ^0 terms as arguments after the translation. The entire translation process can be performed automatically if the apply functions are supplied beforehand in $\Lambda^0_{\sigma\to\sigma}$ terms for each type $\sigma \in \Gamma$. We simply repeat the translation, each time by using apply functions of lower ranks, until the resultant term is in language Λ^0.

4.2 Relaxation of language Λ^0 *to language* Λ^1

We observe that, for a closed term $M \in \Lambda$ in its normal form, a semantically equivalent term $M' \in \Lambda^0$ can be exponentially longer than M. See, for example, function *apply* and *Apply* in Example 4.1. This makes the syntactic method somewhat unattractive. Furthermore, much work is spent in translating term M into M'. We now define a language Λ^1 to address this problem. Language Λ^1 is a superset of Λ^0, looks more like usual functional languages (hence, needs less translation work), and often provides shorter terms than those in Λ^0.

Definition 4.2
The sub-language Λ^1_σ of Λ_σ is inductively defined for each type $\sigma \in \Gamma$ as follows.

- $0, 1 \in \Lambda^1_{\mathit{2}}$; and
- $\lambda\,\vec{x}\,.\,M \in \Lambda^1_{\sigma_1\to\ldots\to\sigma_n\to\mathit{2}}$ if
  - $\vec{x}$ consists of variables of types $\sigma_1, \ldots, \sigma_n$, and M contains no free variable other than those from $\vec{x}$,
  - $M \in \Lambda_{\mathit{2}}$ and it is in $\boldsymbol{\beta} \sqcup \sqcap\boldsymbol{d}$ normal form $\bigsqcup_{i\in I} \bigsqcap_{j\in J_i} M_{i,j}$, and
  - each term $M_{i,j}$ is either a bound variable of type $\mathit{2}$, or an application of the form $(x_i\ e_1\ \ldots\ e_m)$, where variable x_i is in $\vec{x}$ and is of type $\sigma_i = \tau_{i_1} \to \ldots \to \tau_{i_m} \to \mathit{2}$, and term e_k is either a variable from $\vec{x}$ or is of type $\Lambda^1_{\tau_{i_k}}$ for each $1 \le k \le m$.

The only difference between Λ^1 and Λ^0 is that the bound variables of a Λ^1 term are allowed to appear as function arguments. In doing so, we lose the completeness property. For example, the two terms $\lambda\,f\,.\,\lambda\,x\,.\,f\ x$ and $\lambda\,f\,.\,\lambda\,x\,.\,(f\ 0) \sqcup ((f\ 1) \sqcap x)$ are in language $\Lambda^1_{(2\to2)\to2\to2}$, are semantically equivalent, but cannot be shown to be syntactically equivalent under the $\simeq$ relation.

However, note that, for a given type $\sigma \in \Gamma$, there are still a finite number of terms in Λ^1_σ, and for terms $M, N \in \Lambda^1$, the normal form for MN is also in Λ^1. From these two important properties, we can show that the two methods described in section 3 for computing the least fixed points still work. That is, when applying $Y \in \Lambda^0_{(\sigma\to\sigma)\to\sigma}$, the term for least fixed point finding, to a $\Lambda^1_{\sigma\to\sigma}$ term M, the resulting normal form will be a Λ^1 term denoting the least fixed point of M. The iterative method is guaranteed to converge as well because Λ^1 is still sound and there are only a finite number of Λ^1_σ terms to iterate with. (However, since language Λ^1 is no longer complete, it may happen that the iteration oscillates between two semantically equivalent but syntactically incomparable Λ^0 terms. This problem can be solved by comparing the current term of approximation to each of the previous

terms, not just the immediately previous term, in the iteration history to see if it has re-appeared.)

To show that terms in language Λ^1 can be much shorter than terms in Λ^0 and much closer to the usual functional programs, let us consider, for example, the higher-order *if* functional, which receives three arguments (the first is a boolean condition, the remaining two are functions) and will return one of the two functions depending on the boolean condition. The strictness property of this higher-order *if* functional is in domain $D_{2\to\sigma\to\sigma\to\sigma}$ with some type $\sigma = \tau_1 \to \tau_2 \to \ldots \to \tau_n \to 2$, and can be defined in language $\Lambda^1_{2\to\sigma\to\sigma\to\sigma}$ by

$$\lambda\, p\,.\,\lambda\, f\,.\,\lambda\, g\,.\,\lambda\, x_1\,.\,\lambda\, x_2 \ldots .\,\lambda\, x_n\,.\,(p \sqcap (f\ x_1\ x_2\ \ldots\ x_n)) \sqcup (p \sqcap (g\ x_1\ x_2\ \ldots\ x_n)),$$

where p is of type 2, f and g of type $\sigma = \tau_1 \to \tau_2 \to \ldots \to \tau_n \to 2$, and x_i of type τ_i for $1 \le i \le n$. It is clear that the translation of the above term in language $\Lambda^0_{2\to\sigma\to\sigma\to\sigma}$ will be longer because we must translate $(f\ x_1\ x_2\ \ldots\ x_n)$ and $(g\ x_1\ x_2\ \ldots\ x_n)$ to longer terms to satisfy the requirement of Λ^0.

Example 4.3

The term $F \equiv \lambda\, f\,.\,\lambda\, x\,.\,f\ x$ is in $\Lambda^1_{(2\to 2)\to 2\to 2}$. By using the fixed point term Y defined in Example 3.5, we arrive at $\lambda\, x\,.\,0$ as the normal form of YF, and thus as the least fixed point of F. The approximation method also finds $\lambda\, x\,.\,0$ as the least fixed point.

Clack and Peyton Jones (1985) give a rather interesting example. It is a term H in language $\Lambda_{((2\to 2)\to 2)\to(2\to 2)\to 2}$, defined as

$$H \equiv \lambda\, f\,.\,\lambda\, g\,.\,g\ (f\ g),$$

where variable f is of type $(2 \to 2) \to 2$ and g of type $2 \to 2$. If we naively perform an approximation sequence starting with $B \equiv \lambda\, g\,.\,0$ to calculate the least fixed point of H, we will get

$$\begin{aligned} HB &\to^* \lambda\, g\,.\,g\ 0, \\ H(HB) &\to^* \lambda\, g\,.\,g\ (g\ 0), \\ H(H(HB)) &\to^* \lambda\, g\,.\,g\ (g\ (g\ 0)), \\ &\ldots \end{aligned}$$

and the sequence will not converge under relation $\preceq$. However, if H is translated (as described in Example 4.1) into a semantically equivalent term H^1 in Λ^1 as

$$H^1 \equiv \lambda\, f\,.\,\lambda\, g\,.\,(g\ 0) \sqcup ((g\ 1) \sqcap (f\ g)).$$

The approximation sequence will reach $\lambda\, g\,.\,g\ 0$ as the least fixed point of H^1.

We can also translate term H^1 into a term H^0 in language Λ^0 as

$$H^0 \equiv \lambda\, f\,.\,\lambda\, g\,.\,(g\ 0) \sqcup ((g\ 1) \sqcap (f\ (\lambda\, x\,.\,x))),$$

then calculate the least fixed point of H^0.

4.3 Embedding of other finite domains

The finite domains $D_\sigma, \sigma \in \Gamma$, and its language Λ are very useful when, for example, computing the strictness property of functional programs over flat basic domains, because the basic abstract domain for strictness happens to be $\mathit{2} = \{0, 1\}$. But are they flexible enough for other abstract domains based not on $\mathit{2}$? For example, we may like to have abstract domains based on domain $\mathit{3} = \{\bot, \|, \top\}$, with the ordering $\bot \sqsubseteq_{\mathit{3}} \| \sqsubseteq_{\mathit{3}} \top$, and define abstract semantics of programs accordingly. Are domains $D_\sigma, \sigma \in \Gamma$, along with language Λ, helpful in such situations?

Figure 1 provides a good hint. We can embed domain $\mathit{3}$ into domain $\mathit{2} \to \mathit{2}$ by the encoding $\bot = [0 \mapsto 0, 1 \mapsto 0]$, $\| = [0 \mapsto 0, 1 \mapsto 1]$, and $\top = [0 \mapsto 1, 1 \mapsto 1]$, and define the three elements of $\mathit{3}$ in language $\Lambda^0_{2\to 2}$ accordingly as $\bot = [\![\lambda x \,.\, 0]\!]$, $\| = [\![\lambda x \,.\, x]\!]$, and $\top = [\![\lambda x \,.\, 1]\!]$. Least fixed point computations on the domains of function spaces generated by $\mathit{3}$ can then be carried out in Λ^0 as usual. As another example, we can also embed the boolean domain $\mathit{bool} = \{\bot, t, f\}$, with the ordering $\bot \sqsubseteq t$ and $\bot \sqsubseteq f$ but neither $f \sqsubseteq t$ nor $t \sqsubseteq f$, in domain $\mathit{2} \to \mathit{2} \to \mathit{2}$ by $\bot = [\![\lambda x \,.\, \lambda y \,.\, x \sqcap y]\!]$, $t = [\![\lambda x \,.\, \lambda y \,.\, x]\!]$, and $f = [\![\lambda x \,.\, \lambda y \,.\, y]\!]$. However, since domains $D_\sigma, \sigma \in \Gamma$, are always distributive, only distributive finite domains can be properly embedded in D_σ.

We might wish to perform least fixed point computations directly on domain $\mathit{3}$, and the function spaces generated by it, using a new language without the explicit encoding of the three elements in language $\Lambda^0_{2\to 2}$. For example, we might want to define a new language based only on the three constants $\bot, \|, \top$ and the two operations $\sqcup$ and $\sqcap$ between them (without incorporating $0, 1$ and their $\sqcup$ and $\sqcap$ operations). Moreover, the inference rules for syntactically weaker and the reduction rules for normalization are similarly adapted to perform computations on this new language. However, it turns out that this is not a vital approach, at least if the syntax and reduction rules on Λ are not greatly changed. As an example, Figure 1 shows that there are 10 elements in domain $\mathit{3} \to \mathit{3}$; but the language $\Lambda^0_{3\to 3}$, based on the above idea, can only represent six of them: $\lambda x \,.\, \bot$, $\lambda x \,.\, \|$, $\lambda x \,.\, \top$, $\lambda x \,.\, x$, $\lambda x \,.\, x \sqcup \|$, and $\lambda x \,.\, x \sqcap \|$. The new language is not able to express the element $[\bot \mapsto \bot, \ \| \mapsto \bot, \ \top \mapsto \|]$ in domain $\mathit{3} \to \mathit{3}$, for example. The difficulty seems to arise from the fact that the new language cannot even define the *step* functions (See Proposition 3.4).

4.4 Complexity and approximation

With respect to the efficiency of the proposed syntactic method, is it better than simply using the frontier method? We believe it is better, especially if we use the iterative method on language Λ^1. We observe that a functional program often has simple textual structure. That is, function applications in typical functional programs often have bound variables as arguments. This makes the translation from a functional program into a Λ^1 term easy, and the length of the resultant Λ^1 term comparable to the length of the original program. The time spent in the $\preceq$ relationship testing between two successive approximation is then comparable to the cases

```
append []      ys  =  ys
append (x::xs) ys  =  x::(append xs ys)

foldr f []      y  =  y
foldr f (x::xs) y  =  f x (foldr f xs y)

cat xss  =  foldr append xss []
```

Fig. 2. The definitions of functions `append`, `foldr`, and `cat`.

in the frontier method. Furthermore, an approximation sequence in the syntactic method usually reaches the least fixed point in fewer iterations than an approximation sequence in the frontier method, because the former utilizes the textual information from the program and performs symbolic simplification, while the latter searches along the semantic domains, no matter what the given program looks like.

Of course, whether a scheme for strictness analysis is 'fast' as compared to other schemes will depend not only on the different approaches they take but also on the particular implementations they adapt. We emphasizes here on a viable alternative for strictness analysis based on a new syntactic approach, rather than on how fast our implementation of the syntactic method is. Also notice that, in the worst case, the syntactic method will, just as the frontier method, require exponential running time (with respect to the program length) even for first-order functions. Wang (1989) further shows that, for a special class of second-order functions excluding *if* functionals, strictness analysis is already *NP*-hard.

An example involving an abstract domain of at least 10^6 elements is elaborated in the next section, using the proposed syntactic method. This example was previously used by Hunt (1989) to show that a naive frontier method is not effective when dealing with abstract domain of considerable size. Using the proposed syntactic method, we are able to compute least fixed points successfully.

Safe approximation schemes can also be developed based on the proposed syntactic method. The approximation scheme will calculate less accurate fixed points, but do it in fewer iterations. For example, a reduction rule like

$$(M \sqcap N) \rightarrow_{\sqcap_{\text{approx}}} M,$$

when $N \preceq M$ is not provable, can be used to speed up the approximation sequence to arrive at a less accurate approximation of the least fixed point.

5 Strictness analysis over non-flat domains: An example

In this section, we use the proposed syntactic method to solve a substantial problem. The problem is to derive the strictness property of a functional program which concatenates a list of lists into a single list. This function, named `cat`, is composed from a right-associative `foldr` function and the usual `append` function. The whole program text is in Figure 2.

The abstraction techniques for strictness analysis from Burn, Hankin, and Abram-

sky (1986) and Wadler (1987) are used to solve this problem. Recall that Burn, Hankin, and Abramsky define an abstraction of a domain as a lattice of its non-empty Scott-closed subsets, where the Scott-closed subset are ordered by subset inclusion in the lattice. For strictness analysis on a basic domain D, D is usually abstracted to the domain with only two Scott-closed subsets: $\{\bot_D\}$ and D. This abstract domain is usually written as $\mathit{2}$. Its two elements are defined as $0_{\mathit{2}} = \{\bot_D\}$ and $1_{\mathit{2}} = D$, with the ordering $0_{\mathit{2}} \sqsubseteq 1_{\mathit{2}}$. (More precisely, we may say that element 0_{2} has set $\{\bot_D\}$ as its concretization, and 1_{2} has set D as its concretization.) If an element $d \in D$ has abstraction $d^{\#} = 0_{\mathit{2}}$, we know for sure that d is the undefined element in domain D. If $d^{\#} = 1_{\mathit{2}}$, then d can be any element in D. Note that 0_{2} and 1_{2} can be expressed as 0 and 1 in language Λ^0.

Similarly, Wadler defines an abstract domain $\mathit{4}$ for domain list(D), which contains all the lists whose elements are from domain D. The abstract domain $\mathit{4}$ has four elements: $0_{\mathit{4}}, 1_{\mathit{4}}, 2_{\mathit{4}}$, and 3_{4}. They are defined as the following Scott–closed subsets of list(D):

$$
\begin{aligned}
0_{\mathit{4}} &= \{\bot_{\mathrm{list}(D)}\},\\
1_{\mathit{4}} &= 0_{\mathit{4}} \cup \{e \in \mathrm{list}(D) \mid e \text{ has an undefined or infinite tail}\},\\
2_{\mathit{4}} &= 1_{\mathit{4}} \cup \{e \in \mathrm{list}(D) \mid e \text{ is of finite length but contains at least one element from } 0_{\mathit{2}}\},\\
3_{\mathit{4}} &= \mathrm{list}(D),
\end{aligned}
$$

with the ordering $0_{\mathit{4}} \sqsubseteq 1_{\mathit{4}} \sqsubseteq 2_{\mathit{4}} \sqsubseteq 3_{\mathit{4}}$. The abstract versions of the three primitive functions `hd`, `tl`, and `cons` on domain list(D) are described in Table 2. For example, if $xs^{\#}_{\mathit{4}} = 2_{\mathit{4}}$ – meaning that xs is the undefined list, or a list with an undefined or infinite tail, or a finite-length list with at least one list element undefined – then $tl^{\#}_{\mathit{4}\to\mathit{4}}\ xs^{\#}_{\mathit{4}} = 3_{\mathit{4}}$ – meaning that the tail of xs can be any list.

An abstract domain $\mathit{6}$ for domain list(list(D)) are also defined, with its abstract elements as the following:

$$
\begin{aligned}
0_{\mathit{6}} &= \{\bot_{\mathrm{list}(\mathrm{list}(D))}\},\\
1_{\mathit{6}} &= 0_{\mathit{6}} \cup \{e \in \mathrm{list}(\mathrm{list}(D)) \mid e \text{ has an undefined or infinite tail}\},\\
2_{\mathit{6}} &= 1_{\mathit{6}} \cup \{e \in \mathrm{list}(\mathrm{list}(D)) \mid e \text{ is of finite length but contains at least one element from } 0_{\mathit{4}}\},\\
3_{\mathit{6}} &= 2_{\mathit{6}} \cup \{e \in \mathrm{list}(\mathrm{list}(D)) \mid e \text{ is of finite length but contains at least one element from } 1_{\mathit{4}}\},\\
4_{\mathit{6}} &= 3_{\mathit{6}} \cup \{e \in \mathrm{list}(\mathrm{list}(D)) \mid e \text{ is of finite length but contains at least one element from } 2_{\mathit{4}}\},\\
5_{\mathit{6}} &= \mathrm{list}(\mathrm{list}(D)),
\end{aligned}
$$

and with the ordering $0_{\mathit{6}} \sqsubseteq 1_{\mathit{6}} \sqsubseteq 2_{\mathit{6}} \sqsubseteq 3_{\mathit{6}} \sqsubseteq 4_{\mathit{6}} \sqsubseteq 5_{\mathit{6}}$. Table 3 contains the abstraction of functions `hd`, `tl`, and `cons` over domain $\mathit{6}$.

Note that not only is domain $\mathit{4}$ a distributive lattice, but it is also isomorphic to domain $(\mathit{2} \to \mathit{2}) \to \mathit{2}$. This means that we can express the four elements in domain

Table 2. *The abstraction of the primitive functions* `hd`, `tl`, *and* `cons` *over the finite domain* 4

$xs_4^\#$	$hd_{4\to 2}^\#\ xs_4^\#$	$tl_{4\to 4}^\#\ xs_4^\#$	$cons_{2\to 4\to 4}^\#\ x_2^\#\ xs_4^\#$	
			$x_2^\# = 0_2$	$x_2^\# = 1_2$
0_4	0_2	0_4	1_4	1_4
1_4	1_2	1_4	1_4	1_4
2_4	1_2	3_4	2_4	2_4
3_4	1_2	3_4	2_4	3_4

Table 3. *The abstraction of the primitive functions* `hd`, `tl`, *and* `cons` *over the finite domain* 6.

$xss_6^\#$	$hd_{6\to 4}^\#\ xss_6^\#$	$tl_{6\to 6}^\#\ xss_6^\#$	$cons_{4\to 6\to 6}^\#\ xs_4^\#\ xss_6^\#$			
			$xs_4^\# = 0_4$	$xs_4^\# = 1_4$	$xs_4^\# = 2_4$	$xs_4^\# = 3_4$
0_6	0_4	0_6	1_6	1_6	1_6	1_6
1_6	3_4	1_6	1_6	1_6	1_6	1_6
2_6	3_4	5_6	2_6	2_6	2_6	2_6
3_6	3_4	5_6	2_6	3_6	3_6	3_6
4_6	3_4	5_6	2_6	3_6	4_6	4_6
5_6	3_4	5_6	2_6	3_6	4_6	5_6

4 as the following terms in language $\Lambda^0_{(2\to 2)\to 2}$:

$$\begin{aligned} 0_4 &= [\![\lambda\, x_{2\to 2}\,.\, 0]\!], \\ 1_4 &= [\![\lambda\, x_{2\to 2}\,.\, x\, 0]\!], \\ 2_4 &= [\![\lambda\, x_{2\to 2}\,.\, x\, 1]\!], \\ 3_4 &= [\![\lambda\, x_{2\to 2}\,.\, 1]\!]. \end{aligned}$$

Furthermore, recall that, by completeness (Proposition 3.6), the syntactically weaker relationship $\preceq$ in language $\Lambda^0_{(2\to 2)\to 2}$ captures exactly the semantically weaker relationship $\sqsubseteq$ in domain 4. The strictness properties of the primitive functions `hd`, `tl`, and `cons` can be defined in language Λ^0 too. The same property also applies to domain 6, which can be embedded in domain $(((2 \to 2) \to 2) \to 2) \to 2$. The definitions of these semantic elements in language Λ^0 are summarized in Table 4.

Given the definitions in Table 4, we then can compute the strictness properties of functions `append`, `foldr`, and `cat`. Wadler's technique takes pattern-matching on input arguments into account when analyzing a functional program. Take function `append` as an example. Suppose that the first argument to `append` is only as defined

$$
\begin{array}{lll}
\multicolumn{3}{l}{append^{\#}_{4\to 4\to 4}\ x^{\#}_{4}\ y^{\#}_{4} =} \\
 & if & 3_4 \sqsubseteq x^{\#} \\
 & then & \bigsqcup\{y^{\#},\ \ cons^{\#}\ 1_2\ (append^{\#}\ 3_4\ y^{\#})\} \\
else & if & 2_4 \sqsubseteq x^{\#} \\
 & then & \bigsqcup\{cons^{\#}\ 0_2\ (append^{\#}\ 3_4\ y^{\#}),\ \ cons^{\#}\ 1_2\ (append^{\#}\ 2_4\ y^{\#})\} \\
else & if & 1_4 \sqsubseteq x^{\#} \\
 & then & cons^{\#}\ 1_2\ (append^{\#}\ 1_4\ y^{\#}) \\
else & & 0_4
\end{array}
\tag{2}
$$

$$
\begin{array}{l}
append^{\#} = fix\ (\lambda f_{4\to 4\to 4}\,.\,\lambda x_4\,.\,\lambda y_4\,.\,\lambda z_3\,. \\
\quad \bigsqcup\{\ \ \bigsqcap\{x\,2_3,\ f\,1_4\,y\,0_3\}, \\
\qquad\ \ \bigsqcap\{x\,2_3,\ f\,1_4\,y\,1_3,\ z\,1_2\}, \\
\qquad\ \ \bigsqcap\{x\,2_3,\ z\,0_2\}, \\
\qquad\ \ \bigsqcap\{x\,1_3,\ f\,3_4\,y\,1_3,\ z\,1_2\}, \\
\qquad\ \ \bigsqcap\{x\,1_3,\ f\,2_4\,y\,0_3\}, \\
\qquad\ \ \bigsqcap\{x\,0_3,\ f\,3_4\,y\,0_3\}, \\
\qquad\ \ \bigsqcap\{x\,0_3,\ y\,z\}\})
\end{array}
\tag{3}
$$

$$
\begin{array}{l}
append^{\#} = \lambda x_4\,.\,\lambda y_4\,.\,\lambda z_3\,. \\
\quad \bigsqcup\{\ \ \bigsqcap\{x\,2_3,\ z\,0_2\}, \\
\qquad\ \ \bigsqcap\{x\,1_3,\ y\,1_3,\ z\,1_2\}, \\
\qquad\ \ \bigsqcap\{x\,0_3,\ y\,0_3\}\}
\end{array}
\tag{4}
$$

Fig. 3. Strictness analysis for function `append`.

Note. Equation (2) describes the strictness property of `append`, according to its program text, in which Wadler's abstraction mechanism is used. Equation (3) expresses `append`'s strictness property as the least fixed point of a Λ^1 term. The least fixed point is then calculated by an approximation sequence by using the syntactic calculus starting from term $\lambda x_4\,.\,\lambda y_4\,.\,\lambda z_3\,.\,0_2$, the least element in domain $4 \to 4 \to 4$. The result is then translated into a Λ^0 term in Equation (4).

as 2_4 (i.e. $2_4 \sqsubseteq (x :: xs)^{\#}$ but $3_4 \not\sqsubseteq (x :: xs)^{\#}$). What will be the strictness properties for x and xs? By consulting the valuation table for $cons^{\#}$ in Table 2 for entries 2_4, we know that it suffices to consider either the case of $2_4 = cons^{\#}\ 0_2\ 3_4$, or the case of $2_4 = cons^{\#}\ 1_2\ 2_4$. (The case for $2_4 = cons^{\#}\ 0_2\ 2_4$ is covered by both of the above two cases.) Therefore, the analysis must take both cases into

consideration. That is, the final result should be the least upper bound of the two cases. (See the if–clause with condition $2_4 \sqsubseteq x^{\#}$ in Equation (2) in Figure 3.) Note that this yields more information than simply evaluating $hd^{\#}\ 2_4 = 1_2$ and $tl^{\#}\ 2_4 = 3_4$ to get the strictness of x and xs, which tells us nothing at all in this particular case. The strictness of *append* can then be expressed as Equation (2) in Figure 3.

If a semantic method is to be used, then the least fixed point semantics for $append^{\#}$ can now be evaluated by an approximation iteration starting with the least element in domain $4 \to 4 \to 4$. Since we are interested in the syntactic method, instead we write $append^{\#}$ as the least fixed point of a Λ^1 term, as Equation (3) in Figure 3. The Λ^1 terms is obtained by a simple translation from Equation (2) by substituting $cons^{\#}$ by its definition from Table 4, expanding the *if* statements (*i.e.*, the *step* functions in Proposition 3.4), and normalization. The least fixed point of this Λ^1 term can be calculated by an approximation sequence starting from $\lambda x_4 . \lambda y_4 . \lambda z_3 . 0_2$, the least defined term in language $\Lambda^1_{4\to 4\to 4}$. The resulting least fixed point will be a Λ^1 term. Equation (4) in Figure 3 is its translation in the Λ^0 language. It is the strictness property of the `append` function.

The same process can apply to the `foldr` function, resulting in the Λ^0 term in Figure 4 as its strictness property. Substituting the definition of $append^{\#}$ and $foldr^{\#}$ in the definition of $cat^{\#}$ and normalizing the result (Equation (5) in Figure 5) yields the strictness property of `cat` (Equation (6) in Figure 5). To see how accurate $cat^{\#}$ is, we can apply it to terms $0_6, 1_6, 2_6, 3_6, 4_6$, and 5_6, resulting Table 5. The result is as good as we can hope for, showing that Wadler's abstraction mechanism for list domains is quite accurate for this particular example.

All these results are calculated by a bare-bone Standard ML program which is used to normalize a given Λ^1 term in a naive way, and to evaluate the least fixed point of a given Λ^1 term by a syntactic approximation iteration. The strictness property of program `foldr`, as described in figure 4, takes about 20 minutes to calculate when running the least fixed point finding program under SML of New Jersey (version 0.66) on a Sun 4/290 with 32 MB physical memory.

6 Conclusion and comparison to other works

We have shown how to develop a syntactic approach, based on the language Λ^0, for finding least fixed points of higher-order functions over finite domains. This syntactic method is sound and complete with respect to the semantics of least fixed point computation on finite domains, and bears close relationship to the simply typed λ-calculus.

It is interesting to compare the development here with the work of Abramsky (1991) and Jensen (1990; 1991). Their work also provides a junction between semantics and logics for functional programming languages. Their work is mostly concerned with the dual relationship between domain theory and its axiomatic logics; ours is concerned with least fixed points on finite domains and their corresponding calculus. While their work usually provides a decidable theory without

$$
\begin{aligned}
\mathit{foldr}^{\#} = {} & \lambda\, f_{4\to 4\to 4}\,.\,\lambda\, x_{6}\,.\,\lambda\, y_{4}\,.\,\lambda\, z_{3}\,.\\
& \bigsqcup\{\ \prod\{x\,4_5,\ z\,0_2,\ f\,3_4\,0_4\,2_3\},\\
& \quad \prod\{x\,4_5,\ z\,1_2,\ f\,3_4\,1_4\,1_3,\ f\,3_4\,0_4\,2_3\},\\
& \quad \prod\{x\,4_5,\ f\,3_4\,2_4\,0_3,\ f\,3_4\,1_4\,1_3,\ f\,3_4\,0_4\,2_3\},\\
& \quad \prod\{x\,3_5,\ f\,0_4\,2_4\,2_3,\ z\,0_2,\ y\,1_3\},\\
& \quad \prod\{x\,3_5,\ f\,0_4\,3_4\,2_3,\ z\,0_2,\ f\,3_4\,2_4\,0_3,\ y\,1_3\},\\
& \quad \prod\{x\,3_5,\ f\,0_4\,3_4\,1_3,\ z\,1_2,\ y\,0_3\},\\
& \quad \prod\{x\,3_5,\ f\,0_4\,3_4\,2_3,\ z\,0_2,\ y\,0_3\},\\
& \quad \prod\{x\,3_5,\ z\,0_2,\ f\,0_4\,1_4\,2_3,\ y\,2_3\},\\
& \quad \prod\{x\,3_5,\ z\,1_2,\ f\,0_4\,2_4\,2_3,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \quad \prod\{x\,3_5,\ z\,1_2,\ f\,3_4\,1_4\,1_3,\ f\,0_4\,3_4\,2_3,\ y\,0_3\},\\
& \quad \prod\{x\,3_5,\ z\,1_2,\ f\,0_4\,2_4\,1_3,\ y\,1_3\},\\
& \quad \prod\{x\,3_5,\ f\,0_4\,3_4\,2_3,\ f\,3_4\,2_4\,0_3,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \quad \prod\{x\,3_5,\ f\,0_4\,3_4\,1_3,\ f\,3_4\,2_4\,0_3,\ y\,1_3\},\\
& \quad \prod\{x\,3_5,\ f\,0_4\,3_4\,0_3,\ y\,0_3\},\\
& \quad \prod\{x\,2_5,\ f\,1_4\,2_4\,2_3,\ z\,0_2,\ y\,1_3\},\\
& \quad \prod\{x\,2_5,\ f\,1_4\,3_4\,2_3,\ z\,0_2,\ f\,3_4\,2_4\,0_3,\ y\,1_3\},\\
& \quad \prod\{x\,2_5,\ f\,1_4\,3_4\,1_3,\ z\,1_2,\ y\,0_3\},\\
& \quad \prod\{x\,2_5,\ f\,1_4\,3_4\,2_3,\ z\,0_2,\ y\,0_3\},\\
& \quad \prod\{x\,2_5,\ z\,0_2,\ f\,1_4\,1_4\,2_3,\ y\,2_3\},\\
& \quad \prod\{x\,2_5,\ z\,1_2,\ f\,1_4\,2_4\,2_3,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \quad \prod\{x\,2_5,\ z\,1_2,\ f\,3_4\,1_4\,1_3,\ f\,1_4\,3_4\,2_3,\ y\,0_3\},\\
& \quad \prod\{x\,2_5,\ z\,1_2,\ f\,1_4\,2_4\,1_3,\ y\,1_3\},\\
& \quad \prod\{x\,2_5,\ f\,1_4\,3_4\,2_3,\ f\,3_4\,2_4\,0_3,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \quad \prod\{x\,2_5,\ f\,1_4\,3_4\,1_3,\ f\,3_4\,2_4\,0_3,\ y\,1_3\},\\
& \quad \prod\{x\,2_5,\ f\,1_4\,3_4\,0_3,\ y\,0_3\},\\
& \quad \prod\{x\,1_5,\ f\,2_4\,2_4\,2_3,\ z\,0_2,\ y\,1_3\},\\
& \quad \prod\{x\,1_5,\ f\,2_4\,3_4\,2_3,\ z\,0_2,\ f\,3_4\,2_4\,0_3,\ y\,1_3\},\\
& \quad \prod\{x\,1_5,\ f\,2_4\,3_4\,1_3,\ z\,1_2,\ y\,0_3\},\\
& \quad \prod\{x\,1_5,\ f\,2_4\,3_4\,2_3,\ z\,0_2,\ y\,0_3\},\\
& \quad \prod\{x\,1_5,\ z\,0_2,\ f\,2_4\,1_4\,2_3,\ y\,2_3\},\\
& \quad \prod\{x\,1_5,\ z\,1_2,\ f\,2_4\,2_4\,2_3,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \quad \prod\{x\,1_5,\ z\,1_2,\ f\,3_4\,1_4\,1_3,\ f\,2_4\,3_4\,2_3,\ y\,0_3\},\\
& \quad \prod\{x\,1_5,\ z\,1_2,\ f\,2_4\,2_4\,1_3,\ y\,1_3\},\\
& \quad \prod\{x\,1_5,\ f\,2_4\,3_4\,2_3,\ f\,3_4\,2_4\,0_3,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \quad \prod\{x\,1_5,\ f\,2_4\,3_4\,1_3,\ f\,3_4\,2_4\,0_3,\ y\,1_3\},\\
& \quad \prod\{x\,1_5,\ f\,2_4\,3_4\,0_3,\ y\,0_3\},\\
& \quad \prod\{x\,0_5,\ z\,1_2,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \quad \prod\{x\,0_5,\ f\,3_4\,2_4\,0_3,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \quad \prod\{x\,0_5,\ f\,3_4\,2_4\,0_3,\ y\,1_3\},\\
& \quad \prod\{x\,0_5,\ y\,0_3\},\\
& \quad \prod\{x\,0_5,\ y\,1_3,\ z\,1_2\},\\
& \quad \prod\{x\,0_5,\ y\,2_3,\ z\,0_2\}\}
\end{aligned}
$$

Fig. 4. The strictness property of function `foldr`, described in language Λ^0.

$$cat^{\#}\ xss^{\#} \quad = \quad foldr^{\#}\ append^{\#}\ xss^{\#}\ 3_4 \tag{5}$$

$$cat^{\#} = \lambda x_6 \,.\, \lambda y_3 \,.\quad \bigsqcup\{\quad \bigsqcap\{x\,4_5,\ y\,0_2\},\quad \bigsqcap\{x\,1_5,\ y\,1_2\},\quad x\,0_5\} \tag{6}$$

Fig. 5. Strictness analysis for function `cat`.

Note. Equation (5) describe the strictness property of `cat`, according to its program text. The strictness of `cat` is then calculated by first substituting the definition of $append^{\#}$ (in Figure 3), $foldr^{\#}$ (in Figure 4), and 3_4 (in Table 4) into Equation (5), then by normalizing the resulting term. The result is Equation (6), which is the strictness property of function `cat`.

giving an explicit proof strategy, the augmented, simply typed λ-calculus in our approach provides a simple way to compute the desired results.

Recent work has shown progress in the development of fast strictness analyzers for higher-order functions. See, for example, Ferguson and Hughes (1993), Hankin and Hunt (1992), Hankin and Le Métayer (1994), Nocker (1993) and Seward (1993). Ferguson and Hughes formulate abstract interpretations as sequential algorithms on concrete data structures (CDS). CDS are trees with labeled edges, representing the states of computation. Hankin and Hunt provide techniques to reduce the sizes of abstract domains such that least fixed points can be approximated quickly. Hankin and Le Métayer, in work related to Jensen's strictness logic (1991), develop a type-based system for abstract interpretation. They also propose a notation of lazy type to improve efficiency. Nocker performs strictness analysis by term reduction in abstract domains, mimicking the effect of term reduction in the concrete domains. Elements of his abstract domains are graphs, and the domains can be infinite. Seward defines an abstract lambda calculus for expressing recursive domain equations, and uses term-rewriting to derive solutions. Approximation techniques are used to derive strictness properties of higher-order functions.

With the exception of Hankin and Hunt (1992), each of the papers mentioned above, like ours, uses some kind of syntactic form to represent elements in an abstract domain. Each also uses the respective reduction machinery to calculate the abstract semantics of a program. Some of them, Nocker (1993) and Seward (1993) for example, must depend on approximation schemes to ensure the analyses will terminate. In this paper, in addition to proposing yet another framework of abstract interpretation based on syntactic reduction, we have put considerable effort in proving it to be sound and complete.

We would like to emphasize that our syntactic method is compositional, while many of the recent works are not. Take the definition of `cat` function in section 5 as an example. Our syntactic method analyzes the strictness of `foldr` and `append` from their definitions, and these results are composed to derive the strictness of

`cat` (just as `cat` is composed from `foldr` and `append`). Once derived, the syntactic forms for the strictness properties of `cat`, `foldr`, and `append` can be reused whenever the three functions are applied in other contexts. Many recent strictness analyzers, however, will calculate only the strictness of `cat`, and those parts of `foldr` and `append` which are related to `cat`. (Think of it as inlining `append` into `foldr` to make `cat`.) When `foldr` and `append` are used in other contexts, new analyses will be performed again. Our compositional approach fits better in a modular program development environment where the strictness property of a function, like the type of the function, can be put into its interface file and be consulted whenever it is needed. A non-compositional approach will need to export the *code* of the function in order to perform strictness analysis in places where the function is used. We also observe that several of the above recent strictness analyzers have difficulty analysing higher-order functions, like `foldr`, by their definitions. On contrary, our method is able to take `foldr` by itself and derives its complete strictness property.

Acknowledgments

The authors thank the referees for their critical comments and the editors for their enduring patience.

A Strong normalization

We will use a technique from Dershowitz and Manna (1979), based on multiset ordering, to show that every term in language Λ is strongly normalizable. A multiset is a bag, where an element may occur more than once. The equality $A = B$ for two multisets A and B means that any element occurring n times in A also occurs exactly n times in B, and *vice versa*. Also, $A \uplus B$ is the multiset containing $m + n$ occurrences of any element occurring m times in A and n times in B.

Let $\mathcal{M}(S)$ denote the set of all finite multisets with elements taken from the set S. If S is a partially ordered set with irreflexive ordering $\sqsubset_S$, then define an irreflexive ordering $\sqsubset_{\mathcal{M}(S)}$ for $\mathcal{M}(S)$ such that

$A \sqsubset_{\mathcal{M}(S)} B$ iff there exists multisets $X, Y \in \mathcal{M}(S)$, where $\emptyset \neq X \subseteq B$, such that $A = (B - X) \uplus Y$, and for all $y \in Y$ there exists a $x \in X$ such that $y \sqsubset_S x$.

It is shown by Dershowitz and Manna that $\mathcal{M}(S)$ is a partially ordered set with (irreflexive) ordering $\sqsubset_{\mathcal{M}(S)}$ if S is with $\sqsubset_S$. Also, $\sqsubset_{\mathcal{M}(S)}$ is well-founded iff $\sqsubset_S$ is. That is, if there is no infinite descending chain in S under ordering $\sqsubset_S$, then there is no infinite descending chain in $\mathcal{M}(S)$ under ordering $\sqsubset_{\mathcal{M}(S)}$ either.

For example, $\mathcal{M}(\boldsymbol{N})$, the set of all finite multisets of natural numbers, is partially ordered by $\sqsubset_{\mathcal{M}(N)}$, where $\sqsubset_N$ is defined as $<$. Furthermore, since there is no infinite decreasing sequence for a given natural number, we know that there is also no infinite descending sequence, using relation $\sqsubset_{\mathcal{M}(N)}$, for a given element in $\mathcal{M}(\boldsymbol{N})$.

Proposition A.1
Every term $M \in \Lambda$ is strongly normalizable.

Proof
It is well known that the simply typed λ-calculus is strongly normalizable (see, for example, Girard, Taylor & Lafont (1989)). Language Λ differs from simply typed λ-calculus in that it introduces new terms and new reduction rules for ground type $\mathit{2}$. It suffices to show that all the newly introduced terms are $\boldsymbol{\beta} \sqcup \sqcap \boldsymbol{d}$ strongly normalizable to complete the proof. There are four classes of new terms: $0, 1, M \sqcup N$, and $M \sqcap N$, where $M, N \in \Lambda_{\mathit{2}}$. It is clear that both 0 and 1 are strongly normalizable. It remains to show that both $\bigsqcup_{i\in I} M_i$ and $\bigsqcap_{i\in I} M_i$ are strongly normalizable if each M_i is.

Let $\nu(M)$ be the maximal number of steps needed to reduce a term M to a normal form. That is,

$$\begin{array}{lcll} \nu(M) & = & 0, & \text{if } M \text{ is a normal form} \\ \nu(M) & = & \max_{i\in I} \nu(N_i) \ \ +1, & \text{if } M \to N_i \text{ for } i \in I. \end{array}$$

It is clear that M is strongly normalizable iff $\nu(M)$ is finite, and whenever $M \to M'$, we have $\nu(M') < \nu(M)$. For a disjunctive term $\bigsqcup_{i\in I} M_i$, define multiset–valued function μ by

$$\mu(\textstyle\bigsqcup_{i\in I} M_i) = \{\nu(M_i) \mid i \in I\}.$$

It is clear that $\mu(\bigsqcup_{i\in I} M_i) \in \mathcal{M}(\boldsymbol{N})$ if each M_i is strongly normalizable. We show that, by a multiset induction on value of function μ, $\bigsqcup_{i\in I} M_i$ is indeed strongly normalizable if each M_i is.

There are two possible ways in which a term $\bigsqcup_{i\in I} M_i$ can be one-step reduced.

- $\bigsqcup_{i\in I} M_i \to_{\sqcup} \bigsqcup_{i\in I-J} M_j$,
  where $\emptyset \neq J \subset I$. This is a one-step $\sqcup$-reduction at the top level of the disjunction.
  Then, it is clear that $\mu(\bigsqcup_{i\in I-J} M_i) \sqsubset_{\mathcal{M}(N)} \mu(\bigsqcup_{i\in I} M_i)$.
- $\bigsqcup_{i\in I} M_i \to \bigsqcup_{i\in I} M'_i$,
  where there exists a $k \in I$ such that $M_k \to M'_k$, and for all $i \in I - \{k\}$, $M'_i \equiv M_i$. This is a one-step reduction in the sub–term M_k.
  It is clear that $\nu(M'_k) < \nu(M_k)$, and $\nu(M'_i) = \nu(M_i)$ for $i \in I - \{k\}$. If M'_k is not a disjunctive term, then it follows that $\mu(\bigsqcup_{i\in I} M'_i) \sqsubset_{\mathcal{M}(N)} \mu(\bigsqcup_{i\in I} M_i)$.
  If $M'_k \equiv \bigsqcup_{j\in J} N_j$ is a disjunctive term itself, then $\nu(N_j) \leq \nu(M'_k) < \nu(M_k)$ for each $j \in J$. Thus,

$$\mu(\bigsqcup_{i\in I} M'_i) \ = \ \{\nu(M_i) \mid i \in I - \{k\}\} \uplus \{\nu(N_j) \mid j \in J\} \ \sqsubset_{\mathcal{M}(N)} \ \mu(\bigsqcup_{i\in I} M_i).$$

In both cases, the value of $\mu(\bigsqcup_{i\in I} M_i)$ decreases whenever $\bigsqcup_{i\in I} M_i$ is one-step reduced. Since there is no infinite descending sequence in $\mathcal{M}(\boldsymbol{N})$, it follows that $\bigsqcup_{i\in I} M_i$ strongly normalizes if each M_i does.

For a conjunctive term $\bigsqcap_{i\in I} M_i$, its strong normalization proof is similar to the disjunction case above, but is slightly more complicated due to $\boldsymbol{d}$-reduction.† Let

† We can no longer use a multiset induction based on the function μ defined by

$$\mu(\textstyle\bigsqcap_{i\in I} M_i) = \{\nu(M_i) \mid i \in I\}.$$

us define $\xi(M)$ to be the maximal number of $\sqcup$ symbol in the terms reducible from M. That is,

$$\xi(M) = \max\{\text{number of } \sqcup \text{ symbol in } M' \mid M \to^* M'\}.$$

It is clear that if M is strongly normalizable then $\xi(M)$ is finite, and whenever $M \to M'$, we have $\xi(M') \leq \xi(M)$. Furthermore, for an index set I with $|I| > 1$, we have $\xi(M_i) < \xi(\bigsqcup_{i\in I} M_i)$ for all $i \in I$. We then define a multiset-valued function μ for conjunctive term $\bigsqcap_{i\in I} M_i$ by

$$\mu(\bigsqcap_{i\in I} M_i) = \{\nu(M_i) \mid i \in I\} \uplus \{\xi(M_i) \mid i \in I\}.$$

Similar to the two cases in disjunctive terms, it is clear that when a conjunctive term $\bigsqcap_{i\in I} M_i$ is one-step reduced either by a $\sqcap$-reduction at the top level or by a one-step reduction in sub-term M_i, the function value μ is decreased according to ordering $\sqsubset_{\mathcal{M}(N)}$. Now, for a one-step $\boldsymbol{d}$-reduction at the top level,

- $\bigsqcap_{i\in I} \bigsqcup_{j\in J_i} M_{i,j} \;\to_d\; \bigsqcup_{p\in\prod_{i\in I} J_i} \bigsqcap_{i\in I}\; M_{i,p|i}$,
  where $|I| > 1$, and for some $i \in I, |J_i| > 1$.
  Then, for all tuples $p \in \prod_{i\in I} J_i$ and all $i \in I$, $\xi(M_{i,p|i}) \leq \xi(\bigsqcup_{j\in J_i} M_{i,j})$ and $\nu(M_{i,p|i}) \leq \nu(\bigsqcup_{j\in J_i} M_{i,j})$. Furthermore, since there is an index set J_i with $|J_i| > 1$ for some $i \in I$, there is an index $i \in I$ such that $\xi(M_{i,p|i}) < \xi(\bigsqcup_{j\in J_i} M_{i,j})$.
  To summarize, we have for all tuple $p \in \prod_{i\in I} J_i$

  $$\mu(\bigsqcap_{i\in I} M_{i,p|i}) \quad \sqsubset_{\mathcal{M}(N)} \quad \mu(\bigsqcap_{i\in I} \bigsqcup_{j\in J_i} M_{i,j}).$$

  It follows that for all p, $\bigsqcap_{i\in I} M_{i,p|i}$ is strongly normalizable. By the result from the case for disjunctive terms, $\bigsqcup_{p\in\prod_{i\in I} J_i} \bigsqcap_{i\in I}\; M_{i,p|i}$ strongly normalizes.

We conclude that conjunctive terms are strongly normalizable.

This completes the proof that all Λ terms are strongly normalizable. $\square$

The induction will fail.
A conjunctive term like $\bigsqcap_{i\in I} \bigsqcup_{j\in J_i} M_{i,j}$ can be reduced to $\bigsqcup_{p\in\prod_{i\in I} J_i} \bigsqcap_{i\in I}\; M_{i,p|i}$ by a one–step d–reduction. It is possible that for some tuple $p \in \prod_{i\in I} J_i$, we have $\nu(M_{i,p|i}) = \nu(\bigsqcup_{i\in J_i} M_{i,j})$ for all $i \in I$. This occurs when both $M_{i,p|i}$ and $\bigsqcup_{i\in J_i} M_{i,j}$ are in normal form, for example. This means that

$$\mu(\bigsqcap_{i\in I} M_{i,p|i}) \not\sqsubset_{\mathcal{M}(N)} \mu(\bigsqcap_{i\in I} \bigsqcup_{i\in J_i} M_{i,j}),$$

and the induction fails.
As an example, consider the reduction $x \sqcap (a \sqcup b) \;\to_d\; (x \sqcap a) \sqcup (x \sqcap b)$, where x, a, b are variables. Because terms $x, (a \sqcap b), (x \sqcap a)$, and $(x \sqcap b)$ are all normal forms, we then would have

$$\mu(x \sqcap (a \sqcup b)) = \{0, 0\} \not\sqsubset_{\mathcal{M}(N)} \{0, 0\} = \mu(x \sqcap a),$$

and

$$\mu(x \sqcap (a \sqcup b)) = \{0, 0\} \not\sqsubset_{\mathcal{M}(N)} \{0, 0\} = \mu(x \sqcap b).$$

The induction fails.

B Church–Rosser

We first prove several technical lemmas before showing that $\boldsymbol{\beta} \sqcup \sqcap \boldsymbol{d}$ is Church–Rosser.

Lemma B.1
Let $M, N \in \Lambda$ and $M \preceq N$. If $M \to M'$, then there exists a term N' such that $N \to^* N'$ and $M' \preceq N'$. Conversely, if $N \to N'$, then there exists a term M' such that $M \to^* M'$ and $M' \preceq N'$.

Proof
We show that if $M \to M'$, then there exists a term N' such that $N \to^* N'$ and $M' \preceq N'$. The proof for its dual case is similar.

We use a case-by-case analysis. For example, if $M \to M'$ is via an $\boldsymbol{r}$-reduction in a sub-term of M, then we will show that, at most, an $\boldsymbol{r}$-reduction in the corresponding sub-term of N will reduce N to N' such that $M' \preceq N'$. We show the case for $\boldsymbol{d}$-reduction. The other three cases for $\boldsymbol{\beta}$, $\sqcup$, and $\sqcap$ reductions are similar.

Without loss of generality, we can assume that $M \equiv \bigsqcap_{i\in I} \bigsqcup_{j\in J_i} M_{i,j}$ and $M' \equiv \bigsqcup_{p\in\prod_{i\in I} J_i} \bigsqcap_{i\in I} M_{i,p|i}$. Also, N can be written as $\bigsqcap_{k\in K} \bigsqcup_{l\in L_k} N_{k,l}$ and reduced to term $\bigsqcup_{q\in\prod_{k\in K} L_k} \bigsqcap_{k\in K} N_{k,q|k}$ by at most one $\boldsymbol{d}$-reduction. Our goal is to show that

$$\bigsqcup_{p\in\prod_{i\in I} J_i} \bigsqcap_{i\in I} M_{i,p|i} \quad \preceq \quad \bigsqcup_{q\in\prod_{k\in K} L_k} \bigsqcap_{k\in K} N_{k,q|k} \tag{7}$$

if

$$\bigsqcap_{i\in I} \bigsqcup_{j\in J_i} M_{i,j} \quad \preceq \quad \bigsqcap_{k\in K} \bigsqcup_{l\in L_k} N_{k,l}. \tag{8}$$

By Lemma 2.8, relationship (8) implies

$$(\forall k \in K)(\exists i \in I)(\forall j \in J_i)(\exists l \in L_k)(M_{i,j} \preceq N_{k,l}).$$

We then can assume that there exists a total function $f : K \to I$ and, for each $i \in I$ and each $k \in K$, a total function $g_{i,k} : J_i \to L_k$ such that

$$(\forall k \in K)(\forall j \in J_{f(k)})(M_{f(k),j} \preceq N_{k,g_{f(k),k}(j)}). \tag{9}$$

From f, we define its inverse image function $f^{-1} : I \to 2^K$ such that

$$f^{-1}(i) = \{k \in K \mid f(k) = i\}.$$

Note that $\bigcup_{i\in I} f^{-1}(i) = K$ because f is total.

By predicate (9) and Lemma 2.8, it follows that for all $i \in I$ and for all $j \in J_i$,

$$M_{i,j} \quad \preceq \quad \bigsqcap_{k\in f^{-1}(i)} N_{k,g_{i,k}(j)}$$

because

$$M_{i,j} \quad \preceq \quad N_{k,g_{i,k}(j)}$$

for all $k \in K$ with $f(k) = i$. Then for all tuples $p \in \prod_{i\in I} J_i$,

$$\bigsqcap_{i\in I} M_{i,p|i} \quad \preceq \quad \bigsqcap_{i\in I} \bigsqcap_{k\in f^{-1}(i)} N_{k,g_{i,k}(p|i)}$$

because

$$M_{i,p|i} \quad \preceq \quad \bigsqcap_{k\in f^{-1}(i)} N_{k,g_{i,k}(p|i)}$$

for all $i \in I$. Furthermore,

$$\bigsqcap_{i\in I}\bigsqcap_{k\in f^{-1}(i)} N_{k,g_{i,k}(p|i)} \quad \preceq \quad \bigsqcap_{k\in K} N_{k,g_{f(k),k}(p|f(k))}$$

because f is total. It follows that

$$\bigsqcap_{i\in I} M_{i,p|i} \quad \preceq \quad \bigsqcap_{k\in K} N_{k,g_{f(k),k}(p|f(k))} \tag{10}$$

for all tuples $p \in \prod_{i\in I} J_i$.

Now, define a function m from the Cartesian product $\prod_{i\in I} J_i$ to the Cartesian product $\prod_{k\in K} L_k$ such that for all tuples $p \in \prod_{i\in I} J_i$ and indices $k \in K$

$$m(p)|k = g_{f(k),k}(p|f(k)).$$

Function m is well defined because f, g, and $|$ are all well defined. From (10) and m, we have for all $p \in \prod_{i\in I} J_i$,

$$\bigsqcap_{i\in I} M_{i,p|i} \quad \preceq \quad \bigsqcap_{k\in K} N_{k,m(p)|k},$$

which implies

$$(\forall p \in \textstyle\prod_{i\in I} J_i)(\exists q \in \prod_{k\in K} L_k)(\bigsqcap_{i\in I} M_{i,p|i} \quad \preceq \quad \bigsqcap_{k\in K} N_{k,q|k}).$$

By Lemma 2.8, the above predicate proves relationship (7).

This completes the proof. □

Lemma B.2
Let $M, N \in \Lambda$. If $M \simeq N$, then there exists a term $L \in \Lambda$ such that both $M \to^* L$ and $N \to^* L$.

Proof
The proof is by an induction on the structures of terms M and N. The induction bases occur when both M and N are either $0, 1$, or an identical variable. Then, $M \simeq N$ implies $M \equiv N$, which establishes the induction bases.

By Lemma 2.8, it suffices to show that, if $\bigsqcup_{i\in I} M_i \simeq \bigsqcup_{j\in J} N_j$ then they can be reduced to a common term, and if $\bigsqcap_{i\in I} M_i \simeq \bigsqcap_{j\in J} N_j$ then they can also be reduced to a common term, where I, J are index sets and $M_i, N_j \in \Lambda_2$. We show the case for disjunction. The case for conjunction is similar.

The term $\bigsqcup_{i\in I} M_i$ can be reduced to $\bigsqcup_{i\in I'} M_i$ by a $\sqcup$-reduction such that for all $p, q \in I'$ neither $M_p \preceq M_q$ nor $M_q \preceq M_p$. That is, we keep only the maximal terms in the disjunction. By a similar reduction, $\bigsqcup_{j\in J} N_j$ reduces to $\bigsqcup_{j\in J'} N_j$. It is clear that $\bigsqcup_{i\in I'} M_i \simeq \bigsqcup_{j\in J'} N_j$. Furthermore, for each $i \in I'$ there is only one $j \in J'$ such that $M_i \simeq N_j$, and *vice verse*. By the induction hypothesis, M_i and N_j reduce to a common term. It then follows that $\bigsqcup_{i\in I'} M_i$ and $\bigsqcup_{j\in J'} N_j$ can be reduced to a common term. □

A reduction relation $\boldsymbol{r}$ is locally confluent iff for terms L, M, M', whenever $L \to_r M$ and $L \to_r M'$, there exists a term N such that $M \to_r^* N$ and $M' \to_r^* N$.

Lemma B.3
$\boldsymbol{\beta} \sqcap \sqcup \boldsymbol{d}$ is locally confluent.

Proof

Suppose that a term L is one-step reduced to terms M and N respectively by a reduction of one of two non-overlapping redexes in term L, then M and N can be reduced to a common term by one more reduction at each of their other redexes. The reductions are locally confluent in these cases. What remains to be shown are the cases when the redexes overlap.

We show local confluence for reductions on overlapping redexes by a case-by-case analysis. We show two cases, where a $\sqcup$-redex contains other redexes, and where a $\boldsymbol{d}$-redex contains other redexes. The cases for $\sqcap$ and $\boldsymbol{\beta}$ are not shown here because the former is similar to the case for $\sqcup$, and the latter is straightforward.

A $\sqcup$-redex $\bigsqcup_{i\in I} M_i$ can be one-step reduced

- to term $\bigsqcup_{i\in I-J} M_i$ by a $\sqcup$-reduction, or
- to another term $\bigsqcup_{i\in I-J'} M_i$, where $J \neq J'$, by a $\sqcup$-reduction, or
- to term $\bigsqcup_{i\in I} M'_i$ with a one-step reduction in sub-term $M_k \to M'_k, k \in I$; otherwise $M_i \equiv M'_i$ for $i \neq k$.

The first two cases are easy because $\bigsqcup_{i\in I-J} M_i \simeq \bigsqcup_{i\in I-J'} M_i$; hence, by Lemma B.2, the two terms can be reduced to the same term. It suffices to show that $\bigsqcup_{i\in I-J} M_i$ and $\bigsqcup_{i\in I} M'_i$ reduce to a common term to complete the case for the $\sqcup$-redex. There are two sub-cases, depending whether k is in the set $I - J$ or not, when $M_k \to M'_k$ in term $\bigsqcup_{i\in I} M'_i$. Suppose that $k \in I - J$. Then for all $j \in J$ such that $M_j \preceq M_k$, let $M_j \to^* M''_j$ and $M''_j \preceq M'_k$ (Lemma B.1). Otherwise, let $M''_i \equiv M'_i$ for all other $i \in I$. It follows that both $\bigsqcup_{i\in I-J} M_i \to^* \bigsqcup_{i\in I-J} M''_i$ and $\bigsqcup_{i\in I} M'_i \to^* \bigsqcup_{i\in I-J} M''_i$. On the other hand, suppose $k \in J$. Then for $i \in I - J$ such that $M_k \preceq M_i$, let $M_i \to^* M''_i$ and $M'_k \preceq M''_i$ (Lemma B.1). Furthermore, for those $j \in J$ and $M_j \preceq M_i$, also let $M_j \to^* M''_j$ and $M''_j \preceq M''_i$. Otherwise, let $M''_i \equiv M'_i$ for all other $i \in I$. It follows that both $\bigsqcup_{i\in I-J} M_i$ and $\bigsqcup_{i\in I} M'_i$ reduce to $\bigsqcup_{i\in I-J} M''_i$.

The case for $\boldsymbol{d}$–redex is slightly more complicated. A redex $\bigsqcap_{i\in I} \bigsqcup_{j\in J_i} M_{i,j}$ can be one-step reduced

- to term $\bigsqcap_{i\in I'} \bigsqcup_{j\in J_i} M_{i,j}$, where $I' \subset I$, by a $\sqcap$-reduction, or
- to term $\bigsqcap_{i\in I} \bigsqcup_{j\in J'_i} M_{i,j}$, where $J'_k \subset J_k$ for some $k \in I$, and $J'_i = J_i$ for $i \neq k$, by a $\sqcup$–reduction, or
- to term $\bigsqcup_{p\in\prod_{i\in I} J_i} \bigsqcap_{i\in I} M_{i,p|i}$ by a $\boldsymbol{d}$-reduction, or
- to term $\bigsqcap_{i\in I} \bigsqcup_{j\in J_i} M'_{i,j}$ with a one-step reduction in sub–term $M_{k,l} \to M'_{k,l}$, where $k \in I$ and $l \in J_k$; otherwise, $M_{i,j} \equiv M'_{i,j}$ for $i \neq k$ or $j \neq l$.

It turns out that both $\bigsqcap_{i\in I'} \bigsqcup_{j\in J_i} M_{i,j}$ and $\bigsqcap_{i\in I} \bigsqcup_{j\in J'_i} M_{i,j}$ can be reduced to term $\bigsqcap_{i\in I'} \bigsqcup_{j\in J'_i} M_{i,j}$. Furthermore,

$$\bigsqcap_{i\in I'} \bigsqcup_{j\in J_i} M_{i,j} \;\to\; \bigsqcup_{p\in\prod_{i\in I'} J_i} \bigsqcap_{i\in I} M_{i,p|i} \;\simeq\; \bigsqcup_{p\in\prod_{i\in I} J_i} \bigsqcap_{i\in I} M_{i,p|i}$$

and

$$\bigsqcap_{i\in I} \bigsqcup_{j\in J'_i} M_{i,j} \;\to\; \bigsqcup_{p\in\prod_{i\in I} J'_i} \bigsqcap_{i\in I} M_{i,p|i} \;\simeq\; \bigsqcup_{p\in\prod_{i\in I} J_i} \bigsqcap_{i\in I} M_{i,p|i}.$$

This shows the confluence of the first three cases in the above list. It remains to show that that last case is confluent with the first three. But the confluence

of $\bigsqcap_{i\in I}\bigsqcup_{j\in J_i} M'_{i,j}$ with $\bigsqcap_{i\in I'}\bigsqcup_{j\in J_i} M_{i,j}$ and $\bigsqcap_{i\in I}\bigsqcup_{j\in J'_i} M_{i,j}$ are covered in the cases for $\sqcap$ and $\sqcup$, respectively. Finally, it can be shown both $\bigsqcup_{p\in\prod_{i\in I} J_i}\bigsqcap_{i\in I} M_{i,p|i}$ and $\bigsqcap_{i\in I}\bigsqcup_{j\in J_i} M'_{i,j}$ can be reduced to $\bigsqcup_{p\in\prod_{i\in I} J_i}\bigsqcap_{i\in I} M'_{i,p|i}$. This completes the proof. □

We now prove the main result.

Proposition B.4
$\boldsymbol{\beta}\sqcup\sqcap\boldsymbol{d}$ is Church–Rosser.

Proof
Since $\boldsymbol{\beta}\sqcup\sqcap\boldsymbol{d}$ is strongly normalizing (Proposition 2.10) and locally confluent (Lemma B.3), it follows that it is confluent, which implies Church–Rosser. □

References

Abramsky, S. (1991) Domain theory in logical form. *Ann. Pure and Applied Logic*, **51**(1–2): 1–77, March.

Barendregt, H. P. (1984) *The Lambda Calculus: Its Syntax and Semantics: Studies in Logic and the Foundations of Mathematics 103*. North–Holland, revised edition.

Burn, G. L., Hankin, C. L. and Abramsky, S. (1986) Strictness analysis for higher-order functions. *Science of Computer Programming*, **7**(3): 249–278.

Chuang, T.-R. (1993) New Techniques for the Analysis and Implementation of Functional Programs. *PhD thesis*, Department of Computer Science, New York University, August. (Also available as Technical Report 644.)

Clack, C. and Peyton Jones, S. L. (1985) Strictness analysis – a practical approach. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pp. 35–49. Nancy, France, September. (*Lecture Notes in Computer Science 201*. Springer-Verlag.)

Dershowitz, N. and Manna, Z. (1979) Proving termination with multiset orderings. *Comm. ACM*, **22**(8): 465–476, August.

Ferguson, A. and Hughes, J. (1993) Fast abstract interpretation using sequential algorithms. In P. Cousot, M. Falaschi, G. Filè and A. Rauzy, editors, *Static Analysis – The Third International Workshop*, pp. 45–59. Padova, Italy, September. (*Lecture Notes in Computer Science 724*. Springer-Verlag.)

Girard, J.-Y., Taylor, P. and Lafont, Y. (1989) *Proofs and Types: Cambridge Tracts in Theorectical Computer Science 7*. Cambridge University Press.

Hankin, C. and Hunt, S. (1992) Approximate fixed points in abstract interpretation. In B. Krieg–Brückner, editor, *4th European Symposium on Programming*, pp. 219–232. Rennes, France, February. (*Lecture Notes in Computer Science 582*, Springer-Verlag.)

Hankin, H. and Le Métayer, D. (1994) Deriving algorithms from type inference systems: Application to strictness analysis. In *Proceedings 21st ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pp. 202–212. Portland, Oregon, January. ACM Press.

Hunt, S. (1989) Frontiers and open sets in abstract interpretation. In *Functional Programming Languages and Computer Architecture*, pp. 1–11. Imperial College, London, UK, September. ACM/Addison-Wesley.

Hunt, S. and Hankin, C. (1991) Fixed points and frontiers: a new perspective. *J. Functional Programming*, **1**(1): 91–120, January.

Jensen, T. P. (1990) Abstract interpretation *vs.* type inference: A topological perspective. In S. L. Peyton Jones, G. Hutton and C. Kehler Holst, editors, *Functional Programming, Glasgow 1990: Proceedings of the 1990 Glasgow Workshop*, pp. 141–145. Ullapool, Scotland, August. Springer-Verlag.

Jensen, T. P. (1991) Strictness analysis in logical form. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, pp. 352–366. Cambridge, MA, USA, August. (*Lecture Notes in Computer Science 523*. Springer-Verlag.)

Martin, C. and Hankin, C. (1987) Finding fixed points in finite lattices. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pp. 426–445. Portland, Oregon, USA, September. (*Lecture Notes in Computer Science 274*. Springer-Verlag.)

Martin, C. C. (1989) Algorithms for Finding Fixpoints in Abstract Interpretation. *PhD thesis*, Department of Computing, Imperial College of Science, Technology and Medicine, University of London.

Nöcker, E. (1993) Strictness analysis using abstract reduction. *Conference on Functional Programming Languages and Computer Architecture*, pp. 255–265. University of Copenhagen, Denmark, June. ACM Press.

Peyton Jones, S. L. and Clack, C. (1987) Finding fixpoints in abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pp. 246–265. Ellis Horwood.

Plotkin, G. D. (1977) LCF considered as a programming language. *Theoretical Computer Science*, **5**: 223–255.

Seward, J. (1993) Solving recursive domain equations by term rewriting. In J. T. O'Donnell and K. Hammond, editors, *Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop*, pp. 265–279. Ayr, Scotland, July. Springer-Verlag.

Wadler, P. (1987) Strictness analysis on non–flat domains (by abstract interpretation over finite domains). In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pp. 246–265. Ellis Horwood.

Wang, P. P. (1989) A special case of second-order strictness analysis. *Technical Memo MIT/LCS/TM–383*, Massachusetts Institute of Technology, February.

Table 4. *The definitions in language* Λ^0 *of all the semantic elements in the finite domains* $2, 3, 4, 5,$ *and* 6

Element	Term				
	in Λ^0_2	in Λ^0_3	in Λ^0_4	in Λ^0_5	in Λ^0_6
0	0_2	$\lambda x_2 . 0_2$	$\lambda x_3 . 0_2$	$\lambda x_4 . 0_2$	$\lambda x_5 . 0_2$
1	1_2	$\lambda x_2 . x$	$\lambda x_3 . x\, 0_2$	$\lambda x_4 . x\, 0_3$	$\lambda x_5 . x\, 0_4$
2	n/a	$\lambda x_2 . 1_2$	$\lambda x_3 . x\, 1_2$	$\lambda x_4 . x\, 1_3$	$\lambda x_5 . x\, 1_4$
3	n/a	n/a	$\lambda x_3 . 1_2$	$\lambda x_4 . x\, 2_3$	$\lambda x_5 . x\, 2_4$
4	n/a	n/a	n/a	$\lambda x_4 . 1_2$	$\lambda x_5 . x\, 3_4$
5	n/a	n/a	n/a	n/a	$\lambda x_5 . 1_2$

(a)

Element	Term
$hd^{\#}_{4\to 2}$	$\lambda x_4 . x\, 2_3$
$tl^{\#}_{4\to 4}$	$\lambda x_4 . \lambda y_3 . \bigsqcup \{ \bigsqcap \{x\, 2_3,\ y\, 0_2\},\ x\, 1_3 \}$
$cons^{\#}_{2\to 4\to 4}$	$\lambda x_2 . \lambda y_4 . \lambda z_3 . \bigsqcup \{ \bigsqcap \{x,\ y\, 0_3\},\ \bigsqcap \{y\, 1_3,\ z\, 1_2\},\ z\, 0_2 \}$
$hd^{\#}_{6\to 4}$	$\lambda x_6 . \lambda y_3 . x\, 4_5$
$tl^{\#}_{6\to 6}$	$\lambda x_6 . \lambda y_5 . \bigsqcup \{ \bigsqcap \{x\, 4_5,\ y\, 0_4\},\ x\, 3_5 \}$
$cons^{\#}_{4\to 6\to 6}$	$\lambda x_4 . \lambda y_6 . \lambda z_5 . \bigsqcup \{ \bigsqcap \{x\, 0_3,\ y\, 0_5\}, \bigsqcap \{x\, 1_3,\ y\, 1_5,\ z\, 3_4\}, \bigsqcap \{x\, 2_3,\ y\, 2_5,\ z\, 2_4\}, \bigsqcap \{y\, 3_5,\ z\, 1_4\}, z\, 0_4 \}$

(b)

Note. Part (a) shows the definitions of the elements in domains $2, 3, 4, 5$, and 6 as terms in language Λ^0. Note that the languages and the terms are defined inductively by using the following abbreviations: $2 \equiv 2$, $3 \equiv 2 \to 2$, $4 \equiv 3 \to 2$, $5 \equiv 4 \to 2$, and $6 \equiv 5 \to 2$. All the constants in the languages are subscripted by their types to avoid possible confusion. If written in full, for example, the definition $3_6 \equiv \lambda x_5 . x\, 2_4 \equiv \lambda x_{((2\to 2)\to 2)\to 2} . x\, (\lambda y_{2\to 2} . y\, 1_2)$.
The strictness properties of functions `hd`, `tl`, and `cons` are described in (b). Note that a primitive function has different strictness properties over different abstract domains. Here 4 is the abstract domain for lists, and 6 is the abstract domain for lists of lists.

Table 5. *The results of applying* $\mathrm{cat}^{\#}$ *to all the elements in the abstract domain* 6

$xxs^{\#}$	0_6	1_6	2_6	3_6	4_6	5_6
$cat^{\#}\ xxs^{\#}$	0_4	1_4	1_4	1_4	2_4	3_4

Note. The results are obtained by calculating the normal forms of $cat^{\#}\ xss^{\#}$, where the definition of $cat^{\#}$ is from Equation (6) in Figure 5, and the definition of $0_6, 1_6, \ldots, 5_6$ is from Table 4.
The results in the above table are interpreted in the following way. If the argument to the `cat` function is an undefined list ($xss^{\#} = 0_6$), then the result is an undefined list. If the argument is (at most) a list with an undefined/infinite tail, or is (at most) a finite list with one of its element either being an undefined list or being a list with an undefined/infinite tail ($xss^{\#} = 1_6, 2_6$, or 3_6), then the result is (at most) a list with an undefined/infinite tail. If the argument is (at most) a finite list in which all elements are finite lists but one of them contains an undefined element ($xss^{\#} = 4_6$), then the result is (at most) a finite list with one of its elements undefined. If the argument is (at most) a finite list in which all elements are finite lists with all elements defined ($xss^{\#} = 5_6$), then the result is (at most) a finite list with all its elements defined.