A Syntactic Approach to Fixed Point Computation on Finite Domains

Tyng–Ruey Chuang and Benjamin Goldberg
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University*

Abstract

We propose a syntactic approach to performing fixed point computation on finite domains. Finding fixed points in finite domains for monotonic functions is an essential task when calculating abstract semantics of functional programs. Previous methods for fixed point finding have been mainly based on semantic approaches which may be very inefficient even for simple programs.

We outline the development of a syntactic approach, and show that the syntactic approach is sound and complete with respect to semantics. A few examples are provided to illustrate this syntactic approach.

1 Motivation and Introduction

Finding fixed points for 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, 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 fixed points on finite domains. Most notable is the frontier method developed by Clack & Peyton Jones [3,10], Martin & Hankin [9], and Hunt & Hankin [5,6]. Young [12] also discusses related issues.

Let us briefly describe how the frontier representation of a function works. Take a function's strictness property as an example. A function's strictness property is usually described by a monotonic function f from a finite argument domain D to the two element domain $\mathcal{2}$, with $0 \sqsubseteq_{\mathcal{2}} 1$. A maximal–0–frontier representation of a 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.

When f is recursively defined, f is evaluated as the least fixed point of a functional F from domain $D \to \mathcal{2}$ to $D \to \mathcal{2}$, where $D \to \mathcal{2}$ is the monotonic function space from D to $\mathcal{2}$. The least fixed point of F is approximated by a sequence starting from the least element in domain $D \to \mathcal{2}$, $\bot_{D\to\mathcal{2}}$, 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. When the frontier does not change between successive iterations, then the fixed point has been found.

The frontier method is attractive in several ways. The representation is economical in space, hence makes easy the equality testing between two successive functions in the approximation sequence. 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 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. 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 building

*The authors' address: 251 Mercer Street, New York, NY 10012, U.S.A. E–mail: chuang@cs.nyu.edu, goldberg@cs.nyu.edu. This research has been supported, in part, by the National Science Foundation (#CCR–8909634) and DARPA (DARPA/ONR #N00014–91–J1472).

1992 ACM LISP & F.P.-6/92/CA
© 1992 ACM 0-89791-483-X/92/0006/0109...$1.50

mechanism, unless, of course, when functions are fully applied to their arguments.

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, which is defined as the least fixed point of the following functional F,

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

where $x_0, x_1, \ldots, x_{10} \in \mathcal{2}$, and $\sqcup$ is the (infix) least upper bound function. By symbolic evaluation, we have

$$f = \lambda x_0 . \lambda x_1 . \ldots . \lambda x_{10} . x_0$$

as the least fixed point. The process begins with the weakest approximation $f = \lambda x_0 . \lambda x_1 . \ldots . \lambda x_{10} . 0$, and takes only two iterations. By the above result, we also know that (the curried 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 from the very bottom of the domain $\mathcal{2}^{11} \to \mathcal{2}$. It is not difficult to see that the frontier method will have to make, step by step, 2^{10} approximations to reach this maximal–0–frontier, which is right in the middle of every 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 $\{\}$) in domain $\mathcal{2}^{11} \to \mathcal{2}$. In such "badly behaved" cases, the frontier method is very inefficient compared to the symbolic evaluation method. This phenomenon has been observed by Clack & Peyton Jones [3,10]. Hunt & Hankin [5,6] further suggest that higher–order functional programs are often badly behaved.

In this paper, we will develop a syntactic method suitable for symbolic calculation of fixed points on finite domains. This method uses a simply–typed λ–calculus augmented with four predefined constants — $0, 1, \sqcap$, and $\sqcup$ — and their associated reduction rules. At first thought, one might doubt whether such a task may be accomplished [3]. There are several reasons for the difficulty. First, there is no fixed point term in the simply–typed λ–calculus! That is, there is no such term $Y \in \Lambda_{(\sigma\to\sigma)\to\sigma}$, where σ is a type and $\Lambda_{(\sigma\to\sigma)\to\sigma}$ is the set of lambda terms whose type is $(\sigma \to \sigma) \to \sigma$, such that for any term $F \in \Lambda_{\sigma\to\sigma}$, YF is β–convertible to $F(YF)$. Also, it has been shown that there exists no "parallel or" term $\bigsqcup \in \Lambda_{\mathcal{2}\to(\mathcal{2}\to\mathcal{2})}$ such that $(\bigsqcup\ 1\ M)$ β–reduces to 1, $(\bigsqcup\ M\ 1)$ β–reduces to 1, and $(\bigsqcup\ 0\ 0)$ β–reduces to 0 for all term $M \in \Lambda_{\mathcal{2}}$ (see Girard, Taylor & Lafont [4], for example). However, this "parallel or" operation is frequently used in abstract interpretation.

However, these problems can be solved if the types in the language Λ only denote finite domains with suitable algebraic structures, and, in addition to the β rule, algebraic reduction rules for those finite domains are allowed in the calculus. We will show that, in a slightly restricted sense, there is a fixed point term $Y \in \Lambda_{(\sigma\to\sigma)\to\sigma}$ such that it can be shown, in a syntactic way, that YF is semantically equivalent to $F(YF)$ for any term F in a sub–language of $\Lambda_{\sigma\to\sigma}$. In fact, this syntactic method can be shown to be both sound and complete with respect to semantic methods, where the frontier method is one of them.

What are the advantages of using a syntactic method over a semantic method for computing fixed points on finite domains? One advantage is that the syntactic method may be more efficient than the semantic one, as illustrated by the above example. Another advantage is that we have a more uniform way to calculate the (operational) semantics of a functional program, whether it is the standard semantics or an abstract semantics, since these calculations differ only in how the reductions are performed. This is in contrast to the frontier method of computing a program's abstract semantics, in which the method used is totally different from the syntactic calculus (such as the λ–calculus) used to compute a program's standard semantics.

This paper will mainly address fixed point finding in the monotonic function spaces generated by the basic domain $\mathcal{2}$. We will later show how to relax this restriction and make the method applicable to basic domains other than $\mathcal{2}$.

2 The Language Λ and Its Properties

In this section we define a typed language Λ and the associated reduction rules. Most of the definition is standard or intuitive.

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

- $\mathcal{2} \in \Gamma$, 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 $N \in \Lambda_\sigma$,
- $(\lambda x . M) \in \Lambda_{\sigma\to\tau}$ if $x \in \Lambda_\sigma$ and $M \in \Lambda_\tau$,
- $0, 1 \in \Lambda_{\mathcal{2}}$, and
- $(M \sqcap N), (M \sqcup N) \in \Lambda_{\mathcal{2}}$ if $M, N \in \Lambda_{\mathcal{2}}$.

□

Type *2* is the *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 $\sqcap, \sqcup \in \Lambda_{2\to 2\to 2}$. The type constructor $\to$ is right associative. That is, $\sigma_1 \to \sigma_2 \to \ldots \to \sigma_n$ is a shorthand for $(\sigma_1 \to (\sigma_2 \to (\ldots \to \sigma_n)))$. 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 also take the liberty to omit some parentheses 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.

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

Definition 2.3

- Type *2* denotes the domain $D_2 = \{0, 1\}$, with the ordering $0 \sqsubseteq_2 1$, 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 $x \in D_\sigma$.

□

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 variable names to $\bigcup_{\sigma\in\Gamma} D_\sigma$. 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
- $[\![(M \sqcap N)]\!]\rho = ([\![M]\!]\rho) \sqcap ([\![N]\!]\rho)$,
  $[\![(M \sqcup N)]\!]\rho = ([\![M]\!]\rho) \sqcup ([\![N]\!]\rho)$.

□

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. Also, 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. 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 a binary relation $\preceq$ (pronounced "syntactically weaker") 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 are valid for all $F, G, H \in \Lambda$ and all $L, M, N \in \Lambda_2$.

- (application)

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

- (abstraction)

$$\frac{F\ \mathcal{R}\ G}{(\lambda\, x\, .\, F)\ \mathcal{R}\ (\lambda\, x\, .\, G)}$$

- (⊓)

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

- (⊔)

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

□

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

- $0 \preceq M, \quad M \preceq 1$,
- $(M \sqcap N) \preceq M, \quad M \preceq (M \sqcup N)$,
- $M \preceq (M \sqcap M), \quad (M \sqcup M) \preceq M$,
- $(M \sqcap N) \preceq (N \sqcap M), \quad (M \sqcup N) \preceq (N \sqcup M)$,
- $(L \sqcap (M \sqcap N)) \preceq ((L \sqcap M) \sqcap N)$,
  $((L \sqcap M) \sqcap N) \preceq (L \sqcap (M \sqcap N))$,
  $(L \sqcup (M \sqcup N)) \preceq ((L \sqcup M) \sqcup N)$,
  $((L \sqcup M) \sqcup N) \preceq (L \sqcup (M \sqcup N))$.

□

By using the definition of compatibility in defining the relation $\preceq$, we see that $\preceq$ is 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. Note that it is easy to check whether two Λ terms satisfy the $\preceq$ relationship or not. Based on $\preceq$, we define the following reduction rules for Λ.

Definition 2.7 The reduction relations $\boldsymbol{\beta}$, $\sqcap$, $\sqcup$, and $\boldsymbol{d}$ on language Λ are defined as follows.

- $\boldsymbol{\beta} = \{((\lambda x.M)N,\ M[x := N]) \mid M, N \in \Lambda\}$,
- $\sqcap = \{(M \sqcap N,\ M) \mid M, N \in \Lambda_2, M \preceq N\} \cup \{(M \sqcap N,\ N) \mid M, N \in \Lambda_2, N \preceq M\}$,
- $\sqcup = \{(M \sqcup N,\ M) \mid M, N \in \Lambda_2, N \preceq M\} \cup \{(M \sqcup N,\ N) \mid M, N \in \Lambda_2, M \preceq N\}$, and
- $\boldsymbol{d} = \{(L \sqcap (M \sqcup N),\ (L \sqcap M) \sqcup (L \sqcap N)) \mid L, M, N \in \Lambda_2\} \cup \{((M \sqcup N) \sqcap L,\ (M \sqcap L) \sqcup (N \sqcap L)) \mid L, M, N \in \Lambda_2\}$.

□

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 $\rightarrow_{\beta \sqcap \sqcup d}$ and $\rightarrow^*_{\beta \sqcap \sqcup d}$. The standard definitions of *$\boldsymbol{r}$-redex* and *$\boldsymbol{r}$-normal form* are used. (See Barendregt [2] for more details.) We will use *normal form* as an abbreviation for $\boldsymbol{\beta} \sqcap \sqcup \boldsymbol{d}$–normal form. Also, that a term $M \in \Lambda$ is strongly normalizable means that M is $\boldsymbol{\beta} \sqcap \sqcup \boldsymbol{d}$–strongly normalizable; *i.e.*, there is no infinite $\boldsymbol{\beta} \sqcap \sqcup \boldsymbol{d}$–reduction sequence starting with M.

Proposition 2.8 Every term $M \in \Lambda$ is strongly normalizable. □

PROOF OUTLINE. That simply–typed λ–calculus is strongly normalizable is well known, and, for example, is well presented in [4]. We show here how to use this result and to extend it to give a strong normalization proof for Λ.

We notice that the language Λ does not introduce new types other than those already expressible in the simply–typed λ–calculus. But it does introduce new terms and new reduction rules for terms of type *2*. It suffices to show that all the newly introduced terms are strongly normalizable to complete the proof. There are four classes of new terms: $0, 1, (M \sqcap N)$, and $(M \sqcup N)$, where $M, N \in \Lambda_2$. It is clear that both 0 and 1 are strongly normalizable. It remains to show that $(M \sqcap N)$ and $(M \sqcup N)$ are strongly normalizable if both M and N are strongly normalizable.

We define $\nu(M)$ to be the bound of the number of steps needed to reduce M to a normal form. If M is strongly normalizable, then $\nu(M)$ is finite. Also define $l(M)$ to be the length of the term M. There are four possible ways how a term $(M \sqcup N)$ can be one–step–reduced: it can either be reduced to M if $N \preceq M$, to N if $M \preceq N$, to $(M' \sqcup N)$ if $M \rightarrow M'$, or to $(M \sqcup N')$ if $N \rightarrow N'$. Using induction on $\nu(M)+\nu(N)$, it can be shown that $(M \sqcup N)$ is strongly normalizable if both M and N are.

The case for $(M \sqcap N)$ is slightly more complicated. There are six cases in which $(M \sqcap N)$ can be one–step–reduced: it can either be reduced to M if $M \preceq N$, to N if $N \preceq M$, to $(M' \sqcap N)$ if $M \rightarrow M'$, to $(M \sqcap N')$ if $N \rightarrow N'$, to $((M \sqcap N') \sqcup (M \sqcap N''))$ if $N \equiv (N' \sqcup N'')$, or to $((M' \sqcap N) \sqcup (M'' \sqcap N))$ if $M \equiv (M' \sqcup M'')$. Then using induction on $\nu(M) + \nu(N)$, and, for the induction base and each induction hypothesis, using a second induction on $l((M \sqcap N))$, we show that $(M \sqcap N)$ is strongly normalizable. This double induction is necessary because in the latest two cases, we may have $\nu(M) + \nu(N') = \nu(M) + \nu(N)$ (or $\nu(M)+\nu(N'') = \nu(M)+\nu(N)$, or $\nu(M')+\nu(N) = \nu(M)+\nu(N)$, or $\nu(M'') + \nu(N) = \nu(M) + \nu(N)$). But we have $l((M \sqcap N')) < l((M \sqcap N))$ and the second induction on on $l((M \sqcap N))$ establishes the result. ◇

Note that $\rightarrow^*$ is not Church–Rosser. For example, by a one–step $\sqcap$–reduction, we have both $\lambda x.\lambda y.(x \sqcap y)$ and $\lambda x.\lambda y.(y \sqcap x)$ as normal forms of $\lambda x.\lambda y.((x \sqcap y) \sqcap (y \sqcap x))$, because both $(x \sqcap y) \preceq (y \sqcap x)$ and $(y \sqcap x) \preceq (x \sqcap y)$. But $\lambda x.\lambda y.(x \sqcap y)$ and $\lambda x.\lambda y.(y \sqcap x)$ cannot be reduced further to a common term. We could introduce reduction rules aiming for commutativity and associativity to make the calculus Church–Rosser (see Appendix A), but then we lose strong normalizability. However, we will see that not all is lost. The following proposition shows that the normal forms of a Λ term might be related to one another by the $\preceq$ relation. Let us write $M \simeq N$ if $M, N \in \Lambda, M \preceq N$, and $N \preceq M$.

Proposition 2.9 Let $M \in \Lambda$. If both M' and M'' are normal forms of M, then $M' \simeq M''$. □

PROOF OUTLINE. See Appendix A. ◇

Corollary 2.10 Let $M, N \in \Lambda$ and $M \preceq N$. If M^* is a normal form of M and N^* is a normal form of N, then $M^* \preceq N^*$. □

PROOF OUTLINE. For each M' such that $M \rightarrow M'$, there is a N' such that $N \rightarrow^* N'$ and $M' \preceq N'$; and vice versa. By Proposition 2.9, the proof then follows. ◇

Proposition 2.11 (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 environment ρ. □

PROOF OUTLINE. By the definition of $\preceq$ and $\rightarrow^*$. ◇

Proposition 2.12 (Incompleteness) There exist normal forms $M, N \in \Lambda$ such that $[\![M]\!]\rho \sqsubseteq [\![N]\!]\rho$ for all environment ρ, but $M \preceq N$ is not provable. □

As an example of incompleteness, let us consider the following two $\Lambda_{(2\to 2)\to 2\to 2}$ terms,

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

It can be shown that $[\![M]\!] = [\![N]\!]$, but neither $M \preceq N$ nor $N \preceq M$ is provable. Since $\beta \sqcap \sqcup\boldsymbol{d}$–reduction is incomplete with respect to the semantics of the language Λ, Λ is not considered to be an ideal representation for the elements in domain $\bigcup_{\sigma\in\Gamma} D_\sigma$.

3 The Language Λ^0 and Its Properties

We define a restricted language Λ^0 of Λ, with the intention to make $\beta \sqcap \sqcup\boldsymbol{d}$–reduction complete with respect to the semantic interpretation of language Λ^0. Often a term T in Λ is written 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 *matrix* of T.

Definition 3.13 The sub–language Λ^0 of Λ is inductively defined as follows.

- $0, 1 \in \Lambda^0_2$; and
- $\lambda\vec{x}.N \in \Lambda^0_{\sigma_1\to\ldots\to\sigma_n\to 2}$ if
  - $\vec{x}$ consists of variables of types $\sigma_1,\ldots,\sigma_n$, and N contains no free variable other than those from $\vec{x}$,
  - $N \in \Lambda_2$ and it is in minimal disjunctive normal form, and
  - each atomic term in N is either $0, 1$, 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 $1 \le k \le m$.

□

Suppose that we have commutativity and associativity reduction rules for $\sqcap$ (the conjunctive operator) and $\sqcup$ (the disjunctive operator). We say a term N is in *minimal disjunctive normal form* if $N \equiv \bigsqcup_{i\in I} \bigsqcap_{j\in J_i} N_{i,j}$ for some index sets I and $J_{i\in I}$, $N_{i,j}$ is not a conjunction nor a disjunction of other sub–terms, and N cannot be further reduced by the $\sqcap\sqcup$ rules. We call the term $N_{i,j}$ an *atomic* term. Note that, by the definition of Λ^0, each term in Λ^0 is closed and in $\beta \sqcap \sqcup\boldsymbol{d}$ normal form. Also, it can been shown by induction that, for each given type $\sigma \in \Gamma$, there are only finite number of Λ^0_σ terms.

Example 3.14 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 commutativity and associativity):

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

But the following 5 terms are not in $\Lambda^0_{(2\to 2)\to(2\to 2)}$:

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

That is because the first term's matrix is not of type 2, the arguments in the second's and the third's function applications are not in language Λ^0_2, and the matrices of last two are not in the minimal disjunctive normal forms. □

In addition to the soundness property inherited from language Λ, we also have the following nice results for language Λ^0.

Proposition 3.15 Let $M, N \in \Lambda^0$. Then the normal forms for $(MN), (\lambda x.M), (M \sqcap N)$, and $(M \sqcup N)$ are all in Λ^0. □

Proposition 3.16 (Definability) For every element $f \in D_\sigma$ there is a term $F \in \Lambda^0_\sigma$ such that $[\![F]\!] = f$. □

PROOF OUTLINE. 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 the inductive 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.

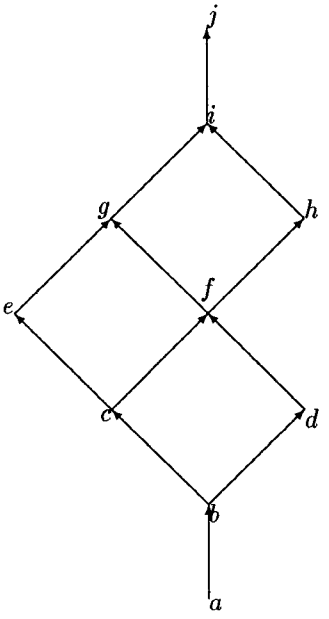
The step function $step_{a,b}$ in domain $D_{\tau\to\gamma}$, where $a \in D_\tau$ and $b \in D_\gamma$, is defined by

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

Furthermore, an element $f \in D_\sigma$ can be expressed as the least upper bound of a set of step functions in D_σ. That is, $f = \bigsqcup_{i\in I} step_{a_i,b_i}$ for some index set I. This construction is standard and can be found, for example, in Plotkin [11]. 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

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



$$j = [\lambda x.0 \mapsto \lambda x.1,\ \lambda x.x \mapsto \lambda x.1,\ \lambda x.1 \mapsto \lambda x.1]$$

$$i = [\lambda x.0 \mapsto \lambda x.x,\ \lambda x.x \mapsto \lambda x.1,\ \lambda x.1 \mapsto \lambda x.1]$$

$$h = [\lambda x.0 \mapsto \lambda x.0,\ \lambda x.x \mapsto \lambda x.1,\ \lambda x.1 \mapsto \lambda x.1]$$

$$g = [\lambda x.0 \mapsto \lambda x.x,\ \lambda x.x \mapsto \lambda x.x,\ \lambda x.1 \mapsto \lambda x.1]$$

$$f = [\lambda x.0 \mapsto \lambda x.0,\ \lambda x.x \mapsto \lambda x.x,\ \lambda x.1 \mapsto \lambda x.1]$$

$$e = [\lambda x.0 \mapsto \lambda x.x,\ \lambda x.x \mapsto \lambda x.x,\ \lambda x.1 \mapsto \lambda x.x]$$

$$d = [\lambda x.0 \mapsto \lambda x.0,\ \lambda x.x \mapsto \lambda x.0,\ \lambda x.1 \mapsto \lambda x.1]$$

$$c = [\lambda x.0 \mapsto \lambda x.0,\ \lambda x.x \mapsto \lambda x.x,\ \lambda x.1 \mapsto \lambda x.x]$$

$$b = [\lambda x.0 \mapsto \lambda x.0,\ \lambda x.x \mapsto \lambda x.0,\ \lambda x.1 \mapsto \lambda x.x]$$

$$a = [\lambda x.0 \mapsto \lambda x.0,\ \lambda x.x \mapsto \lambda x.0,\ \lambda x.1 \mapsto \lambda x.0]$$

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

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

$$step_{a,b}\ x\ z_1\ \ldots\ z_m = (\bigsqcap\{(x\ y_1\ \ldots\ y_n) \mid a\ y_1\ \ldots\ y_n = 1\}) \sqcap (b\ z_1\ \ldots\ z_m).$$

By inductive hypotheses, all elements in domains $D_{\tau_1}, \ldots, D_{\tau_n}, D_\tau$, and D_γ, are definable. Hence, the above step function can be defined in language Λ^0_σ as a normal form of the following term

$$\lambda x.\lambda z_1.\ldots.\lambda z_m.\ (\bigsqcap_{(a\ y_1\ \ldots\ y_n)=1}\ (x\ Y_1\ \ldots\ Y_n)) \sqcap (B\ z_1\ \ldots\ z_m),$$

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

Since 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,

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

Normal forms of F are in language Λ^0_σ. ◇

Example 3.17 There is a function y in domain $D_{((2\to 2)\to(2\to 2))\to(2\to 2)}$ such that for all elements x in domain $D_{(2\to 2)\to(2\to 2)}$, $(y\ x)$ is the least fixed point of x. Can we find a term Y in language $\Lambda^0_{((2\to 2)\to(2\to 2))\to(2\to 2)}$ such that $[\![Y]\!] = y$?

Before start calculating Y, let us first draw a diagram of domain $D_{(2\to 2)\to(2\to 2)}$. The diagram in Figure 1 illustrates the ordering of the 10 elements in domain $D_{(2\to 2)\to(2\to 2)}$. The functionalities of the 10 elements are also described as maps from domain $D_{2\to 2}$ to $D_{2\to 2}$. Note that in describing the maps, we confuse the syntactical lambda notations to the semantic elements they denote. For example, $\lambda x.0$ is meant to be the element in domain $D_{2\to 2}$ which always return 0 when applied, it is not meant to be a term in the language $\Lambda^0_{2\to 2}$.

It is not difficult to see that the least fixed point function y can be expressed as

$$y = \bigsqcup\{step_{e,(\lambda x\ x)}, step_{i,(\lambda x\ 1)}\}.$$

By a little calculation and simplification, we can obtain the following fixed point term Y,

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

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

(As a side note, the 10 elements in domain $D_{(2\to 2)\to(2\to 2)}$ happen to be defined in Example 3.14 by the 10 $\Lambda^0_{(2\to 2)\to(2\to 2)}$ terms.) □

Proposition 3.18 (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 $[\![M]\!] \not\sqsubseteq [\![N]\!]$, which is a contradiction.

Suppose that terms M and N are of type $\sigma = \sigma_1 \to \ldots \to \sigma_n \to \mathcal{2}$. 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 a structural induction on type σ. The base case is $\sigma = \mathcal{2}$, in which the proposition is true. We want to show that the proposition is true for type $\sigma = \sigma_1 \to \ldots \to \sigma_n \to \mathcal{2}$, 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 indexes $k \in K$, $(\bigsqcap_{j \in J_i} M_{i,j}) \preceq (\bigsqcap_{l \in L_k} N_{k,l})$ is not provable. Based on the conjunctive term $\bigsqcap_{j \in J_i} M_{i,j}$, we will construct an environment ρ such that $[\![\bigsqcup_{i \in I} \bigsqcap_{j \in J_i} M_{i,j}]\!]\rho = 1$ but $[\![\bigsqcup_{k \in K} \bigsqcap_{l \in L_k} N_{k,l}]\!]\rho = 0$. That is, $[\![M]\!] \not\sqsubseteq [\![N]\!]$. This will complete the proof.

Given i, we now describe how to construct such an environment ρ. Let x_h be a bound variable in $\vec{x}$. Suppose x_h is of type $\mathcal{2}$ and $M_{i,j} \equiv x_h$ for some $j \in J_i$, then x_h is mapped 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 $\sigma_h = \tau_{h_1} \to \ldots \tau_{h_m} \to \mathcal{2}$, then define a set P by

$$P = \{\langle [\![e_{h_1}]\!], \ldots, [\![e_{h_m}]\!] \rangle \mid M_{i,j} \equiv x_h \; e_{h_1} \; \ldots \; e_{h_m}, \text{ for some } j \in J_i\},$$

and map x_h to the following function in ρ,

$$\lambda y_1 \ldots . \lambda y_{h_m} . \; \textit{if } (p \sqsubseteq \langle y_1, \ldots, y_{h_m} \rangle \textit{ for some } p \in P) \textit{ then } 1 \textit{ else } 0.$$

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

It is easy to see that $[\![\bigsqcap_{j \in J_i} M_{i,j}]\!]\rho = 1$. This also gives us $[\![\bigsqcup_{i \in I} \bigsqcap_{j \in J_i} M_{i,j}]\!]\rho = 1$. It remains to show that $[\![\bigsqcup_{k \in K} \bigsqcap_{l \in L_k} N_{k,l}]\!]\rho = 0$. Since for all indexes $k \in K$, $(\bigsqcap_{j \in J_i} M_{i,j}) \preceq (\bigsqcap_{l \in L_k} N_{k,l})$ is not provable, then, for each fixed $k \in K$, there exists an index $l \in L_k$ such that for all indexes $j \in J_i$, $M_{i,j} \preceq N_{k,l}$ is not provable. If $N_{k,l} \equiv x_h$, where x_h in $\vec{x}$ and of type $\mathcal{2}$, then x_h must not occur in the conjunctive term $\bigsqcap_{j \in J_i} M_{i,j}$. By the construction of the environment ρ, x_h is mapped to 0. Hence, $[\![\bigsqcap_{l \in L_k} N_{k,l}]\!]\rho = 0$. The only other case is $N_{k,l} \equiv x_h \; e_{h_1} \; \ldots \; e_{h_m}$, where x_h is of type $\sigma_h = \tau_{h_1} \to \ldots \tau_{h_m} \to \mathcal{2}$ and $e_{h_1} \in \Lambda^0_{\tau_{h_1}}, \ldots, e_{h_m} \in \Lambda^0_{\tau_{h_m}}$. If x_h never occurs in $\bigsqcap_{j \in J_i} M_{i,j}$, then by the definition of ρ, $[\![N_{k,l}]\!]\rho = 0$. Otherwise, we must have some $M_{i,j} \equiv x_h \; f_{h_1} \; \ldots \; f_{h_m}$, where $f_{h_1} \in \Lambda^0_{\tau_{h_1}}, \ldots, f_{h_m} \in \Lambda^0_{\tau_{h_m}}$, but not all of $f_{h_1} \preceq e_{h_1}, \ldots, f_{h_m} \preceq e_{h_m}$ are provable. By inductive hypothesis, we have $[\![f_{h_1}]\!] \not\sqsubseteq [\![e_{h_1}]\!]$, or ..., or $[\![f_{h_m}]\!] \not\sqsubseteq [\![e_{h_m}]\!]$. Then, by the construction of environment ρ, we also have $[\![N_{k,l}]\!]\rho = 0$. That is, $[\![\bigsqcap_{l \in L_k} N_{k,l}]\!]\rho = 0$ in case when $N_{k,l}$ is a function application. Since, in all cases, $[\![\bigsqcap_{l \in L_k} N_{k,l}]\!]\rho = 0$ for any fixed $k \in K$, we have $[\![\bigsqcup_{k \in K} \bigsqcap_{l \in L_k} N_{k,l}]\!]\rho = 0$. This completes the proof. ◇

Example 3.19 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

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

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

$$\begin{aligned} [\![Y]\!] \; g \; x = 1 \quad \text{but} \quad [\![Z]\!] \; g \; x = 0, \text{ and} \\ [\![Y]\!] \; g' \; x' = 0 \quad \text{but} \quad [\![Z]\!] \; g' \; x' = 1. \end{aligned}$$

This will show that $[\![Y]\!] \not\sqsubseteq [\![Z]\!]$ nor $[\![Z]\!] \not\sqsubseteq [\![Y]\!]$.

The conjunctive term $(g \; (\lambda x . 0) \; 1) \sqcap (g \; (\lambda x . x) \; 0)$ in Y's matrix cannot be proved to be syntactically weaker than either one of the two conjunctive terms in Z's matrix. It follows that we can choose

$$\begin{aligned} g = \lambda f . \lambda x . \\ \quad \textit{if } \langle \lambda x . 0, 1 \rangle \sqsubseteq \langle f, x \rangle \text{ or } \langle \lambda x . x, 0 \rangle \sqsubseteq \langle f, x \rangle \\ \quad \textit{then } 1 \textit{ else } 0, \\ x = 0, \end{aligned}$$

to make $[\![Y]\!] \; g \; x = 1$ and $[\![Z]\!] \; g \; x = 0$. Likewise, the witness of the unprovability of $Z \preceq Y$ is the conjunctive term $(g \; (\lambda x . x) \; 1) \sqcap x$ in Z's matrix. Hence we choose

$$\begin{aligned} g' = \lambda f . \lambda x . \textit{if } \langle \lambda x . x, 1 \rangle \sqsubseteq \langle f, x \rangle \textit{ then } 1 \textit{ else } 0, \\ x' = 1, \end{aligned}$$

to make $[\![Y]\!] \; g' \; x' = 0$ and $[\![Z]\!] \; g' \; x' = 1$. □

Corollary 3.20 Let $M, N \in \Lambda^0$. Then, $M \simeq N$ implies $[\![M]\!] = [\![N]\!]$, and $[\![M]\!] = [\![N]\!]$ implies $M \simeq N$. □

Proposition 3.21 Let $\sigma \in \Gamma$. Then there is a term $Y \in \Lambda^0_{(\sigma\to\sigma)\to\sigma}$ such that for all terms $F \in \Lambda^0_{\sigma\to\sigma}$,

1. $[\![YF]\!] = \textit{fix } [\![F]\!]$, where *fix* is the least fixed point function; and
2. if M is a normal form of (YF) and N is a normal form of $(F(YF))$, then $M \simeq N$.

□

PROOF OUTLINE.

1. By Proposition 3.16.
2. By Proposition 2.11, Corollary 3.20, and the above.

◇

The above properties of language Λ^0 enable us to syntactically calculate the least fixed point of a term $M \in \Lambda^0$. In fact, there are two ways to do this. That first method

uses Propositions 3.16 and 3.21 to find the fixed point term $Y \in \Lambda^0_{(\sigma\to\sigma)\to\sigma}$ for the given type $\sigma \in \Gamma$, and then calculate a normal form for (YM). The normal forms for (YM) are the least fixed point of M. These terms are equivalent under the relation $\simeq$.

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 $N^{(k+1)}$ is a normal form of $(MN^{(k)})$, until it finds $N^{(i+1)} \preceq N^{(i)}$. Term $N^{(i)}$ is then the least fixed point of M. The iteration is ensured to be terminated because, for a given type $\sigma \in \Gamma$, there are only finite number of Λ^0_σ terms.

Example 3.22 Suppose we want to calculate the least fixed point of the following $\Lambda^0_{(2\to2)\to(2\to2)}$ term M, defined as

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

By using Proposition 3.16, we can find a least fixed point term $Y \in \Lambda^0_{((2\to2)\to(2\to2))\to(2\to2)}$. For example,

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

as defined in Example 3.17 is such a fixed point term. Furthermore, (YM) has $\lambda x\,.\,0$ as a normal form. Therefore, $\lambda x\,.\,0$ is the least fixed point of M.

Or we can use a fixed point approximation sequence starting with $\lambda x\,.\,0$. Since $(M\ (\lambda x\,.\,0)) \to^* \lambda x\,.\,0$, we have $\lambda x\,.\,0$ as the least fixed point of M. □

Note that the above results only apply to language Λ^0. For a closed term M in language Λ but not in language Λ^0, we must first translate M into a semantically equivalent term $M' \in \Lambda^0$. There is a systematic way to do the translation. The translation is based on the idea that a term $(MN) \in \Lambda$ can be rewritten into a semantically equivalent term which includes M and N, but does not have M applied to N. The translation proceeds until all the functional applications in a term satisfy the requirements of Λ^0 (*i.e.*, application (MN) must have M as a bound variable and N as a Λ^0 term). $\beta \sqcap \sqcup\boldsymbol{d}$–reduction is also performed during the translation process to make sure that the final result is a Λ^0 term.

Example 3.23 Suppose we have a $\Lambda_{(2\to2\to2)\to(2\to2\to2)}$ term M, defined as

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

We want to translate M into a $\Lambda^0_{(2\to2\to2)\to(2\to2\to2)}$ term. First, we observe that the infix operator for function application, $\bullet$, of type $(2 \to 2) \to (2 \to 2)$, can be defined in $\Lambda^0_{(2\to2)\to(2\to2)}$ as

$$\bullet \equiv \lambda h\,.\,\lambda z\,.\,(h\ 0) \sqcup ((h\ 1) \sqcap z).$$

We then translate M as the following,

$$\begin{aligned}
& \lambda f\,.\,\lambda x\,.\,\lambda y\,.\,(f\ x)\ y \\
\Rightarrow\ & \lambda f\,.\,\lambda x\,.\,\lambda y\,.\,(f\ x) \bullet y \\
\Rightarrow\ & \lambda f\,.\,\lambda x\,.\,\lambda y\,.\,(f\ x\ 0) \sqcup ((f\ x\ 1) \sqcap y) \\
\Rightarrow\ & \lambda f\,.\,\lambda x\,.\,\lambda y\,.\,((f\ 0\ 0) \sqcup ((f\ 1\ 0) \sqcap x)) \sqcup \\
& \quad (((f\ 0\ 1) \sqcup ((f\ 1\ 1) \sqcap x)) \sqcap y) \\
\Rightarrow\ & \lambda f\,.\,\lambda x\,.\,\lambda y\,.\,(f\ 0\ 0) \sqcup ((f\ 1\ 0) \sqcap x) \sqcup \\
& \quad ((f\ 0\ 1) \sqcap y) \sqcup ((f\ 1\ 1) \sqcap x \sqcap y).
\end{aligned}$$

The last term is in language Λ^0. □

4 Remarks

We would like to make several remarks regarding the feasibility of using the outlined syntactic approach to compute least fixed points. One of the questions comes from the observation that, for a closed term $M \in \Lambda$, a semantically equivalent term $M' \in \Lambda^0$ seems to be much longer than M, which makes the syntactic method unattractive. Furthermore, in general, how do we translate an abstract semantics of a functional program into language Λ, especially when the ground type is not *2*.

To answer the first question, we now define a language Λ^1. Language Λ^1 is a superset of Λ^0, looks more like usual functional languages, and often provides shorter terms than those in Λ^0.

Definition 4.24 The sub–language Λ^1 of Λ is inductively defined as follows.

- $0, 1 \in \Lambda^1_2$; and
- $\lambda \vec{x}\,.\,N \in \Lambda^0_{\sigma_1\to\,..\to\sigma_n\to2}$ if
  - $\vec{x}$ consists of variables of types $\sigma_1, \ldots, \sigma_n$, and N contains no free variable other than those from $\vec{x}$,
  - $N \in \Lambda_2$ and it is in minimal disjunctive normal form, and
  - each atomic term in N is either 0, 1, 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 is either a variable from $\vec{x}$ or is in $\Lambda^1_{\tau_{i_k}}$ for each $1 \le k \le m$.

□

The only difference between Λ^1 and Λ^0 is that the atomic terms in Λ^1 are allowed to have function applications whose arguments are bound variables. 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 Λ^1 and are semantically equivalent, but cannot be shown

to be syntactically equivalent under the $\simeq$ relation. However, the two methods described in Section 3 for computing fixed points still work. That is, the fixed point term $Y \in \Lambda^0_{(\sigma\to\sigma)\to\sigma}$ works both for terms in language $\Lambda^0_{\sigma\to\sigma}$ and $\Lambda^1_{\sigma\to\sigma}$. And the approximation–sequence method is guaranteed to be converging because, for a given type $\sigma \in \Gamma$, there are only a finite number of terms in Λ^1_σ.

Example 4.25 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 3.22, we have $\lambda x.0$ as a normal form of (YF), and thus as a least fixed point of F. The approximation method also finds $\lambda x.0$ as the least fixed point.

Another interesting example can be found in [3]. It is a $\Lambda_{((2\to 2)\to 2)\to((2\to 2)\to 2)}$ term H, 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 $B \equiv \lambda g.0$, we will have

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

which does not reach a limit under relation $\preceq$. However, if H is translated (as described in Example 3.23) into a semantically equivalent $\Lambda^1_{((2\to 2)\to 2)\to((2\to 2)\to 2)}$ term

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

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

Is it difficult to translate from a typed functional program (or the abstract semantics of the program) to a semantic equivalent Λ^1 term? The translation will be straightforward if we can encode the basic semantic domains used in the functional programming language into domains $D_\sigma, \sigma \in \Gamma$. Usually, this is rather easy for an abstract semantics because their basic domains are finite. For example, a three element domain $\mathcal{3} = \{a, b, c\}$, with ordering $a \sqsubseteq b \sqsubseteq c$ can be embedded in domain $D_{2\to 2}$ with a, b, and c defined as $a = \lambda x.0$, $b = \lambda x.x$, and $c = \lambda x.1$. The boolean domain $Bool = \{0, t, f\}$, with ordering $0 \sqsubseteq t$ and $0 \sqsubseteq f$ but neither $f \sqsubseteq t$ nor $t \sqsubseteq f$, can be encoded in language $\Lambda^0_{2\to 2\to 2}$ by $0 \equiv \lambda x.\lambda y.x \sqcap y$, $t \equiv \lambda x.\lambda y.x$, and $f \equiv \lambda x.\lambda y.y$. Also, by η equality (*i.e.*, $M \leftrightarrow_\eta \lambda x.Mx$), higher–order functions can be easily encoded in Λ^1. For example, the strictness property for higher–order *if* functional, which is in domain $D_{2\to\sigma\to\sigma\to\sigma}$ with type $\sigma = \tau_1 \to \tau_2 \to \ldots \to \tau_n \to 2$, can be defined as

$$\begin{aligned} \mathit{if} \equiv \lambda p.\lambda f.\lambda g.\lambda x_1.\lambda x_2.\ldots.\lambda x_n. \\ (p \sqcap (f\ x_1\ x_2\ \ldots\ x_n)) \sqcup (p \sqcap (g\ x_1\ x_2\ \ldots\ x_n)), \end{aligned}$$

which is in language $\Lambda^1_{2\to\sigma\to\sigma\to\sigma}$.

Regarding the complexity of the proposed syntactic method, is it better than simply using the frontier method? We believe it is better, especially if we use the approximation method on language Λ^1. We observe that a functional program often has simple textual structure. That is, function applications in typical functional programs usually have bound variables as arguments. This makes the translation from a functional program into a Λ^1 term easy, and the length of the resulted 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 in the frontier method. However, an approximation sequence in the syntactic method usually leads to a limit more quickly than an approximation in the frontier method, because the former utilizes the textual information from the program, while the latter blindly searches along the chains in the semantic domains. Of course, in the worst cases, the syntactic method will, just as the frontier method, require exponential running time (with respect to the program length).

On the other hand, there remains several engineering issues to be explored to better utilize the outlined syntactical approach. For example, we can define a representation for terms in language Λ^0 (and Λ^1) such that the provability of $\preceq$ relationship between them can be checked easier. A lexicographic encoding of Λ^0 terms according to the sequence of their bound variables comes to mind naturally. We can also develop safe approximation schemes based on the proposed syntactical method. The approximation scheme will calculate less accurate fixed points, but do it faster. For example, a reduction rule like

$$(M \sqcap N) \to_{\sqcap_{\mathrm{approx}}} M,$$

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

5 Conclusion and Comparison to Other Works

We have shown how to develop a syntactic approach to fixed point computation on finite domains. The syntactic method is sound and complete with with respect to the semantics of 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 the work of Abramsky [1] and Jensen [7,8]. 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 fixed points on finite domains and their corresponding calculus. While their work usually provides a decidable theory without giving an explicit proof strategy, the augmented, simply-typed λ-calculus in our approach provides a simple way to compute the desired results.

A Outline of Some Proofs

We need some technical definitions and lemmas in order to show that Proposition 2.9 is true. We define two new reduction relations: $\boldsymbol{c}$ (for commutativity) and $\boldsymbol{a}$ (for associativity).

Definition A.26 The reduction relations $\boldsymbol{c}$ and $\boldsymbol{a}$ on language Λ are defined as follows.

- $\boldsymbol{c} = \{(M \sqcap N,\ N \sqcap M) \mid M, N \in \Lambda_2\} \cup \{(M \sqcup N,\ N \sqcup M) \mid M, N \in \Lambda_2\}$, and
- $\boldsymbol{a} = \{(L \sqcap (M \sqcap N), (L \sqcap M) \sqcap N) \mid L, M, N \in \Lambda_2\} \cup \{((L \sqcap M) \sqcap N, L \sqcap (M \sqcap N)) \mid L, M, N \in \Lambda_2\} \cup \{(L \sqcup (M \sqcup N), (L \sqcup M) \sqcup N) \mid L, M, N \in \Lambda_2\} \cup \{((L \sqcup M) \sqcup N, L \sqcup (M \sqcup N)) \mid L, M, N \in \Lambda_2\}$

□

It is clear that neither $\boldsymbol{c}$ nor $\boldsymbol{a}$ is strongly normalizing.

Lemma A.27

1. $\sqcap \sqcup \boldsymbol{dca}$ is Church-Rosser; and
2. $\rightarrow_\beta^*$ and $\rightarrow_{\sqcap \sqcup dca}^*$ commute.

□

By the above results, we know that $\boldsymbol{\beta} \sqcap \sqcup \boldsymbol{dca}$ is Church-Rosser. The following lemma is easy but very useful for our goal.

Lemma A.28 Let $M \in \Lambda$. If M is in $\boldsymbol{\beta} \sqcap \sqcup \boldsymbol{d}$-normal form and $M \rightarrow_{ca} M'$, then M' is in $\boldsymbol{\beta} \sqcap \sqcup \boldsymbol{d}$-normal form and $M \simeq M'$. □

Now we can show the following result.

Proposition A.29 Let $M \in \Lambda$. If both M' and M'' are $\boldsymbol{\beta} \sqcap \sqcup \boldsymbol{d}$-normal forms of M, then $M' \simeq M''$. □

PROOF OUTLINE. Since $\boldsymbol{\beta} \sqcap \sqcup \boldsymbol{dca}$ is Church-Rosser, there is a $N \in \Lambda$ such that $M' \rightarrow_{\beta \sqcap \sqcup dca}^* N$ and $M'' \rightarrow_{\beta \sqcap \sqcup dca}^* N$. In fact, by Lemma A.28, we have $M' \rightarrow_{ca}^* N$ and $M'' \rightarrow_{ca}^* N$. Since relation $\simeq$ is reflexive and symmetrical, it follows that $M' \simeq M''$. ◇

References

[1] Samson Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51(1–2):1–77, March 1991.

[2] Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[3] Chris Clack and Simon L. Peyton Jones. Strictness analysis — a practical approach. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 35–49. Nancy, France, September 1985. Lecture Notes in Computer Science, Volume 201, Springer-Verlag.

[4] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theorectical Computer Science*. Cambridge University Press, 1989.

[5] Sebastian Hunt. Frontiers and open sets in abstract interpretation. In *Functional Programming Languages and Computer Architecture*, pages 1–11. Imperial College, London, U.K., September 1989. A.C.M./Addison-Wesley.

[6] Sebastian Hunt and Chris Hankin. Fixed points and frontiers: a new perspective. *Journal of Functional Programming*, 1(1):91–120, January 1991.

[7] Thomas P. Jensen. Abstract interpretation *vs.* type inference: A topological perspective. In Simon L. Peyton Jones, Graham Hutton, and Caresten Kehler Holst, editors, *Functional Programming, Glasgow 1990: Proceedings of the 1990 Glasgow Workshop*, pages 141–145. Ullapool, Scotland, U.K., August 1990. Springer-Verlag.

[8] Thomas P. Jensen. Strictness analysis in logical form. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 352–366. Cambridge, Massachusetts, U.S.A., August 1991. Lecture Notes in Computer Science, Volume 523, Springer-Verlag.

[9] Chris Martin and Chris Hankin. Finding fixed points in finite lattices. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 426–445. Portland, Oregon, U.S.A., September 1987. Lecture Notes in Computer Science, Volume 274, Springer-Verlag.

[10] Simon L. Peyton Jones and Chris Clack. Finding fixpoints in abstract interpretation. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 11, pages 246–265. Ellis Horwood, 1987.

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

[12] Jonathan Hood Young. *The Theory and Practice of Semantic Program Analysis for Higher-Order Functional Programming Languages*. PhD thesis, Department of Computer Science, Yale University, May 1989.