Journal of Functional Programming **1** (1): 91–120, January 1991

Fixed points and frontiers: a new perspective

SEBASTIAN HUNT AND CHRIS HANKIN
Department of Computing, Imperial College, London

Abstract

Abstract interpretation is the collective name for a family of semantics-based techniques for compile-time analysis of programs. One of the most costly operations in automating such analyses is the computation of fixed points. The frontiers algorithm is an elegant method, invented by Chris Clack and Simon Peyton Jones, which addresses this issue.

In this article we present a new approach to the frontiers algorithm based on the insight that frontiers represent upper and lower subsets of a function's argument domain. This insight leads to a new formulation of the frontiers algorithm for higher-order functions, which is considerably more concise than previous versions.

We go on to argue that for many functions, especially in the higher-order case, finding fixed points is an intractable problem unless the sizes of the abstract domains are reduced. We show how the semantic machinery of abstract interpretation allows us to place upper and lower bounds on the values of fixed points in large lattices by working within smaller ones.

Capsule review

Abstract interpretation is an important and well-studied method for performing compile-time analysis of functional programs. So far much more attention has been paid to its theoretical properties than its implementation in practice. A serious problem which must be addressed by any practical implementation is that of finding fixed points for the abstract functions. This is simple enough in theory, but a naive algorithm has exponential complexity, so more subtle techniques are required.

In 1987, Clack and Peyton Jones proposed an algorithm for finding fixed points, based on a representation of functions called 'frontiers', which offered substantial performance benefits in typical cases. Their paper was argued quite informally, there was no proof of correctness, and the method was restricted to first-order functions over flat domains.

Hunt and Hankin have taken this basic idea, worked out the underlying theory, and written a clear and detailed exposition of the algorithm and its proof of correctness. Further, they have extended the applicability of the algorithm to structured domains and higher-order functions.

Finally, based on their theoretical foundation, they are able to propose an important new approximation technique in section 5. A straightforward extension of the frontiers algorithm to higher-order functions turns out to be unacceptably slow, and section 5 shows how to cut down the size of the lattice so as to get safe bounds on the required result in acceptable time.

The paper is quite formal, but it will repay careful reading. It is a classic demonstration of the exciting interplay between theory and practice that characterizes research in functional programming.

1 Introduction

Abstract interpretation is the collective name for a family of semantics-based techniques for the compile-time analysis of computer programs. In this article we restrict our attention to abstract interpretation of typed functional languages using finite lattices, in the style of Burn, Hankin and Abramsky (1986) and Burn (1987). A more general introduction to abstract interpretation is given by Abramsky and Hankin (1987).

Consider a simple typed lambda-calculus with constants:

$$\begin{array}{ll} \text{types:} & \sigma ::= \mathit{int} \mid \mathit{bool} \mid \sigma \to \sigma \mid \sigma \times \sigma \\ \text{expressions:} & e ::= x^{\sigma} \mid c_{\sigma} \mid ee \mid \lambda x^{\sigma}.e. \end{array}$$

The expressions are assumed to be well typed.

An *interpretation*, I, for this language consists of two parts. The first part specifies a complete partial order (cpo), D^I_{σ}, for each base type, σ, in the language; interpretations for the compound types are induced in the obvious way:

$$\begin{aligned} D^I_{\sigma_1 \to \sigma_2} &= [D_{\sigma_1} \to D_{\sigma_2}] \quad \text{(the cpo of continuous functions)} \\ D^I_{\sigma_1 \times \sigma_2} &= D_{\sigma_1} \times D_{\sigma_2}. \end{aligned}$$

The second part specifies a value, $c^I_{\sigma} \in D^I_{\sigma}$, for each of the language's constants, c_{σ}. This induces an interpretation for all expressions (relative to some environment, ρ) as follows:

$$\begin{aligned} [\![c_{\sigma}]\!]^I \rho &= c^I_{\sigma} \\ [\![x^{\sigma}]\!]^I \rho &= \rho(x^{\sigma}) \\ [\![e_1\, e_2]\!]^I \rho &= ([\![e_1]\!]^I \rho)([\![e_2]\!]^I \rho) \\ [\![\lambda x^{\sigma}.e]\!] \rho &= \lambda \xi \in D^I_{\sigma}.[\![e]\!]^I \rho[x^{\sigma} \mapsto \xi]. \end{aligned}$$

We will assume that the language has constants, $Y_{(\sigma \to \sigma) \to \sigma}$, and that for each interpretation, I, $Y^I_{(\sigma \to \sigma) \to \sigma} = \mathit{fix}_{D^I_{\sigma}}$, where

$$\mathit{fix}_D = \lambda f \in [D \to D].\bigsqcup_{i=0}^{\infty} f^n \bot.$$

For *abstract* interpretations we insist that the interpretations of base types (and hence of all types) are finite lattices.

A much-studied example is the use of abstract interpretation for strictness analysis. A function $f: D \to D'$ is said to be strict if its result is undefined whenever its argument is undefined, i.e. if

$$f \bot_D = \bot_{D'}.$$

Mycroft (1981) shows that an abstract interpretation with the domain **2**, i.e. ($\{0, 1\}$, $0 \sqsubseteq 1$), can be used for the strictness analysis of a first-order functional language. This interpretation is related to the standard interpretation of the language by means of an abstraction function mapping $\bot$ to 0 and every other value to 1. Strict binary functions, such as $+$ and $-$, are given the value $\lambda xy.x \sqcap y$ (since this evaluates to 0 if and only if either argument is 0) while the conditional has the value $\lambda xyz.x \sqcap (y \sqcup z)$.

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

Mycroft's interpretation is generalized to a higher-order language in Burn, Hankin and Abramsky (1986), and an interpretation for lists is introduced in Wadler (1987).

When a function definition is recursive it is necessary to find the least fixed point of its associated functional under the abstract interpretation. The sequence $f^n \perp$ in the above definition of *fix* is known as the Ascending Kleene Chain (AKC); each element of this chain is an approximation to the next. To find the least fixed point in an abstract interpretation it is sufficient to compute successive members of the AKC until two are found to be equal. When this happens the series has converged and the least fixed point has been found. This is guaranteed to happen in a finite number of iterations since the domains are of finite height.

Although finite, the domains can still be very large because of the exponential growth of a function space in the size of the argument domain. Thus, an efficient method of representing the value of each approximation is needed. This is the purpose of frontiers.

The use of frontiers as a compact representation of functions over finite lattices was first proposed in Clack and Peyton Jones (1985), where an algorithm for establishing the frontiers of a function is developed. This algorithm is restricted to first-order functions in function spaces of the form $[\mathbf{2}^n \to \mathbf{2}]$. In Martin and Hankin (1987) the algorithm is extended to cope with higher-order functions.

In this article we present a new approach to frontiers based on the insight that they represent upper and lower subsets of a function's argument domain. This insight leads to a new formulation of the frontiers algorithm for higher-order functions which is considerably clearer than previous versions.

We go on to argue that for many functions, especially in the higher-order case, finding fixed points is an intractable problem unless the sizes of the abstract domains are reduced. We show how the semantic machinery of abstract interpretation allows us to place upper and lower bounds on the values of fixed points in large lattices by working within smaller ones. We discuss the interaction of this approach with the use of frontiers.

A less technically detailed version of this article (Hunt, 1989) was presented at The Fourth International Conference on Functional Programming Languages and Computer Architecture, 1989.

In order that this article should be reasonably self-contained, the mathematical prerequisites for an understanding of the detailed technical development are briefly reviewed in appendix A.

2 Notation

In this section we introduce some notation used in the rest of the article.

Given a subset $X \subseteq D$, we denote the complement of X with respect to D by $C_D(X)$, or just $C(X)$ when D is clear from the context.

We write **2** for the lattice $(\{0, 1\}, \sqsubseteq)$ with $0 \sqsubseteq 1$.

The function $lift_D$: $D \to D_\perp$ is the obvious injection into a lifted poset.

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

For a function $f: A \to B$ and subsets $X \subseteq A$ and $Y \subseteq B$, the set $\{fa \mid a \in X\}$ is denoted by fX, and the set $\{a \in A \mid fa \in Y\}$ by $f^{-1}Y$.

Given a poset D and subset $X \subseteq D$,

(i) the *upward closure* of X, written $\uparrow_D X$, is the set $\bigcup_{d \in X} \{d' \in D \mid d \sqsubseteq d'\}$;

(ii) the *downward closure* of X, written $\downarrow_D X$, is the set $\bigcup_{d \in X} \{d' \in D \mid d' \sqsubseteq d\}$.

We will also write $\uparrow_D x$ and $\downarrow_D x$ as shorthand for the sets $\uparrow_D \{x\}$ and $\downarrow_D \{x\}$ respectively.

We will denote the poset of monotone functions from D to D', ordered pointwise on D, by $[D \to D']$.

We will be working with a family of finite lattices, $\mathscr{L}$, with the following inductive definition:

- $\mathbf{2} \in \mathscr{L}$
- $D_\perp \in \mathscr{L}$ if $D \in \mathscr{L}$
- $D_1 \times \ldots \times D_n \in \mathscr{L}$ if $D_i \in \mathscr{L}$, $1 \leqslant i \leqslant n$
- $[D \to D'] \in \mathscr{L}$ if D, $D' \in \mathscr{L}$

with the usual induced orderings for lifted posets and products. The definition of this family is motivated by the study of strictness analysis as described in Burn, Hankin and Abramsky (1986), Burn (1987) and Wadler (1987), but a similar structure has been used in other abstract interpretations such as binding time analysis (Jones, Sestoft and Søndergaard, 1985) and escape analysis (Goldberg and Park, 1990).

Throughout this article, we will routinely omit subscripts; for example, writing $\uparrow$ instead of $\uparrow_D$, when D is clear from the context.

At a number of places in what follows, we appeal to the concept of *duality* to avoid unnecessary duplication of proofs. Essentially, this hinges on the observation that a finite lattice 'turned upside-down' is still a finite lattice. The concept is formalized in appendix A.

3 Representation of functions

As was mentioned in the introduction, to find fixed points in an abstract interpretation, successive members of the AKC are compared until the series has converged. To compare two functions for equality it is necessary to check that the functions agree at every point in their domain. A naive approach to performing such comparisons evaluates both functions at each point in their domain in turn. Even for a first-order function in an interpretation using only the two-point domain, **2**, this involves 2^n tests, where n is the number of arguments. For higher-order functions the situation is much worse.

A better approach is to use a compact concrete representation for function values such that equality of functions coincides with equality of their representations.

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

3.1. Factoring functions

In this subsection and the next, we introduce our representation for functions. We will define a mapping, Rep, such that

$$\text{Rep}(f) = \text{Rep}(g) \quad \Leftrightarrow \quad f = g$$

Rep is defined by induction on the structure of the function spaces in $\mathscr{L}$:

$$\text{Rep}_{D_1 \to \ldots \to D_k \to D'}(f) = \text{Rep}_{(D_1 \times \ldots \times D_k) \to D'}(\text{uncurry}_k(f))$$

where D' is not a function space, $k > 1$ and $\text{uncurry}_k(f)(x_1, x_2, \ldots x_k) = f x_1 x_2 \ldots x_k$.

$$\text{Rep}_{D \to (D'_1 \times \ldots \times D'_n)}(f) = (\text{Rep}_{D \to D'_1}(\pi_1 \circ f), \ldots, \text{Rep}_{D \to D'_n}(\pi_n \circ f))$$

where π_i is the ith projection.

$$\text{Rep}_{D \to D'_\perp}(f) = (\text{Rep}_{D \to \mathbf{2}}(L_f), \text{Rep}_{D \to D'}(H_f)),$$

where $L_f \in [D \to \mathbf{2}]$ is the low factor of f:

$$L_f(x) = \begin{cases} 0 & \text{if} \quad f(x) = \perp \\ 1 & \text{otherwise} \end{cases}$$

and $H_f \in [D \to D']$ is the high factor of f:

$$H_f(x) = \begin{cases} \perp_{D'} & \text{if} \quad f(x) = \perp \\ d & \text{if} \quad f(x) = \mathit{lift}(d). \end{cases}$$

Functions in $[D \to D'_\perp]$ and their high and low factors are related by the following lemma:

Lemma 1

For all posets D, D', functions $f, g \in [D \to D'_\perp]$,

$$f \sqsubseteq g \quad \Leftrightarrow \quad (L_f \sqsubseteq L_g \quad \text{and} \quad H_f \sqsubseteq H_g)$$

Proof

$\Rightarrow$: Assume $f \sqsubseteq g$. There are two cases to consider for any $x \in D$:

$f(x) = \perp$: $L_f(x) = 0 \sqsubseteq L_g(x)$ and $H_f(x) = \perp_{D'} \sqsubseteq H_g(x)$

$f(x) \neq \perp$: $g(x) \neq \perp$, since $g(x) \sqsupseteq f(x)$.

Hence $L_f(x) = 1 = L_g(x)$ and $\mathit{lift}(H_f(x)) = f(x) \sqsubseteq g(x) = \mathit{lift}(H_g(x))$.

$\Leftarrow$: Assume $L_f \sqsubseteq L_g$ and $H_f \sqsubseteq H_g$. Again, there are two cases to consider for any $x \in D$:

$L_f(x) = 0$: $fx = \perp \sqsubseteq gx$.

$L_f(x) = 1$: $L_g(x) = 1$, since $L_f \sqsubseteq L_g$

so $\mathit{lift}(H_f(x)) = f(x)$ and $\mathit{lift}(H_g(x)) = g(x)$

Hence $f(x) \sqsubseteq g(x)$, since $H_f \sqsubseteq H_g$.

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

Corollary 2
For all posets D, D', functions $f, g \in [D \to D'_\perp]$

$$f = g \quad \Leftrightarrow \quad L_f = L_g \quad \text{and} \quad H_f = H_g \qquad \square$$

3.2 Functions in $[D \to \mathbf{2}]$

To complete the definition of the Rep function, we need to define the 'base' case: the function spaces $[D \to \mathbf{2}]$.

3.2.1 Upper sets as function representations

Perhaps an obvious candidate representation for a function $f \in [D \to \mathbf{2}]$ is the set $f^{-1}\{1\}$. To evaluate fx for $x \in D$ we simply check for membership of the representation set; if $x \in f^{-1}\{1\}$ then $fx = 1$, otherwise $fx = 0$. The set $f^{-1}\{0\}$ could be used in a similar manner. Comparison of two functions is straightforward given the following lemma:

Lemma 3
For all posets D, functions $f, g \in [D \to \mathbf{2}]$, the following are equivalent:

(i) $f \sqsubseteq g$
(ii) $f^{-1}\{1\} \subseteq g^{-1}\{1\}$
(iii) $f^{-1}\{0\} \supseteq g^{-1}\{0\}$

Proof
Trivial. $\square$

Since $\{1\}$ is upper and $\{0\}$ is lower, it follows from the definition of Alexandrov-continuity that for poset D, $f \in [D \to \mathbf{2}]$, $f^{-1}\{1\}$ is upper and $f^{-1}\{0\}$ is lower. In fact, there is a one-to-one correspondence between the upper (and lower) subsets of D and the elements of $[D \to \mathbf{2}]$, since each $f \in [D \to \mathbf{2}]$ is the characteristic function of an upper subset of D.

Lemma 4
For any poset D, the functions $\lambda g.g^{-1}\{1\}$ and $\lambda Z.\chi_Z$ establish an isomorphism between $[D \to \mathbf{2}]$ and $\Omega_A(D)$. That is:

$$(\lambda g.g^{-1}\{1\}) \circ (\lambda Z.\chi_Z) = \mathbf{id}_{[D \to \mathbf{2}]}$$

and

$$(\lambda Z.\chi_Z) \circ (\lambda g.g^{-1}\{1\}) = \mathbf{id}_{\Omega_A(D)}$$

Proof
Straightforward. $\square$

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

Corollary 5
For any poset D, for all $f, g \in [D \to \mathbf{2}]$,

$$f = g \quad \Leftrightarrow \quad f^{-1}\{1\} = g^{-1}\{1\} \quad \Leftrightarrow \quad f^{-1}\{0\} = g^{-1}\{0\} \qquad \square$$

3.2.2 Frontiers

As a practical representation for functions, upper and lower sets leave something to be desired. The problem is one of redundancy. For example, it is enough to be told that $X \subseteq \mathbf{2} \times \mathbf{2}$ is lower and contains the element $(1, 1)$, to conclude that X contains $(1, 0)$, $(0, 1)$ and $(0, 0)$, since these all approximate $(1, 1)$. To keep the whole set would be a waste of space.

Luckily, for finite posets we can describe the smallest subset of an upper or lower set which fully determines the whole set. In definition 6 and the following three lemmas, we introduce two operators, **min** and **max**, which can be used to discard redundant information from upper and lower sets respectively. Since these operators are idempotent, a single application is sufficient to produce a compact representation of the set.

Given a poset D, an element $d \in D$ and a subset $X \subseteq D$, we say that *d is minimal in* (resp. *is maximal in*) X if and only if

$$\forall x \in X . x \sqsubseteq d (\text{resp. } x \sqsupseteq d) \quad \Rightarrow \quad x = d$$

Definition 6
Given a poset D, the operations $\mathbf{min}_D$ and $\mathbf{max}_D$ are defined by:

(i) $\mathbf{min}_D(X) = \{x \in X \mid x \text{ is minimal in } X\}$
(ii) $\mathbf{max}_D(X) = \{x \in X \mid x \text{ is maximal in } X\}$.

Duality
$\mathbf{max}_D(X) = \mathbf{min}_{D^{op}}(X)$. $\square$

A subset $X \subseteq D$ is said to be *irredundant* if

$$\forall x, y \in X . x \sqsubseteq y \quad \Rightarrow \quad x = y$$

Lemma 7.
For every poset D and subset $X \subseteq D$, both $\mathbf{min}(X)$ and $\mathbf{max}(X)$ are irredundant.

Proof
Immediate from definitions. $\square$

Lemma 8
For every poset D and irredundant subset $X \subseteq D$,

$$\mathbf{min}(X) = \mathbf{max}(X) = X$$

Proof
Immediate from definitions. $\square$

It is immediate from the preceding two lemmas that **min** and **max** are idempotent.

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

Lemma 9
For poset D, upper and finite subset $X \subseteq D$, lower and finite subset $Y \subseteq D$:

(i) $\uparrow \mathbf{min}(X) = X$
(ii) $\downarrow \mathbf{max}(Y) = Y$

Proof
We prove (i) directly. (ii) follows immediately by duality.
$\subseteq$: Assume $x \in \uparrow \mathbf{min}(X)$. Then, for some $x' \in \mathbf{min}(X)$, $x' \sqsubseteq x$. Thus, for some $x' \in X$, $x' \sqsubseteq x$ since $\mathbf{min}(X) \subseteq X$. Hence $x \in X$, since X is upper.
$\supseteq$: For any $x \in X$, either $x \in \mathbf{min}(X)$ or $\mathbf{min}(X - \{x\})$ and there exists an $x' \in (X - \{x\})$ such that $x' \sqsubseteq x$. It is clear that $\mathbf{min}(\{\mathbf{x}\}) = \{\mathbf{x}\}$; the existence of some $y \in \mathbf{min}(X)$ such that $y \sqsubseteq x$ follows by induction on the size of X, since X is finite. □

Finally, we can define the frontier representation:

Definition 10
For poset D, function $f \in [D \to \mathbf{2}]$:

(i) the *minimum-1-frontier* for f is the set $\mathbf{F\text{-}1}_D(f) = \mathbf{min}_D(f^{-1}\{1\})$
(ii) the *maximum-0-frontier* for f is the set $\mathbf{F\text{-}0}_D(f) = \mathbf{max}_D(f^{-1}\{0\})$

Duality
$\mathbf{F\text{-}0}_D(f) = \mathbf{F\text{-}1}_{D^{op}}(\bar{f})$ □

Now we can complete the definition of the Rep function:

$$\mathrm{Rep}_{D \to \mathbf{2}}(f) = (\mathbf{F\text{-}1}_D(f), \mathbf{F\text{-}0}_D(f)).$$

The redundancy (using both frontiers) allows a more-efficient algorithm for constructing Rep(f), as we shall see below.

Theorem 11 (The representation theorem)
For all D, $D' \in \mathscr{L}$, for all $f, g \in [D \to D']$

$$f = g \quad \Leftrightarrow \quad \mathrm{Rep}_{D \to D'}(f) = \mathrm{Rep}_{D \to D'}(g)$$

Proof
Straightforward induction on the type of f and g, using the fact that uncurry is an isomorphism and that pairing is surjective, together with corollaries 2 and 5. □

Although presented differently, the above definition of a frontier is equivalent to that originally given in Clack and Peyton Jones (1985).

3.2.3 *Using the frontier representations*

We need to be able to implement a number of operations using the frontier representations of a function. Firstly, we need to be able to talk about an argument value's position relative to a frontier:

Definition 12
Given a poset D, an element $d \in D$ and a subset $X \subseteq D$, we say that

(i) d *is above* X *if* $x \sqsubseteq_D d$ for some $x \in X$.
(ii) d is below X if $d \sqsubseteq_D x$ for some $x \in X$.

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

Duality

d is above X in D if and only if d is below X in D^{op}. □

Clearly, x is above X if and only if $x \in \uparrow X$, and x is below X if and only if $x \in \downarrow X$.

We have already seen that using the set $f^{-1}\{1\}$ as a representation for f, to evaluate an application $f(x)$, we must determine whether or not $x \in f^{-1}\{1\}$. Frontiers can be used to evaluate function applications by making use of the following:

Lemma 13

For poset D, upper and finite set $X \subseteq D$, lower and finite set $Y \subseteq D$, $d \in D$,

(i) $d \in X \Leftrightarrow d$ is above $\mathbf{min}(X)$
(ii) $d \in Y \Leftrightarrow d$ is below $\mathbf{max}(Y)$.

Proof

(i) By lemma 9, we have that $\uparrow \mathbf{min}(X) = X$, since X is upper and finite. Hence, $d \in X$ if and only if $d \in \uparrow \mathbf{min}(X)$.
(ii) Similarly, by lemma 9, $\downarrow \mathbf{max}(Y) = Y$, since Y is lower and finite. □

In higher-order abstract interpretation we may have to evaluate meets and joins of functions. The next lemma establishes the basis of an algorithm for this:

Lemma 14

For all posets D, functions $f, g \in [D \to \mathbf{2}]$, the following identities hold:

(i) $(f \sqcup g)^{-1}\{1\} = (f^{-1}\{1\}) \cup (g^{-1}\{1\})$
(ii) $(f \sqcup g)^{-1}\{0\} = (f^{-1}\{0\}) \cap (g^{-1}\{0\})$
(iii) $(f \sqcap g)^{-1}\{1\} = (f^{-1}\{1\}) \cap (g^{-1}\{1\})$
(iv) $(f \sqcap g)^{-1}\{0\} = (f^{-1}\{0\}) \cap (g^{-1}\{0\})$.

Proof

We only prove the first of these. The others can be proved in similar fashion. For any $x \in D$,

$$\begin{aligned} x \in (f \sqcup g)^{-1}\{1\} &\Leftrightarrow (f \sqcup g)(x) = 1 \\ &\Leftrightarrow fx = 1 \quad \text{or} \quad gx = 1 \\ &\Leftrightarrow x \in (f^{-1}\{1\}) \cup (g^{-1}\{1\}) \quad \square \end{aligned}$$

Since closure operators are expensive to evaluate, we want to eliminate their use wherever possible. The next three lemmas give a formal basis for the elimination. Lemma 16 is a technical result which is required in the proof of lemma 17; the reader who is interested only in key results rather than technical detail may omit it.

Lemma 15

For all posets D, subsets $X \subseteq D$:

(i) $\mathbf{min}(\uparrow X) = \mathbf{min}(X)$
(ii) $\mathbf{max}(\downarrow X) = \mathbf{max}(X)$

Proof

We prove (i) directly. (ii) follows by duality.

$\subseteq$: assume $x \in \mathbf{min}(\uparrow X)$, i.e. $x \in \uparrow X$ and x is minimal in $\uparrow X$. Then, $\exists x' \in X . x' \sqsubseteq x$,

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

since $x \in \uparrow X$. Now suppose, for some $x' \in X$ (hence $x' \in \uparrow X$), that $x' \sqsubseteq x$. But then $x' = x$, since x is minimal in $\uparrow X$. Hence $\exists x' \in X . x' \sqsubseteq x$ and $\forall x' \in X . x' \sqsubseteq x \Rightarrow x' = x$. Hence $x \in \mathbf{min}(X)$.

$\supseteq$:assume $x \in \mathbf{min}(X)$, i.e. $x \in X$ and x is minimal in X. Certainly, $x \in \uparrow X$, since $x \in X$. Suppose, for some $x' \in \uparrow X$, that $x' \sqsubseteq x$. Then, for some $x'' \in X$, $x'' \sqsubseteq x' \sqsubseteq x$. But then $x'' = x$, since x is minimal in X. Hence $x' = x$ and so $x \in \mathbf{min}(\uparrow X)$. Hence x is minimal in $\uparrow X$. □

Lemma 16

Let D be a finite poset and $\{A_i\}_{i \in I}$ a family of subsets of D:

(i) $\mathbf{min}(\bigcup_{i \in I} A_i) = \mathbf{min}(\bigcup_{i \in I} \mathbf{min}(A_i))$

(ii) $\mathbf{max}(\bigcup_{i \in I} A_i) = \mathbf{max}(\bigcup_{i \in I} \mathbf{max}(A_i))$

Proof

(i)
$$\begin{aligned}
&\mathbf{min}(\bigcup_{i \in I} A_i) \\
&= \mathbf{min}(\uparrow \bigcup_{i \in I} A_i) \quad \text{lemma 15} \\
&= \mathbf{min}(\bigcup_{i \in I} \uparrow A_i) \\
&= \mathbf{min}(\bigcup_{i \in I} \uparrow \mathbf{min}(\uparrow A_i)) \quad \text{lemma 9} \\
&= \mathbf{min}(\bigcup_{i \in I} \uparrow \mathbf{min}(A_i)) \quad \text{lemma 15} \\
&= \mathbf{min}(\uparrow \bigcup_{i \in I} \mathbf{min}(A_i)) \\
&= \mathbf{min}(\bigcup_{i \in I} \mathbf{min}(A_i)) \quad \text{lemma 15}
\end{aligned}$$

(ii) immediate from (i) by duality. □

The frontiers for the meets and joins of functions can be calculated using the following result:

Lemma 17

For finite lattice D, subsets X, $Y \subseteq D$:

(i) $\mathbf{min}(\uparrow X \cap \uparrow Y) = \mathbf{min}(\{x \sqcup y \mid x \in X, y \in Y\})$

(ii) $\mathbf{max}(\downarrow X \cap \downarrow Y) = \mathbf{max}(\{x \sqcap y \mid x \in X, y \in Y\})$

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

Proof

(i) $\mathbf{min}(\uparrow X \cap \uparrow Y)$

$= \mathbf{min}(\bigcup_{x \in X} \{d \in D \mid d \sqsupseteq x\} \cap \bigcup_{y \in Y} \{d \in D \mid d \sqsupseteq y\})$

$= \mathbf{min}(\bigcup_{x \in X} \bigcup_{y \in Y} \{d \in D \mid d \sqsupseteq x \quad \text{and} \quad d \sqsupseteq y\})$

$= \mathbf{min}(\bigcup_{x \in X} \bigcup_{y \in Y} \mathbf{min}(\{d \in D \mid d \sqsupseteq x \quad \text{and} \quad d \sqsupseteq y\}))$ lemma 16

$= \mathbf{min}(\bigcup_{x \in X} \bigcup_{y \in Y} \{x \sqcup y\})$

(ii) immediate from (i) by duality.

4 An algorithm for finding frontiers

In this section we present an algorithm for finding the frontiers of functions in function spaces of the form $[A \to \mathbf{2}]$, where A is a finite lattice. We begin with an algorithm to find the inverse image of an upper set under a monotone function. This is then adapted to give an algorithm which finds frontiers.

$X := A$;
$Y := A$;
while $(X \cap Y) \neq \varnothing$
  choose x **from** $X \cap Y$;
  if $fx \in O$
    then $Y := Y \cap C(\uparrow x)$
    else $X := X \cap C(\downarrow x)$
endwhile

Fig. 1. Finding the inverse image of an upper set.

4.1 Finding the inverse image of an upper set

For finite posets A and B, fig. 1 shows an algorithm for determining the inverse image of an upper set, $O \subseteq B$, under a monotone function, $f \in [A \to B]$.

The algorithm conducts a search of A to find both the inverse image of O under f (the final value of X) and its complement (the final value of Y). At each iteration, the set $X \cap Y$ is the remaining search space.

It is easy to show that the **while** loop of this algorithm has the following invariant and variant:

(Inv) $X \supseteq f^{-1}O \quad \text{and} \quad Y \supseteq C(f^{-1}O)$
(Var) $|X \cap Y|$

The initial assignments clearly establish **Inv**.

On termination $(X \cap Y) = \varnothing$, which together with **Inv** implies that $X = f^{-1}O$ and $Y = C(f^{-1}O)$.

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

4.2 A frontiers algorithm

We now consider how to implement the algorithm of fig. 1 using frontiers to represent the upper and lower sets.

As a first attempt we can simply use the operations **max** and **min** at each step of the algorithm to make the sets X and Y irredundant. Lemma 9 can be used to show that the resulting algorithm (fig. 2) calculates $\mathbf{min}(f^{-1}O)$ and $\mathbf{max}(C(f^{-1}O))$.

For $f \in [A \to \mathbf{2}]$, the new algorithm can be used to calculate $\mathbf{F\text{-}1}(f)$ and $\mathbf{F\text{-}0}(f)$ by choosing $O = \{1\}$. On termination, X is then $\mathbf{F\text{-}1}(f)$ and Y is $\mathbf{F\text{-}0}(f)$.

We must now show how to avoid the use of the closure operations $\uparrow$ and $\downarrow$ (since their use rather defeats the purpose of working with frontiers).

$X := \mathbf{min}(A)$;
$Y := \mathbf{max}(A)$;
while $(\uparrow X \cap \downarrow Y) \neq \varnothing$
  choose x **from** $\uparrow X \cap \downarrow Y$;
  if $fx \in O$
    then $Y := \mathbf{max}(\downarrow Y \cap C(\uparrow x))$
    else $X := \mathbf{min}(\uparrow X \cap C(\downarrow x))$
endwhile

Fig. 2. A naive frontiers algorithm.

4.2.1 Avoiding the use of $\uparrow$ and $\downarrow$

For any poset D, subsets $X, Y \subseteq D$,

$$(\uparrow X \cap \downarrow Y) \neq \varnothing \quad \Leftrightarrow \quad \exists x \in X, y \in Y . x \sqsubseteq y$$

This observation motivates the following:

Definition 18
For subsets $X, Y \subseteq A$, $\mathbf{edges}(X, Y)$ is the set

$$\{x \in X \mid x \text{ is below } Y\} \cup \{y \in Y \mid y \text{ is above } X\} \quad \square$$

We can then replace the test of the **while** loop by the test

$$\mathbf{edges}(X, Y) \neq \varnothing$$

Furthermore, since $\mathbf{edges}(X, Y) \subseteq (\uparrow X \cap \downarrow Y)$, we can replace the first command in the body of the **while** loop by

$$\mathbf{choose}\ x\ \mathbf{from}\ \mathbf{edges}(X, Y)$$

The question of which element is chosen from the set $\mathbf{edges}(X, Y)$ can have a significant impact on the efficiency of the algorithm. We refer the reader to Clack and Peyton Jones (1985) and Martin (1989) for a discussion of this point.

It remains to calculate the sets for the assignments within the **while** loop. We will concentrate on the case that $f \in [D \to \mathbf{2}]$ and $O = \{1\}$.

Let us consider the first assignment:

$$Y := \mathbf{max}(\downarrow Y \cap C(\uparrow x))$$

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

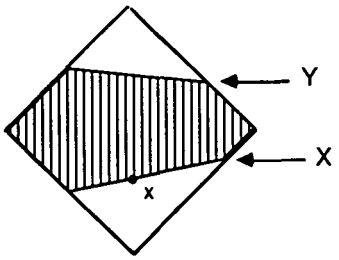

Fig. 3

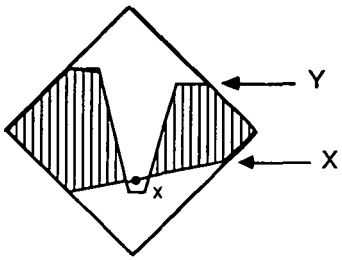

Fig. 4

This assignment is executed when we know that $f(x) = 1$. Recall that $\downarrow Y$ is a superset all those points, x, such that $f(x) \neq 1$, and that at this stage $\downarrow Y$ also contains x (fig. 3). By monotonicity, all of the elements above x (i.e. $\uparrow x$) will also map to 1; thus $C(\uparrow x)$ is the set of points that the value $f(x)$ tells us nothing about. Intersecting $C(\uparrow x)$ with $\downarrow Y$ gives a set which is strictly smaller than $\downarrow Y$ while still being a superset of $f^{-1}\{1\}$, and the outer application of **max** gives the new approximation to the maximum-0-frontier being searched for (fig. 4).

To optimize this procedure, we will find the minimal points which are not below x. We can then define an operation to 'remove' just those points from the (approximate) maximum-0-frontier, Y, without having to calculate $\downarrow Y$. The second assignment is dual to the first and concerns the case when $f(x) = 0$. For this case we need the maximal points which are not above x:

Definition 19

For a poset D, the operations $\mathbf{succs}_D$ and $\mathbf{preds}_D$ are defined as follows, for $x \in D$:

(i) $\mathbf{succs}_D(x) = \mathbf{min}_D(C_D(\downarrow_D x))$

(ii) $\mathbf{preds}_D(x) = \mathbf{max}_D(C_D(\uparrow_D x))$

Duality

$\mathbf{preds}_D(x) = \mathbf{succs}_{D^{op}}(x)$ □

The names **succs** and **preds** may seem a little strange in light of their definitions. The reason for this is that in Martin and Hankin (1987) the corresponding functions were (incorrectly) defined to return the immediate successors/predecessors of a point in a lattice. Note that for chains the definitions are equivalent. The mistake was spotted by Chris Martin and is corrected in his thesis (Martin, 1989).

Using **succs** and **preds**, we can avoid the use of the first closure operator on the right-hand-sides of the two assignments:

Theorem 20

For finite lattice D, subset $Y \subseteq D$ and element $x \in D$:

(i) $\mathbf{min}(\uparrow Y \cap C(\downarrow x)) = \mathbf{min}\{y \sqcup z \mid y \in Y, z \in \mathbf{succs}(x)\}$

(ii) $\mathbf{max}(\downarrow Y \cap C(\uparrow x)) = \mathbf{max}\{y \sqcap z \mid y \in Y, z \in \mathbf{preds}(x)\}$

Proof

By lemma 9 $C(\downarrow x) = \uparrow \mathbf{min}(C(\downarrow x))$ and $C(\uparrow x) = \downarrow \mathbf{max}(C(\uparrow x))$, since $C(\downarrow x)$ upper and $C(\uparrow x)$ lower. Then the result is immediate by lemma 17. □

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

$X := \{\bot_A\}$;
$Y := \{\top_A\}$;
while edges $(X, Y) = \varnothing$
  choose x **from edges** (X, Y);
  if $fx \in O$
    then $Y := Y \cap^{max} \mathbf{preds}(x)$
    else $X := X \cap^{min} \mathbf{succs}(x)$
endwhile

Fig. 5. The final frontiers algorithm.

Remark
There is some room for optimization in the application of theorem 20. For example, if we partition Y into Y_1 and Y_2, where

$$Y_1 = \{y \in Y \mid y \not\sqsubseteq x\} \quad \text{and} \quad Y_2 = \{y \in Y \mid y \sqsubseteq x\}$$

then it can be shown that

$$\mathbf{min}(\uparrow Y \cap C(\downarrow x)) = \mathbf{min}(Y_1 \cup (\uparrow Y_2 \cap C(\downarrow x)))$$

allowing us to reduce the number of joins to be calculated. □

To make use of lemma 9 and theorem 20, we define the operations $\cap^{min}$ and $\cap^{max}$, for $X, Y \subseteq A$, by

$$X \cap^{min} Y = \mathbf{min}(\{x \sqcup y \mid x \in X, y \in Y\})$$

and

$$X \cap^{max} Y = \mathbf{max}(\{x \sqcap y \mid x \in X, y \in Y\}).$$

Lastly, we note that when A is a lattice

$$\mathbf{min}(A) = \{\bot_A\} \quad \text{and} \quad \mathbf{max}(A) = \{\top_A\}$$

The final version of the algorithm is shown in fig. 5.

Before we show how to implement the operations **preds** and **succs** for lattices in $\mathscr{L}$, we must address the question of how the elements of the frontier sets are to be represented.

4.2.2 Representing frontier elements

In section 3 we showed how a function in any of the function spaces in $\mathscr{L}$ can be factored into a collection of functions in spaces of the form $[D \to \mathbf{2}]$. Such functions can then be represented by frontiers, which are just sets of elements of D. However, we have not yet addressed the question of how these elements should themselves be represented. Clearly, when D is **2**, the answer is straightforward enough. It is also clear how elements of product spaces and lifted lattices may be represented, given representations for elements of their component lattices. Thus the problem is again one of representing functions.

We will assume that the functions which occur as elements of frontiers are

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

themselves represented by factorization and the use of frontiers. We shall see that, as in Martin and Hankin (1987), both the minimum-1-frontier *and* the maximum-0-frontier are needed by the implementation of **preds** and **succs**.

4.2.3 Implementing **preds** *and* **succs**

We need to calculate **succs** (x) and **preds** (x), $x \in A$, for each $A \in \mathscr{L}$. We consider each of the cases in the definition of $\mathscr{L}$ in turn. The detailed proofs of the lemmas can be found in appendix B.

(i) $A = \mathbf{2}$.

This case is simple since

$$\begin{aligned} \mathbf{succs}(1) &= \mathbf{min}(C(\downarrow 1)) = \mathbf{min}(\varnothing) = \varnothing \\ \mathbf{succs}(0) &= \mathbf{min}(C(\downarrow 0)) = \mathbf{min}(\{1\}) = \{1\} \end{aligned}$$

Similarly,

$$\mathbf{preds}(0) = \varnothing \quad \text{and} \quad \mathbf{preds}(1) = \{0\}$$

(ii) $A = D_1 \times \ldots \times D_n$.

$$\begin{aligned} &\text{Let } S_i(y) = (\bot_{D_1}, \ldots, \bot_{D_{i-1}}, y, \bot_{D_{i+1}}, \ldots, \bot_{D_n}). \\ &\text{Let } P_i(y) = (\top_{D_1}, \ldots, \top_{D_{i-1}}, y, \top_{D_{i+1}}, \ldots, \top_{D_n}). \end{aligned}$$

Lemma 21

For $x_i \in D_i$, $1 \leqslant i \leqslant n$:

(i) $\mathbf{succs}(x_1, \ldots, x_n) = \bigcup_{i=1}^{n} \{S_i(d) \mid d \in \mathbf{succs}(x_i)\}$

(ii) $\mathbf{preds}(x_1, \ldots, x_n) = \bigcup_{i=1}^{n} \{P_i(d) \mid d \in \mathbf{preds}(x_i)\}$ □

(iii) $A = D_\bot$.

$$\begin{aligned} \mathbf{succs}(\bot) &= \{\mathit{lift}(\bot_D)\} \\ \mathbf{succs}(\mathit{lift}(d)) &= \{\mathit{lift}(d') \mid d' \in \mathbf{succs}(d)\} \\ \mathbf{preds}(\bot) &= \varnothing \\ \mathbf{preds}(\mathit{lift}(\bot_D)) &= \{\bot\} \\ \mathbf{preds}(\mathit{lift}(d)) &= \{\mathit{lift}(d') \mid d' \in \mathbf{preds}(d)\} \quad d \neq \bot_D \end{aligned}$$

(iv) $A = [D_1 \to \ldots \to D_k \to D']$, $k > 1$, where D' is not a function space.

We assume each $f \in A$ to be represented by its uncurried version, $\mathrm{uncurry}_k(f)$. Then, since $\mathrm{uncurry}_k$ is an isomorphism:

$$\mathrm{uncurry}_k(\mathbf{preds}_A(f)) = \mathbf{preds}_{D_1 \times \ldots \times D_k \to D'}(\mathrm{uncurry}_k(f))$$

Similarly for $\mathbf{succs}_A(f)$.

(v) $A = [D \to D'_1 \times \ldots \times D'_n]$.

We assume each $f \in A$ to be represented by the tuple $(f_1, \ldots, f_n)$, where $f_i = \pi_i \circ f$.

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

Then, since the map $f \mapsto (f_1, \ldots, f_n)$ is an isomorphism from $[D \to D'_1 \times \ldots \times D'_n]$ to $[D \to D'_1] \times \ldots \times [D \to D'_n]$,

$$\{(h_1, \ldots, h_n) \mid h \in \mathbf{preds}_A(f)\} = \mathbf{preds}_{[D \to D'_1] \times \ldots \times [D \to D'_n]}(f_1, \ldots, f_n)$$

(vi) $A = [D \to \mathbf{2}]$.

Given the minimum-1-frontier and maximum-0-frontier for $f \in [D \to \mathbf{2}]$, we show how to calculate $\mathbf{preds}(f)$ and $\mathbf{succs}(f)$ and the frontiers for the elements of these sets.

Lemma 22

For finite poset D, function $f \in [D \to \mathbf{2}]$:

(i) $\mathbf{succs}_{[D \to \mathbf{2}]}(f) = \mathbf{min}_{[D \to \mathbf{2}]}(C_{[D \to \mathbf{2}]}(\downarrow_{[D \to \mathbf{2}]} f)) = \{\chi_{\uparrow x} \mid x \in \mathbf{F\text{-}0}_D(f)\}$

(ii) $\mathbf{preds}_{[D \to \mathbf{2}]}(f) = \mathbf{max}_{[D \to \mathbf{2}]}(C_{[D \to \mathbf{2}]}(\uparrow_{[D \to \mathbf{2}]} f)) = \{\chi_{C(\downarrow x)} \mid x \in \mathbf{F\text{-}1}_D(f)\}$ □

Note that to evaluate $\mathbf{succs}(f)$ we need the maximum-0-frontier of f, whereas for $\mathbf{preds}(f)$ we need the minimum-1-frontier. Thus, in general we need both frontiers of a function. We can adapt lemma 22 to give both kinds of frontier for each member of $\mathbf{succs}(f)$ and $\mathbf{preds}(f)$.

Frontiers for the members of $\mathbf{succs}(f)$ are given by:

$$\begin{aligned}
&\{\mathbf{F\text{-}1}(g) \mid g \in \mathbf{succs}(f)\} \\
&= \{\mathbf{F\text{-}1}(\chi_{\uparrow x}) \mid x \in \mathbf{F\text{-}0}(f)\} \\
&= \{\mathbf{min}(\chi^{-1}_{\uparrow x})\{1\} \mid x \in \mathbf{F\text{-}0}(f)\} \\
&= \{\mathbf{min}(\uparrow x) \mid x \in \mathbf{F\text{-}0}(f)\} \\
&= \{\{x\} \mid x \in \mathbf{F\text{-}0}(f)\}
\end{aligned}$$

and:

$$\begin{aligned}
&\{\mathbf{F\text{-}0}(g) \mid g \in \mathbf{succs}(f)\} \\
&= \{\mathbf{F\text{-}0}(\chi_{\uparrow x}) \mid x \in \mathbf{F\text{-}0}(f)\} \\
&= \{\mathbf{max}(\chi^{-1}_{\uparrow x})\{0\} \mid x \in \mathbf{F\text{-}0}(f)\} \\
&= \{\mathbf{max}(C(\uparrow x)) \mid x \in \mathbf{F\text{-}0}(f)\} \\
&= \{\mathbf{preds}(x) \mid x \in \mathbf{F\text{-}0}(f)\}
\end{aligned}$$

Similarly, frontiers for the members of $\mathbf{preds}(f)$ are given by:

$$\{\mathbf{F\text{-}1}(g) \mid g \in \mathbf{preds}(f)\} = \{\mathbf{succs}(x) \mid x \in \mathbf{F\text{-}1}(f)\}$$

and:

$$\{\mathbf{F\text{-}0}(g) \mid g \in \mathbf{preds}(f)\} = \{\{x\} \mid x \in \mathbf{F\text{-}1}(f)\}$$

(vii) $A = [D \to D'_\perp]$.

For $f \in [D \to \mathbf{2}]$, $D' \in \mathscr{L}$, the *least* and *greatest embeddings* of f are defined to be $lemb_{D'}(f) \in [D \to D_\perp]$, where

$$lemb_{D'}(f)(x) = \begin{cases} \perp & \text{if } fx = 0 \\ lift(\perp_{D'}) & \text{otherwise} \end{cases}$$

and $gemb_{D'}(f) \in [D \to D'_\perp]$, where

$$gemb_{D'}(f)(x) = \begin{cases} \perp & \text{if } fx = 0 \\ lift(\top_{D'}) & \text{otherwise} \end{cases}$$

respectively.

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

For $f \in [D \to D']$, the *least* and *greatest liftings* of f are defined to be $llift(f) \in [D \to D'_\perp]$, where

$$llift(f)(x) = \begin{cases} \perp & \text{if } fx = \perp_{D'} \\ lift(f(x)) & \text{otherwise} \end{cases}$$

and $glift(f) \in [D \to D'_\perp]$, where

$$glift(f)(x) = lift(f(x))$$

respectively.

Lemma 23
For all $f \in A = [D \to D'_\perp]$, $\mathbf{succs}(f) = \mathbf{min}(S_1 \cup S_2)$, where

$$S_1 = \{lemb_{D'}(h) \mid h \in \mathbf{succs}(L_f)\}$$
$$S_2 = \{llift(h) \mid h \in \mathbf{succs}(H_f)\} \quad \square$$

Lemma 24
For all $f \in A = [D \to D'_\perp]$, $\mathbf{preds}(f) = \mathbf{max}(P_1 \cup P_2)$, where

$$P_1 = \{gemb_{D'}(h) \mid h \in \mathbf{preds}(L_f)\}$$
$$P_2 = \{glift(h) \mid h \in \mathbf{preds}(H_f)\}$$

Proof
Similar to that for lemma 23 (see appendix B). $\square$

4.2.4 Changing the initial values in the frontiers algorithm

It was pointed out in the original presentation of the frontiers algorithm (Clack and Peyton Jones, 1985), that the frontier sets for one approximation to a function can be used to reduce the size of the search space when finding the frontier sets of the next approximation.

In terms of our presentation, we can explain this by reference to lemma 3. If f_i and f_{i+1} are successive approximations to the value of $f \in [A \to \mathbf{2}]$, then $f_i \sqsubseteq f_{i+1}$ and so $f_i^{-1}\{0\} \supseteq f_{i+1}^{-1}\{0\}$. Recalling the loop invariant (**Inv**), this allows us to initialize Y to $\mathbf{F\text{-}0}(f_i)$ rather than $\{\top_A\}$ when calculating the frontiers for f_{i+1}.

In the same way, if we were given the set $\mathbf{F\text{-}1}(g)$ for some function $g \supseteq f$, we could initialize X to $\mathbf{F\text{-}1}(g)$ rather than $\{\perp_A\}$ when establishing the frontiers for *each* of the f_i.

4.3 Comparison with previous formulations of the frontiers algorithm

Despite substantial differences in presentation, the algorithm described above is essentially a generalization of the algorithm for first-order functions described in Clack and Peyton Jones (1985). For example, when D is of the form $\mathbf{2}^n$, theorem 20 corresponds exactly to the 'shine down' operation.

The fact that frontiers represent upper sets and lower sets is a new insight which has allowed us to present the extension to the higher-order case in a manner which is more

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

straightforward than previous formulations. The ideas behind this insight stem directly from the approach taken in Martin and Hankin (1987) and Martin (1989). Of particular importance was Martin's realization (1989) that the correct generalization of **succs**(x) to non-chain domains should be in terms of the set $\{y \in D \mid y \not\sqsubseteq x\}$ rather than $\{y \in D \mid y] x\}$ (as was suggested in Martin and Hankin, 1987).

One significant difference between our algorithm and the above mentioned versions, is that we do not need to take any special measures, such as those taken in Peyton Jones and Clack (1987) and Martin and Hankin (1987), to ensure that a function is evaluated at most once at each point in its argument lattice.

5 Frontiers are not enough

In this section we argue that despite the benefits gained from the use of frontiers it will often be necessary to reduce the size of the abstract domains before attempting to find the fixed point of a function. We provide a method of doing this without having to change the original abstract interpretation. In general this will entail settling for imprecise but safe approximations to the actual fixed point of a function.

5.1 A problem of complexity

For many of the functions which arise in abstract interpretation, establishing the graph of a function, using *any* method, will be intractable. To see why this is so, consider the familiar function:

$$\begin{aligned} &foldr : (\alpha \to \beta \to \beta) \to list\ \alpha \to \beta \to \beta \\ &foldr\ f\ [\,]\ b = b \\ &foldr\ f\ (a : as)\ b = f\ a\ (foldr\ f\ as\ b) \end{aligned}$$

Suppose we wish to analyse programs using *foldr* for strictness in the style of Burn, Hankin and Abramsky (1986) using Wadler's domains for lists (1987). When *foldr* is used at simple types (e.g. $\alpha = \beta = int$) this is straightforward. However, in many quite ordinary programs, *foldr* is used at more complex types. For example, in the definition of the catenate function

$$\begin{aligned} &cat : list(list\ char) \to list\ char \\ &cat\ l = foldr\ append\ l\ [\,] \end{aligned}$$

foldr is used at a type instance with $\alpha = \beta = list\ char$.

The interpretation used for *char* is **2**. This induces interpretations of $(\mathbf{2}_\perp)_\perp$ (written **4**) for *list char* and $(\mathbf{4}_\perp)_\perp$ (written **6**) for *list*(*list char*). The abstract function for *foldr* at this type would thus be an element of $[\mathbf{[4 \to 4 \to 4] \to 6 \to 4 \to 4}]$.

Even taking monotonicity into account, it is not hard to show that the argument domain alone for (an uncurried version of) this function contains of the order of 10^6 elements. Clearly, if we have to evaluate any one of the approximations to *foldr* at a significant proportion of these elements to establish its value, the operation of finding a fixed point will be too costly to be considered in any practical compiler. (The use of a polymorphic function in this example raises the question of whether it is *necessary*

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

to calculate the values of polymorphic functions at their higher type instances. A detailed discussion of this topic is beyond the scope of this article (see Abramsky, 1986; Hughes, 1988). We could just as well have used a monomorphic example to illustrate the underlying complexity problem.)

The kinds of function which are 'well behaved' with respect to the frontiers algorithm are described in Peyton Jones and Clack (1987); because the frontiers algorithm searches the argument lattice from the top and bottom, working towards the middle, well-behaved functions are those which have frontiers whose elements are either very low down or very high up the argument lattice. For such functions the frontier sets are small and the frontiers algorithm will find them with little effort. On the other hand, badly behaved functions have frontier sets consisting of elements from the middle of the lattice and in the worst case the frontiers algorithm will evaluate the function at every point in the lattice before finding the frontier sets.

Experience with an implementation of a strictness analyser employing the frontiers algorithm suggests that higher-order functions are often badly behaved (as is certainly the case for *foldr*). One reason for this may be that many higher-order functions apply some of their arguments to others and thus behave as more or less exotic variants of the *apply* function. The problem with *apply* is that the value of (*apply f x*) will be high up in the result lattice if *either f or x* are high in their respective lattices.

5.2 Reducing the size of a lattice

Our solution to the complexity problem cited above is to work in a smaller lattice to establish bounds on the required fixed point. We use maps, reminiscent of the *abs* and *conc* maps used in abstract interpretation (see, for example, Burn, Hankin and Abramsky, 1986), to move between larger and smaller lattices. First we must formalize the notion of one lattice being smaller than another.

Definition 25

The *abstraction ordering*, $\preccurlyeq$, on members of $\mathscr{L}$ is as follows:

$$\begin{array}{rll} \mathbf{2} \preccurlyeq D & & \\ D_1 \times \ldots \times D_n \preccurlyeq D'_1 \times \ldots \times D'_n & \text{if} & D_i \preccurlyeq D'_i, 1 \leqslant i \leqslant n \\ D_\perp \preccurlyeq D'_\perp & \text{if} & D \preccurlyeq D' \\ [A \to B) \preccurlyeq [A' \to B'] & \text{if} & A \preccurlyeq A' \quad \text{and} \quad B \preccurlyeq B' \end{array}$$

If $A \preccurlyeq B$, we say that *A is an abstraction of B*. □

We next define two families of abstraction and concretization maps relating elements of members of $\mathscr{L}$: the 'safe' maps, which give overestimates of values, thus allowing us to derive upper bounds on fixed points, and their 'live' counterparts, which give underestimates and allow us to derive lower bounds.

The concepts of safety and liveness are borrowed from the abstract interpretation literature (e.g. Mycroft, 1981). Strictness analysis is a good example of a safe analysis; the overestimation means that we sometimes fail to infer that a function is strict when it is strict (1 is used to represent *possible* termination). Mycroft's termination analysis is a good example of a live analysis; the underestimation means that we sometimes

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

fail to infer that a function terminates when it does (0 is used to represent *possible* non-termination).

Definition 26
For each $D, D' \in \mathscr{L}$, such that $D' \preccurlyeq D$, the *safe* abstraction and concretization maps

$$\mathbf{Abs}^s_{D,D'} \in [D \to D'] \quad \text{and} \quad \mathbf{Conc}^s_{D',D} \in [D' \to D]$$

are defined by:

$$\mathbf{Abs}^s_{D,2}\, x = \begin{cases} 0 & \text{if } x = \bot_D \\ 1 & \text{otherwise} \end{cases}$$

$$\mathbf{Conc}^s_{2,D}\, x = \begin{cases} \bot_D & \text{if } x = 0 \\ \top_D & \text{if } x = 1 \end{cases}$$

$$\mathbf{Abs}^s_{D_\bot, D'_\bot}\, x = \begin{cases} \bot & \text{if } x = \bot \\ \mathbf{Abs}^s_{D,D'}\, d & \text{if } x = \mathit{lift}\, d, \text{ for some } d \in D \end{cases}$$

$$\mathbf{Conc}^s_{D'_\bot, D_\bot}\, x = \begin{cases} \bot & \text{if } x = \bot \\ \mathbf{Conc}^s_{D',D}\, d & \text{if } x = \mathit{lift}\, d, \text{ for some } d \in D' \end{cases}$$

$$\mathbf{Abs}^s_{[A \to B],[A \to B']} f = \mathbf{Abs}^s_{B,B'} \circ f \circ \mathbf{Conc}^s_{A',A}$$

$$\mathbf{Conc}^s_{[A' \to B'],[A \to B]} f = \mathbf{Conc}^s_{B',B} \circ f \circ \mathbf{Abs}^s_{A,A'}$$

For $D = [D_1 \times \ldots \times D_n]$ and $D' = [D'_1 \times \ldots \times D'_n]$,

$$\mathbf{Abs}^s_{D,D'}(x_1, \ldots, x_n) = (x'_1, \ldots, x'_n) \quad \text{where } x'_i = \mathbf{Abs}^s_{D_i, D'_i} x_i,\ 1 \leqslant i \leqslant n$$
$$\mathbf{Conc}^s_{D',D}(x_1, \ldots, x_n) = (x'_1, \ldots, x'_n) \quad \text{where } x'_i = \mathbf{Conc}^s_{D'_i, D_i} x_i,\ 1 \leqslant i \leqslant n. \quad \square$$

Definition 27
The definitions of the *live* abstraction and concretization maps are given by substituting $\mathbf{Abs}^l$ for $\mathbf{Abs}^s$ and $\mathbf{Conc}^l$ for $\mathbf{Conc}^s$ everywhere in definition 26, except for the base case for $\mathbf{Abs}^l$, which is

$$\mathbf{Abs}^l_{D,2} x = \begin{cases} 1 & \text{if } x = \top_D \\ 0 & \text{otherwise} \end{cases} \quad \square$$

The following lemma states the *exact adjointness* property (Mycroft, 1981) of the **Abs** and **Conc** maps.

Lemma 28
For all $A, A' \in \mathscr{L}$ such that $A' \preccurlyeq A$:

(i) $\mathbf{Abs}^l_{A,A'} \circ \mathbf{Conc}^l_{A',A} = \mathbf{id}_{A'} = \mathbf{Abs}^s_{A,A'} \circ \mathbf{Conc}^s_{A',A}$
(ii) $\mathbf{Conc}^l_{A',A} \circ \mathbf{Abs}^l_{A,A'} \sqsubseteq \mathbf{id}_A \sqsubseteq \mathbf{Conc}^s_{A',A} \circ \mathbf{Abs}^s_{A,A'}$

Proof
Straightforward induction on A'. $\square$

Corollary 29
For all $D, D' \in \mathscr{L}$ such that $D' \preccurlyeq D$:

(i) $\mathbf{Conc}^s_{D',D}$ and $\mathbf{Conc}l_{D',D}$ are injective.
(ii) $\mathbf{Abs}^s_{D,D'}$ and $\mathbf{Abs}^l_{D,D'}$ are onto.
(iii) $\mathbf{Abs}^s_{D,D'}$ and $\mathbf{Abs}^l_{D,D'}$ are strict.

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

Lemma 30
For all lattices A, $A' \in \mathscr{L}$ such that $A' \preccurlyeq A$,

(i) $\mathbf{fix}_{A'} = \mathbf{Abs}^{s}_{[[A \to A] \to A], [[A' \to A'] \to A']}(\mathbf{fix}_A)$
(ii) $\mathbf{fix}_{A'} = \mathbf{Abs}^{l}_{[[A \to A] \to A], [[A' \to A'] \to A']}(\mathbf{fix}_A)$.

Proof
(i) Let $D = [A \to A]$ and $D' = [A' \to A']$.
A routine induction on i suffices to show that for all i, for all $f \in D'$,

$$\mathbf{Abs}^{s}_{A, A'}((\mathbf{Conc}^{s}_{D', D} f)^i \perp_A) = f^i \perp_{A'}$$

Then, for any $f \in [A' \to A']$,

$$\begin{aligned}
&(\mathbf{Abs}^{s}_{[[A \to A] \to A], [[A' \to A'] \to A']} \mathbf{fix}_A) f \\
&= \mathbf{Abs}^{s}_{A, A'}(\mathbf{fix}_A(\mathbf{Conc}^{s}_{D', D} f)) \\
&= \mathbf{Abs}^{s}_{A, A'}\left(\bigsqcup_{i=0}^{\infty} (\mathbf{Conc}^{s}_{D', D} f)^i \perp_A\right) \\
&= \bigsqcup_{i=0}^{\infty} \mathbf{Abs}^{s}_{A, A'}((\mathbf{Conc}^{s}_{D', D} f)^i \perp_A), \quad \text{since } \mathbf{Abs}^{s}_{A, A'} \text{ monotone, } A \text{ finite} \\
&= \bigsqcup_{i=0}^{\infty} f^i \perp_{A'} \\
&= \mathbf{fix}_{A'} f
\end{aligned}$$

(ii) The proof for (i) goes through identically, substituting $\mathbf{Abs}^{l}$ for $\mathbf{Abs}^{s}$ and $\mathbf{Conc}^{l}$ for $\mathbf{Conc}^{s}$. □

Theorem 31
For all lattices A, $A' \in \mathscr{L}$ such that $A' \preccurlyeq A$, for all $f \in [A \to A]$.

(i) $\mathbf{Conc}^{s}_{A', A}(\mathbf{fix}_{A'}(\mathbf{Abs}^{s}_{[A \to A], [A' \to A']} f)) \sqsupseteq \mathbf{fix}_A f$
(ii) $\mathbf{Conc}^{l}_{A', A}(\mathbf{fix}_{A'}(\mathbf{Abs}^{l}_{[A \to A], [A' \to A']} f)) \sqsubseteq \mathbf{fix}_A f$

Proof

(i)
$$\begin{aligned}
&\mathbf{Conc}^{s}_{A', A} \circ \mathbf{fix}_{A'} \circ \mathbf{Abs}^{s}_{[A \to A], [A' \to A']} \\
&= \mathbf{Conc}^{s}_{A', A} \circ (\mathbf{Abs}^{s}_{[[A \to A] \to A], [[A' \to A'] \to A']} \mathbf{fix}_A) \circ \mathbf{Abs}^{s}_{[A \to A], [A' \to A']} && \text{lemma 30} \\
&= \mathbf{Conc}^{s}_{A', A} \circ \mathbf{Abs}^{s}_{A, A'} \circ \mathbf{fix}_A \circ \mathbf{Conc}^{s}_{[A' \to A'], [A \to A]} \circ \mathbf{Abs}^{s}_{[A \to A], [A' \to A']} && \text{def. 26} \\
&\sqsupseteq \mathbf{fix}_A && \text{lemma 28}
\end{aligned}$$

(ii)
$$\begin{aligned}
&\mathbf{Conc}^{l}_{A', A} \circ \mathbf{fix}_{A'} \circ \mathbf{Abs}^{l}_{[A \to A], [A' \to A']} \\
&= \mathbf{Conc}^{l}_{A', A} \circ (\mathbf{Abs}^{l}_{[[A \to A] \to A], [[A' \to A'] \to A']} \mathbf{fix}_A) \circ \mathbf{Abs}^{l}_{[A \to A], [A' \to A']} && \text{lemma 30} \\
&= \mathbf{Conc}^{l}_{A', A} \circ \mathbf{Abs}^{l}_{A, A'} \circ \mathbf{fix}_A \circ \mathbf{Conc}^{l}_{[A' \to A'], [A \to A]} \circ \mathbf{Abs}^{l}_{[A \to A], [A' \to A']} && \text{def. 26} \\
&\sqsubseteq \mathbf{fix}_A. \quad \square && \text{lemma 28}
\end{aligned}$$

Theorem 31 tells us that we can find safe and live approximations to the value of a function defined as the fixed point of a functional over some member of $\mathscr{L}$, by abstracting the functional to a smaller member of $\mathscr{L}$ and finding the fixed point there.

5.3. *Applying* Conc *to a frontier*

Given a functional $G \in [A \to A]$, we obtain upper and lower bounds on the value of $\mathbf{fix}_A G$ by evaluating

$$\mathbf{Conc}^{s}_{A', A}(\mathbf{fix}_{A'}(\mathbf{Abs}^{s}_{A, A'} \circ G \circ \mathbf{Conc}^{s}_{A', A}))$$

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

and

$$\mathbf{Conc}^{l}_{A',A}(\mathbf{fix}_{A'}(\mathbf{Abs}^{l}_{A,A'} \circ G \circ \mathbf{Conc}^{l}_{A',A})).$$

(Note that this is not an abstract *interpretation* of a term in some language, but the application of an abstraction *map* to the denotation of a term under such an interpretation.) In general, this will involve applying **Conc** maps to the frontier representations of functional values. We can evaluate such applications using the following result:

Lemma 32
For $D, D' \in \mathscr{L}$ such that $D' \preccurlyeq D$, for $f \in [D' \to \mathbf{2}]$,

$$(f \circ \mathbf{Abs}^{s}_{D,D'})^{-1}\{0\} = \downarrow\{\mathbf{Conc}^{s}_{D',D}\, x \mid x \in f^{-1}\{0\}\}$$

Proof
$\subseteq$: assume $y \in (f \circ \mathbf{Abs}^{s}_{D,D'})^{-1}\{0\}$, i.e. $y \in D$ and $f(\mathbf{Abs}^{s}_{D,D'} y) = 0$.
Then $(\mathbf{Conc}^{s}_{D',D}(\mathbf{Abs}^{s}_{D,D'} y)) \in \{\mathbf{Conc}^{s}_{D',D}\, x \mid x \in f^{-1}\{0\}\}$.
By lemma 28, $y \sqsubseteq \mathbf{Conc}^{s}_{D',D}(\mathbf{Abs}^{s}_{D,D'} y)$.
Hence $y \in \downarrow\{\mathbf{Conc}^{s}_{D',D}\, x \mid x \in f^{-1}\{0\}\}$.
$\supseteq$: assume $y \in \downarrow\{\mathbf{Conc}^{s}_{D',D}\, x \mid x \in f^{-1}\{0\}\}$.
Then $y \sqsubseteq \mathbf{Conc}^{s}_{D',D}\, x$, for some $x \in f^{-1}\{0\}$.
Thus, $f(\mathbf{Abs}^{s}_{D,D'} y) \sqsubseteq f(\mathbf{Abs}^{s}_{D,D'}(\mathbf{Conc}^{s}_{D',D}\, x)) = fx$ (by lemma 28).
Hence $y \in (f \circ \mathbf{Abs}^{s}_{D,D'})^{-1}\{0\}$, since $fx = 0$. □

From this it is easy to show that

$$\mathbf{F\text{-}0}(\mathbf{Conc}^{s}_{[D'-\mathbf{2}],[D-\mathbf{2}]} f) = \{\mathbf{Conc}^{s}_{D',D}\, x \mid x \in \mathbf{F\text{-}0}(f)\}$$

since the **Conc** maps are injective. To obtain $\mathbf{F\text{-}1}(\mathbf{Conc}^{s}_{[D'\to\mathbf{2}],[D\to\mathbf{2}]} f)$ we must calculate the minimum-1-frontier for the complement of (the downward closure of) this set. This can be done using theorem 20 and DeMorgan's laws.

A dual result holds for the live **Conc** maps.

The extension to representations of functions from other domains is straightforward.

5.4 Using the upper and lower bounds

How can we make use of the ability to place upper and lower bounds on the value of a function? Here we outline a possible approach to using **Abs** and **Conc** in strictness analysis.

Suppose the function **f** is defined as **Y**(**F**) in a functional language. Abstract interpretation gives us a function $F \in [A \to A]$, where $A \in \mathscr{L}$, which safely approximates the standard interpretation of **F**. We wish to evaluate $f = \mathbf{fix}_A F$ but the lattice A may be too large for this to be practical. In this case we choose a smaller lattice, $A' \in \mathscr{L}$.

First we use $\mathbf{Abs}^{l}_{A,A'}$ and $\mathbf{Conc}^{l}_{A',A}$ to give us a lower bound on f, call this f_{lb}. Using f_{lb} we can place an upper limit on the degree of strictness which our abstract interpretation would determine if time and space allowed. If we find that even f_{lb} does

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

not imply that **f** is strict in ways in which we are interested, there is no need to proceed any further.

On the other hand, if the lower bound shows that **f** *may* be strict in such ways, we can go on to calculate an upper bound, say f_{ub}, using $\mathbf{Abs}^s_{A,A'}$ and $\mathbf{Conc}^s_{A',A}$. If f_{ub} confirms that **f** is strict, our job is done.

In the remaining case, f_{lb} and f_{ub} 'disagree' concerning the strictness of **f**. We must then decide whether to cut our losses and accept that we are unable to confirm that **f** is strict, or try to calculate improved upper and lower bounds by repeating the process using a new lattice, A'', such that $A' \preccurlyeq A'' \preccurlyeq A$. In choosing A'' we would have to be sure that we did not run into the complexity problem we are trying to avoid in the first place. This would not be entirely dependent on the absolute size of A'', since $\mathbf{Conc}^l_{A',A}(f_{lb})$ and $\mathbf{Conc}^s_{A',A}(f_{ub})$ place lower and upper bounds on the values of $\mathbf{fix}_A(\mathbf{Abs}^l_{A,A'}F)$ and $\mathbf{fix}_A(\mathbf{Abs}^s_{A,A'}F)$. The lower bound allows us to start the fixed point iterations at a point above $\perp_{A'}$ and, as was shown in section 4.2.4, the frontiers algorithm can use both upper and lower bounds as a means of reducing the search space when establishing the frontier of each approximation. One possibility this raises is that the work done in the smaller domains might achieve in a few 'big steps' what would take many 'little steps' in the original domain.

Further experimentation is needed to determine whether the use of **Abs** and **Conc** to render an abstract interpretation tractable should be an iterative process of refinement, as is suggested above, or whether we should choose a reasonably sized domain and stick with it.

6 Conclusions

We have provided a new approach to the use of frontiers in abstract interpretation which has allowed a concise and clear re-formulation of the frontiers algorithm. In addition we have addressed a serious complexity problem arising in practical applications of abstract interpretation. The latter work needs further development in the following areas:

- Heuristics are required to be used in a compiler to estimate the size of a lattice and choose a smaller one where necessary.
- Experimental evidence must be gathered to assess the costs and benefits of attempting to refine the approximations to a fixed point by increasing the sizes of the abstract domains.

Appendices

A Prerequisites

In this appendix we introduce some notation and basic definitions and facts which are central to the technical development of the article. Most of the following material is standard (see, for example, Gierz *et al.* 1980; Vickers, 1989).

A.1 Notation and terminology

A *poset* is a pair $(D, \sqsubseteq_D)$ where $\sqsubseteq_D$ is a partial order on the set D.

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

Given posets D and D', a function $f: D \to D'$ is said to be *order-embedding* (with respect to $\sqsubseteq_D$ and $\sqsubseteq_{D'}$) if and only if

$$\forall x, y \in D . fx \sqsubseteq_{D'} fy \text{ if and only if } x \sqsubseteq_D y$$

It is easy to verify that a function which is order-embedding must be both monotone and one-to-one.

A function $\mu; D \to D'$ is an *isomorphism* if and only if μ is both order-embedding and onto.

By *lattice*, we mean a poset in which all finite subsets have a meet and a join. Given a lattice D, the empty meet, written $\top_D$, is the greatest (top) element and the empty join, written $\bot_D$, is the least (bottom) element. A *complete* lattice is a lattice in which all subsets have a meet and a join. Note that every finite lattice is complete.

A.2 The Alexandrov and Scott topologies

A topology on a set X is a collection, $\Omega(X)$, of subsets of X, closed under finite intersections and arbitrary unions. The members of $\Omega(X)$ are known as the *open sets* of the topology. The *closed sets* of the topology are the members of $\Gamma(X) = \{C_X(A) \mid A \in \Omega(X)\}$. $\Gamma(X)$ is closed under finite unions and arbitrary intersections. A topology may be defined either in terms of its open sets or in terms of its closed sets. (Note that it is possible for a set to be both open *and* closed, in particular $\varnothing$ and X are both open and closed sets of all topologies on X.) $\Omega(X)$ and $\Gamma(X)$ form complete lattices, ordered by $\subseteq$.

Definition A.1

Given topologies $\Omega(X)$ and $\Omega(Y)$, a function $f: X \to Y$ is said to be *continuous* if

$$\forall A \in \Omega(Y) . f^{-1}A \in \Omega(X)$$

or, equivalently, if

$$\forall B \in \Gamma(Y) . f^{-1}B \in \Gamma(X)$$

A.2.1 The Alexandrov topology

Given a poset D and a subset $X \subseteq D$:

(i) X is *upper* if and only if $\forall d \in D,\ x \in X . d \sqsupseteq x \Rightarrow d \in X$
(ii) X is *lower* if and only if $\forall d \in D,\ x \in X . d \sqsubseteq x \Rightarrow d \in X$.

The upper and lower sets are the open sets $\Omega_A(D)$ and closed sets $\Gamma_A(D)$ respectively of the *Alexandrov topology* on D. It is easy to see that for posets D, D', a function $f: D \to D'$ is Alexandrov–continuous if and only if f is monotone.

When D' is a lattice then so is $[D \to D']$, with binary meets and joins given by $f \sqcap g = \lambda x \in D . fx \sqcap gx$ and $f \sqcup g = \lambda x \in D . fx \sqcup gx$.

A.2.2 The Scott topology

Given a poset D, a subset $A \subseteq D$ is said to be *directed* if and only if every finite subset of A has an upper bound in A.

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

The *Scott topology* on a poset D may be defined as follows: a subset $A \subseteq D$ is Scott-closed if and only if A is

(i) lower
(ii) closed under the joins of directed sets, i.e. for every directed subset $S \subseteq A$, if $\bigsqcup S$ exists then $\bigsqcup S \in A$

Given posets D, D', we will denote the poset of Scott-continuous functions from D to D', ordered pointwise on D, by $[D \to_c D']$.

In computing, discussions of continuity often occur in the context of cpos, and a function f between cpos D and D' is sometimes said to be continuous if and only if for every chain $\{x_n\} \subseteq D$,

$$f(\bigsqcup\{x_n\}) = \bigsqcup(f\{x_n\})$$

For the ω-algebraic cpos this notion of continuity is equivalent to Scott-continuity.

A.3 Finite posets

In this article we have been concerned mainly with finite posets, for which the Scott and Alexandrov topologies coincide, i.e. for finite poset D and subset $X \subseteq D$:

(i) X is Scott-open if and only if X is upper
(ii) X is Scott-closed if and only if X is lower.

As a consequence, for finite posets $[D \to_c D'] = [D \to D']$.

Given a poset D, for any subset $X \subseteq D$, the set $\uparrow X$ (resp. $\downarrow X$) is upper (resp. lower) and hence, for finite D, Scott-open (resp. Scott-closed).

A.4 Duality

The *opposite* of a poset D is the poset $D^{op} = (D, \sqsubseteq_D^{-1})$, i.e. D with the ordering reversed. Note that $(D^{op})^{op} = D$.

For any set D, function $f: D \to \mathbf{2}$, we define the *negation* of f to be $\bar{f}: D \to \mathbf{2}$, where

$$\bar{f}x = \begin{cases} 0 & \text{if } fx = 1 \\ 1 & \text{if } fx = 0 \end{cases}$$

Note that $\bar{\bar{f}} = f$.

For poset D, upper set $A \subseteq D$, the *characteristic function* of A, $\chi_A \in [D \to \mathbf{2}]$ is defined by

$$\chi_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A. \end{cases}$$

Let D and D' be posets. The following dualities hold;

- for all $A \subseteq D$, when the join of A exists in D, it is equal to the meet of X in D^{op}
- D is a lattice if and only if D^{op} is a lattice
- for all $f \in [D \to D']$, f is order-embedding with respect to $\sqsubseteq_D$ and $\sqsubseteq_{D'}$ if and only if f is order-embedding with respect to $\sqsubseteq_{D^{op}}$ and $\sqsubseteq_{D'^{op}}$
- for all $A \subseteq D$, $\uparrow_D A = \downarrow_{D^{op}} A$

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

- for all $A \subseteq D$, A is upper in D if and only if A is lower in D^{op}
- $[D \to D']^{op} = [D^{op} \to D'^{op}]$
- $(D \times D')^{op} = D^{op} \times D'^{op}$
- for all $f: D \to \mathbf{2}$, $f \in [D \to \mathbf{2}]$ if and only if $\bar{f} \in [D^{op} \to \mathbf{2}]$
- for all $f, g \in [D \to \mathbf{2}]$, $f \sqsubseteq_{[D \to \mathbf{2}]} g$ if and only if $\bar{g} \sqsubseteq_{[D^{op} \to \mathbf{2}]} \bar{f}$
- for all upper subsets, $A \subseteq D$, $\chi_{C(A)} = \overline{\chi_A}$

B Technical results required for the construction of succs and preds

In this appendix we provide proofs for the lemmas stated in subsection 4.2.3. First we need two new lemmas (recall that a function f is said to be *order-embedding* when $f(x) \sqsubseteq f(y) \Leftrightarrow x \sqsubseteq y$):

Lemma A.2
Given posets D and D', function $f: D \to D'$, and subset $X \subseteq D$, if f is order-embedding then

(i) $\mathbf{min}_{D'}(fX) = f(\mathbf{min}_D(X))$
(ii) $\mathbf{max}_{D'}(fX) = f(\mathbf{max}_D(X))$

Proof
We prove (i) directly. (ii) follows by duality.
$\subseteq$: assume $y \in \mathbf{min}(fX)$, i.e. $y = f(x)$ for some $x \in X$, and $f(x)$ is minimal in fX. Suppose $x' \sqsubseteq x$ for some $x' \in X$. Then $f(x') \sqsubseteq f(x)$, since f is monotone. Hence $f(x') = f(x)$, since $x' \in X$ and $f(x)$ is minimal in fX. But then $x' = x$, since f is one-to-one. Hence $x \in \mathbf{min}(X)$. Hence $y \in f(\mathbf{min}(X))$.
$\supseteq$: assume $y \in f(\mathbf{min}(X))$, i.e. $y = f(x)$ for some $x \in X$ such that x is minimal in X. Suppose $f(x') \sqsubseteq f(x)$, for some $x' \in X$. Then $x' \sqsubseteq x$, since f is order-embedding. But then $x' = x$, since x is minimal in X, and so $f(x') = f(x)$. Hence $y \in \mathbf{min}(fX)$. □

Lemma A.3
For $(x_1, \ldots, x_n) \in D_1 \times \ldots \times D_n$, for each i, $1 \leqslant i \leqslant n$:

(i) $\mathbf{min}(\{(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n \mid d_i \not\sqsubseteq x_i\}) = \{S_i(d) \mid d \in \mathbf{succs}(x_i)\}$
(ii) $\mathbf{max}(\{(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n \mid d_i \not\sqsupseteq x_i\}) = \{P_i(d) \mid d \in \mathbf{preds}(x_i)\}$

Proof
We prove (i) directly. (ii) follows by duality.
It is straightforward to verify that for $X_j \subseteq D_j$, $1 \leqslant j \leqslant n$:

$$\mathbf{min}_{D_1 \times \ldots \times D_n}(X_1 \times \ldots \times X_n) = \mathbf{min}_{D_1}(X_1) \times \ldots \times \mathbf{min}_{D_n}(X_n)$$

Then, for each i, $1 \leqslant i \leqslant n$:

$$\begin{aligned}
&\mathbf{min}(\{(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n \mid d_i \in C(\downarrow x_i)\}) \\
&= \mathbf{min}(D_1 \times \ldots \times D_{i-1} \times C(\downarrow x_i) \times D_{i+1} \times \ldots \times D_n) \\
&= \mathbf{min}(D_1) \times \ldots \times \mathbf{min}(D_{i-1}) \times \mathbf{min}(C(\downarrow x_i)) \times \mathbf{min}(D_{i+1}) \times \ldots \times \mathbf{min}(D_n)) \\
&= \{\bot_{D_1}\} \times \ldots \times \{\bot_{D_{i-1}}\} \times \mathbf{min}(C(\downarrow x_i)) \times \{\bot_{D_{i+1}}\} \times \ldots \{\bot_{D_n}\} \\
&= \{S_i(x) \mid x \in \mathbf{min}(C(\downarrow x_i))\} \quad \square
\end{aligned}$$

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

We can now restate and prove lemmas 21 and 22

Lemma 21
For $x_i \in D_i$, $1 \leqslant i \leqslant n$:

(i) $\mathbf{succs}(x_1, \ldots, x_n) = \bigcup_{i=1}^{n} \{S_i(d) \mid d \in \mathbf{succs}(x_i)\}$

(ii) $\mathbf{preds}(x_1, \ldots, x_n) = \bigcup_{i=1}^{n} \{P_i(d) \mid d \in \mathbf{preds}(x_i)\}$.

Proof
We prove (i) directly. (ii) follows by duality.

$$
\begin{aligned}
&\mathbf{succs}(x_1, \ldots, x_n) \\
&= \mathbf{min}(\{(d_1, \ldots, d_n) \in A \mid (d_1, \ldots, d_n) \not\sqsupseteq (x_1, \ldots, x_n)\}) \\
&= \mathbf{min}\left(\left\{(d_1, \ldots, d_n) \in A \mid \bigvee_{i=1}^{n} d_i \not\sqsupseteq x_i\right\}\right) \\
&= \mathbf{min}\left(\bigcup_{i=1}^{n} \{(d_1, \ldots, d_n) \in A \mid d_i \not\sqsupseteq x_i\}\right) \\
&= \mathbf{min}\left(\bigcup_{i=1}^{n} \mathbf{min}(\{(d_1, \ldots, d_n) \in A \mid d_i \sqsubseteq x_i\})\right) \quad \text{lemma 16} \\
&= \mathbf{min}\left(\bigcup_{i=1}^{n} \{S_i(d) \mid d \in \mathbf{succs}(x_i)\}\right) \quad \text{lemma A.3}
\end{aligned}
$$

It is straightforward to verify that the set $\bigcup_{i=1}^{n} \{S_i(d) \mid d \in \mathbf{succs}(x_i)\}$ is irredundant. The required result then follows immediately by lemma 8. □

Lemma 22
For finite poset D, function $f \in [D \to \mathbf{2}]$:

(i) $\mathbf{succs}_{[D \to \mathbf{2}]}(f) = \mathbf{min}_{[D \to \mathbf{2}]}(C_{[D \to \mathbf{2}]}(\downarrow_{[D \to \mathbf{2}]} f)) = \{\chi_{\uparrow x} \mid x \in \mathbf{F\text{-}0}_D(f)\}$
(ii) $\mathbf{preds}_{[D \to \mathbf{2}]}(f) = \mathbf{max}_{[D \to \mathbf{2}]}(C_{[D \to \mathbf{2}]}(\uparrow_{[D \to \mathbf{2}]} f)) = \{\chi_{C(\downarrow x)} \mid x \in \mathbf{F\text{-}1}_D(f)\}$

Proof
(i) For $g \in [D \to \mathbf{2}]$ it is straightforward to verify that $g \sqsubseteq f$ if and only if $\mathbf{F\text{-}0}(f) \cap g^{-1}\{1\} = \varnothing$.
Then:

$$
\begin{aligned}
&\mathbf{min}(\{g \in [D \to \mathbf{2}] \mid g \not\sqsupseteq f\} \\
&= \mathbf{min}(\{g \in [D \to \mathbf{2}] \mid \mathbf{F\text{-}0}(f) \cap g^{-1}\{1\} \neq \varnothing\}) \\
&= \mathbf{min}(\{g \in [D \to \mathbf{2}] \uparrow \bigvee_{x \in \mathbf{F\text{-}0}(f)} x \in g^{-1}\{1\}\}) \\
&= \mathbf{min}\left(\bigcup_{x \in \mathbf{F\text{-}0}(f)} \{g \in [D \to \mathbf{2}] \mid x \in g^{-1}\{1\}\}\right) \\
&= \mathbf{min}\left(\bigcup_{x \in \mathbf{F\text{-}0}(f)} \{g \in [D \to \mathbf{2}] \mid \uparrow x \subseteq g^{-1}\{1\}\}\right)
\end{aligned}
$$

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

$$= \mathbf{min}\Big(\bigcup_{x \in \mathbf{F\text{-}0}(f)} \{g \in [D \to \mathbf{2}] \mid \chi_{\uparrow x} \sqsubseteq g\}\Big)$$

$$= \mathbf{min}\Big(\bigcup_{x \in \mathbf{F\text{-}0}(f)} \uparrow\chi_{\uparrow x}\Big)$$

$$= \mathbf{min}\Big(\bigcup_{x \in \mathbf{F\text{-}0}(f)} \mathbf{min}(\uparrow\chi_{\uparrow x})\Big) \quad \text{lemma 16}$$

$$= \mathbf{min}\Big(\bigcup_{x \in \mathbf{F\text{-}0}(f)} \mathbf{min}(\{\chi_{\uparrow x}\})\Big) \quad \text{lemma 15}$$

$$= \mathbf{min}\Big(\bigcup_{x \in \mathbf{F\text{-}0}(f)} \{\chi_{\uparrow x}\}\Big)$$

That the set $\{\chi_{\uparrow x} \mid x \in \mathbf{F\text{-}0}(f)\}$ is irredundant follows from the irredundancy of $\mathbf{F\text{-}0}(f)$. The required result then follows immediately by lemma 8.

(ii) follows by duality. □

Before providing the proof for lemma 23 we require the following two additional lemmas:

Lemma A.4

For any $h \in [D \to \mathbf{2}]$, $D' \in \mathscr{L}$:

(i) $\mathbf{min}(\{g \in [D \to D'_\perp] \mid L_g = h\}) = \{lemb_{D'}(h)\}$

(ii) $\mathbf{max}(\{g \in [D \to D'_\perp] \mid L_g = h\}) = \{gemb_{D'}(h)\}$

Proof

(i) Let $X = \{g \in [D \to D'_\perp] \mid L_g = h\}$.

We will show that $lemb_{D'}(h) \in X$ and that $\forall f \in X.\, lemb_{D'}(h) \sqsubseteq f$.

First, let $h' = L_f$, where $f = lemb_{D'}(h)$. Then there are two cases to consider for any $x \in D$:

$hx = 0$: $lemb_{D'}(h)(x) = \perp$, hence $h'x = 0$.

$hx = 1$: $lemb_{D'}(h)(x) = (lift_{D'}\, \perp_{D'}) \neq \perp$, hence $h'x = 1$.

Thus $h' = h$ and so $lemb_{D'}(h) \in X$.

Secondly, assume $f \in X$. Then, again, there are two cases to consider for any $x \in D$:

$fx = \perp$: then $L_f(x) = 0$. Thus $hx = 0$, hence $lemb_{D'}(h)(x) = \perp = fx$.

$fx \neq \perp$: then $L_f(x) = 1$. *Thus* $hx = 1$, hence $lemb_{D'}(h)(x) = (lift_{D'}\, \perp_{D'}) \sqsubseteq fx$.

Hence $lemb_{D'}(h) \sqsubseteq f$.

(ii) Similar. □

Lemma A.5

For any $h \in [D \to D']$:

(i) $\mathbf{min}(\{g \in [D \to D'_\perp] \mid H_g = h\}) = [llift(h)\}$

(*ii*) $\mathbf{max}(\{g \in [D \to D'_\perp] \mid H_g = h\}) = \{glift(h)\}$

Proof

(i) Let $X = \{g \in [D \to D'_\perp] \mid H_g = h\}$.

We will show that $llift(h) \in X$ and that $\forall f \in X.\, llift(h) \sqsubseteq f$.

First, let $h' = H_f$ where $f = llift(h)$. Then there are two cases to consider for any $x \in D$:

https://doi.org/10.1017/S0956796800000071 Published online by Cambridge University Press

$hx = \bot_{D'}$: then $llift(h)(x) = \bot$. Thus $h'x = \bot_{D'} = hx$.
$hx \neq \bot_{D'}$: then $llift(h)(x) = lift_{D'}(hx) \neq \bot$. Thus $h'x = hx$.
Thus $h' = h$ and so $llift(h) \in X$.
Secondly, assume $f \in X$. Then there are three cases to consider for any $x \in D$:
$fx = \bot$: then $H_f(x) = \bot_{D'} = hx$. Thus $llift(h)(x) = \bot = fx$.
$fx = lift_{D'} \bot_{D'}$: then $H_f(x) = \bot_{D'} = hx$. Thus $llift(h)(x) = \bot \sqsubseteq fx$.
$fx = lift_{D'} d, d \neq \bot_{D'}$: then $H_f(x) = d = hx$. Thus $llift(h)(x) = lift_{D'} d = fx$.
Hence $llift(h) \sqsubseteq f$.

(ii) Similar. □

Finally, we can restate and prove:

Lemma 23
For all $f \in A = [D \to D'_\bot]$, $\mathbf{succs}(f) = \mathbf{min}(S_1 \cup S_2)$, where:

$$S_1 = \{lemb_{D'}(h) \mid h \in \mathbf{succs}(L_f)\}$$
$$S_2 = \{llift(h) \mid h \in \mathbf{succs}(H_f)\}$$

Proof
We note that $lemb_{D'}$ and $llift$ are order-embedding.

$$\begin{aligned}
&\mathbf{succs}(f)\\
&= \mathbf{min}(\{g \in A \mid g \sqsupsetneq f\})\\
&= \mathbf{min}(\{g \in A \mid L_g \sqsupsetneq L_f \text{ or } H_g \sqsupsetneq H_f\}) \quad \text{lemma 1}\\
&= \mathbf{min}(\{g \in A \mid L_g \sqsupsetneq L_f\} \cup \{g \in A \mid H_g \sqsupsetneq H_f\})\\
&= \mathbf{min}(\mathbf{min}(\{g \in A \mid L_g \sqsupsetneq L_f\}) \cup \mathbf{min}(\{g \in A \mid H_g \sqsupsetneq H_f\})) \quad \text{lemma 16}
\end{aligned}$$

Now

$$\begin{aligned}
&\mathbf{min}(\{g \in A \mid L_g \sqsupsetneq L_f\})\\
&= \mathbf{min}(\{g \in A \mid L_g \in C(\downarrow L_f)\})\\
&= \mathbf{min}(\{g \in A \mid \bigvee_{h \in C(\downarrow L_f)} L_g = h\})\\
&= \mathbf{min}(\bigcup_{h \in C(\downarrow L_f)} \{g \in A \mid L_g = h\})\\
&= \mathbf{min}(\bigcup_{h \in C(\downarrow L_f)} \mathbf{min}(\{g \in A \mid L_g = h\})) \quad \text{lemma 16}\\
&= \mathbf{min}(\{lemb_{D'}(h) \mid h \in C(\downarrow L_f)\}) \quad \text{lemma A.4}\\
&= \{lemb_{D'}(h) \mid h \in \mathbf{min}(C(\downarrow L_f))\} \quad \text{lemma A.2}\\
&= S_1
\end{aligned}$$

Similarly, using lemma A.5, it can be shown that $\mathbf{min}(\{g \in A \mid H_g \sqsupsetneq H_f\}) = S_2$. □

Acknowledgements

Much of the credit for the ideas behind this article must go to Chris Martin, who developed the frontiers algorithm to the point where a new perspective became apparent.

We would like to thank Jesper Andersen for his help in the preparation of an earlier version of this article (Hunt, 1989), and Simon Peyton Jones for his many valuable comments and suggestions for improving the presentation of the material.

The authors were partially funded by ESPRIT BRA 3124:Semantique and ESPRIT BRA 3074:Semagraph.

References

Abramsky, Samson. 1986. Strictness analysis and polymorphic invariance. In H. Ganzinger and N. D. Jones (editors), *Programs as Data Objects, Lecture Notes in Computer Science 217*. Springer-Verlag.

Abramsky, Samson and Hankin, Chris (editors). 1987. *Abstract Interpretation of Declarative Languages*. Ellis Horwood.

Burn, G. L., Hankin, Chris and Abramsky, Samson. 1986. The theory of strictness analysis for higher-order functions. In H. Ganzinger and N. D. Jones (editors), *Programs as Data Objects, Lecture Notes in Computer Science 217*, pp. 42–62, Springer-Verlag.

Burn, G. L. 1987. *Abstract Interpretation and the Parallel Evaluation of Functional Languages*. PhD thesis, Department of Computing, Imperial College of Science and Technology, University of London.

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

Gierz, G., Hofmann, K. H., Keimel, K., Lawson, J. D., Mislove, M. and Scott, D. S., 1980. *A Compendium of Continuous Lattices*. Springer-Verlag.

Goldberg, B. and Park, Y. G. 1990. Higher-order escape analysis: optimising stack allocation in functional program implementations. In N. D. Jones (editor), *ESOP '90, 3rd European Symposium on Programming, Copenhagen, Denmark. May 1990, Lecture Notes in Computer Science 432*. Springer-Verlag.

Hughes, John. 1988. Abstract interpretation of first-order polymorphic functions. In *Glasgow Workshop on Functional Programming*, University of Glasgow, Department of Computing Science (August 1988). Research Report 89/R4.

Hunt, Sebastian. 1989. Frontiers and open sets in abstract interpretation. In D. MacQueen (editor), *Functional Programming Languages and Computer Architecture*, pp. 1–11. ACM Publications (September 1989).

Peyton Jones, Simon and Clack, Chris. 1987. Finding fixpoints in abstract interpretation. In Samson Abramsky, and Chris Hankin (editors), *Abstract Interpretation of Declarative Languages*, chapter 11. Ellis Horwood.

Jones, N. D., Sestoft, P. and Søndergaard. H. 1985. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud (editor), *Rewriting Techniques and Applications, Lecture Notes in Computer Science 202*, pp. 124–140. Springer-Verlag.

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

Martin, Chris. and Hankin, Chris. 1987. Finding fixed points in finite lattices. In G. Kahn (editor), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 274*, pp. 426–45. Springer-Verlag (September 1987).

Mycroft, A. 1989. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edinburgh.

Vickers, S. J. 1989. *Topology via Logic*. Cambridge University Press.

Wadler, P. 1987. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris Hankin (editors) *Abstract Interpretation of Declarative Languages*, chapter 12. Ellis Horwood.

Sebastian Hunt and Chris Hankin, Department of Computing, Imperial College, London SW7 2BZ.