J. Functional Programming 2 (4): 475–503, October 1992

Computing with lattices: An application of type classes

MARK P. JONES

Programming Research Group, Oxford University, UK[1]

Abstract

This paper presents a simple framework for performing calculations with the elements of (finite) lattices. A particular feature of this work is the use of type classes to enable the use of overloaded function symbols within a strongly typed language. Previous applications of type classes have been in areas that are of most interest to language implementors. This paper suggests that type classes might also be useful as a general tool in the development of clear and modular programs.

Capsule review

Type classes have been the subject of considerable debate among functional language designers and implementors ever since they were proposed by Wadler and Blott in 1989, and they are among the important distinguishing features of the recent Haskell language design. But do programmers actually make use of them – not just as an unavoidable aspect of certain primitives, but by the introduction of completely new classes at a higher level? If they do, what advantage (if any) is obtained?

This paper presents an extended example of the use of type classes as a high level programming tool. It begins with the definition of a new class – the class of *lattices* – and goes on to develop a series of polymorphic routines for working with all kinds of finite lattices. The latter part of the paper compares the resulting program with an earlier implementation in a language without type classes, identifying some clear benefits of using type classes for this application.

1 Introduction

There are many applications for lattices in computer science, particularly in the static analysis of programs where the most common example in the context of functional programming is that of strictness analysis. This paper describes a functional program which provides a simple framework for performing calculations with the elements of finite lattices in such applications. A particular feature of this work is the use of *type classes*, introduced by Wadler and Blott (1989) to enable the use of overloaded function symbols within a strongly typed language, and adopted as part of the

[1] Author's current address: *Department of Computer Science, Yale University, PO Box 2158 Yale Station, New Haven, CT 06520-2158, USA* (email: jones-mark@cs.yale.edu.)

standard for the programming language Haskell (Hudak *et al.*, 1991). The result is a highly modular program which (in the author's opinion) is easier to read, and hence understand, than an earlier version written without the use of type classes.

We begin with a discussion of some of the weaknesses of standard Hindley/Milner typing, and describe how these problems are addressed by the use of type classes (section 2). Section 3 introduces our basic framework for computing with lattices, and illustrates how this can be used to describe a range of different lattices and a general purpose implementation of the least fixed point operator.

The problem of working with lattices of functions is discussed in section 4, where we describe a compact representation for monotonic functions based on the use of *frontiers*. In section 5 we extend the framework to describe a class of *navigable* lattices which provide additional operators for working with frontier values. We describe a number of interesting applications of these operators, including a simple algorithm to enumerate the elements of a finite navigable lattice.

Section 6 illustrates how the use of type classes supports a modular style of programming by showing how additional lattices can be added to our framework without requiring any changes to the original program. Section 7 outlines an earlier implementation of the framework described in this paper, and highlights some of the benefits of using type classes instead of standard Hindley/Milner typing. Finally, motivated by these programs, section 8 describes two features which would be useful in a practical system supporting type classes.

2 An introduction to type classes

2.1 *Difficulties with Hindley/Milner typing*

Many functional programming languages provide some form of polymorphic equality operator:

$$(==) :: a \to a \to Bool$$

which can be used to compare the elements of a wide variety of types including integers, characters and algebraic datatypes such as lists and tuples. It follows that the $(==)$ operator cannot be defined explicitly in a language based on standard Hindley/Milner typing, since there is no common instance of each of these separate types. The equality operator must therefore be incorporated into the language as a 'primitive' function. This has a number of disadvantages:

- The implementation of $(==)$ provided by the system will (almost certainly) test for equality of representation rather than equality of represented values. When this is not appropriate, the programmer must define a new function to implement the required comparison. This results in complexity and confusion, introducing a number of distinct names for a single concept; testing for equality.
- The implementation of a language may be complicated by the need to support polymorphic primitives such as $(==)$; the coding of the primitives will be complex and may require additional runtime tags on data structures.
- The type system is unable to detect illegal uses of $(==)$; for example, an attempt to compare values of type $(a \to b)$ is not usually detected until runtime.

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

- The definition of (==) is completely hidden. An explicit definition is often useful in the development and derivation of programs, and in proofs based on equational reasoning.

In specific cases, some of these problems can be eliminated by *ad hoc* methods; one of the main reasons for the introduction of type classes is to provide a general mechanism for resolving these kinds of problem.

2.2 Type classes and instances

Broadly speaking, a *type class* is a family of types (whose members are called *instances* of the class) for which a collection of functions (the *member functions* of the class) are defined. A type class is introduced by a declaration of the form:

class *ClassName a* **where** *signature*,

where *a* is a type variable representing an arbitrary instance of the class called *ClassName*, and *signature* specifies the names and types (usually involving the type variable *a*) of each of the member functions. It is sometimes useful to define a class with no member functions, in which case the signature part of the declaration (together with the preceding **where** keyword) should be omitted (see, for example, the definition of the class *Tuple* in section 3.4.1).

In the case of the equality operator described above, we define a class *Eq* containing all those types on which the (==) function may be used with the declaration:

class *Eq a* **where**
$$(==) :: a \to a \to Bool.$$

The instances of each type class are defined in a similar manner, and we illustrate this with some simple examples:

- **Primitive types** For reasons of efficiency, most language implementations provide special representations for some primitive types. Suppose, for example, that integers are represented by elements of the type *Int*, and that the equality of integers is described by a primitive monomorphic function:

  $$primEqInt :: Int \to Int \to Bool.$$

  This can be included as part of the definition of the polymorphic equality operator (==) using the instance declaration:

  instance *Eq Int* **where**
  $$(==) = primEqInt.$$

- **Algebraic datatypes** Many languages provide some means of defining and manipulating objects of non-primitive types. An example of this is the type [*a*] (representing lists of elements of type *a*), which behaves as if defined as an algebraic datatype:

  data $[a] = [\,] \mid a : [a]$.

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

Given a definition of $(==)$ for elements of type a, there is a natural way to define $(==)$ for elements of type $[a]$. This is captured by the instance declaration:

$$\begin{aligned}&\textbf{instance } Eq\ a \Rightarrow Eq\ [a] \textbf{ where}\\ &\quad [] == [] &&= True\\ &\quad [] == (y:ys) &&= False\\ &\quad (x:xs) == [] &&= False\\ &\quad (x:xs) == (y:ys) &&= x == y \wedge xs == ys.\end{aligned}$$

The clause $Eq\ a \Rightarrow Eq\ [a]$ in the first line of this declaration indicates that $[a]$ can only be an instance of Eq if a is itself an instance of Eq. The right hand side of the last line uses two distinct forms of the $(==)$ operator; the first is used to compare elements of type a, while the second is used to compare elements of type $[a]$. Function symbols such as $(==)$ are said to be *overloaded* because they can have different interpretations for different argument types.

As a further example, the following instance declaration indicates how the equality of two pairs may be defined in terms of the equality of the individual components in each pair:

$$\begin{aligned}&\textbf{instance } Eq\ a, Eq\ b \Rightarrow Eq\ (a,b) \textbf{ where}\\ &\quad (x,y) == (u,v) = (x == u \wedge y == v).\end{aligned}$$

The three instance declarations above are sufficient to define each of the types Int, $[Int]$, (Int, Int), $[(Int, Int)]$, $(Int, [Int])$, etc., as instances of Eq. Further instance declarations can be used to extend Eq to other types.

The flexibility of being able to give explicit definitions for the equality operation on different types soon becomes a burden when a large number of instances are involved. In this situation, it is useful for the system to be able to generate 'default' instance declarations for commonly used type classes. In the terminology of Haskell, these are known as *derived instances*. For the class Eq, there is a natural way to define an equality operation on the elements of an arbitrary algebraic datatype, so long as all of the subcomponents of the objects of that type are in the class Eq – the definitions of equality on lists and pairs given above can both be obtained in this way. Further details are given in the Haskell report, including a description of how derived instances for Eq (and a number of additional 'standard' classes) are obtained. As we illustrate in section 3, Haskell also allows the programmer to include default definitions for individual member functions as part of each class declaration. For the purposes of this paper, we assume that any instance of Eq which is not explicitly defined will be generated automatically as a derived instance whenever this is possible.

2.3 Polymorphic type checking

The extended form of polymorphism provided by the use of type classes cannot be completely described in terms of universal quantification, as in Damas and Milner (1982). Instead, we use a notion of *qualified types* with type expressions of the form

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

$C \Rightarrow t$, where t is a standard type expression and C is a set of *predicates* constraining the values taken by the type variables in t to be instances of particular classes. For example, the type of $(==)$ is specified by:

$$(==) :: Eq\ a \Rightarrow a \to a \to Bool.$$

The same notation is used to give the types of functions whose definition depends either directly or indirectly on the member functions of a particular class. For example:

$$\begin{aligned} &\textbf{(elem)} && :: Eq\ a \Rightarrow a \to [a] \to Bool \\ &x\ \textbf{elem}\ ys && = any\ (x ==)\ ys \\ &(\subseteq) && :: Eq\ a \Rightarrow [a] \to [a] \to Bool \\ &r \subseteq s && = all\ (\textbf{elem}\ s)\ r. \end{aligned}$$

Expressions of the form $x == y$ can be type-checked by ensuring that the subexpressions x and y have the same type (as a conventional type checker would), and that this type is an instance of the class *Eq*. If we avoid defining an instance of *Eq* for types of the form $(a \to b)$ then any attempt to compare two functions will be detected and reported as an error during type checking.

2.4 Superclasses

In languages supporting a range of types representing numeric data (including perhaps integers, floating point, rational and complex numbers), it is convenient to use overloaded function symbols for the standard arithmetic functions so that $(+)$ can be used to denote the appropriate addition functions on each of the numeric types. This can be described quite easily by defining a type class *Num*:

$$\begin{aligned} &\textbf{class}\ Num\ a\ \textbf{where} \\ &\quad (+) :: a \to a \to a \\ &\quad zero :: a. \end{aligned}$$

Consider the function defined by $isZero\ x = (x == zero)$ which can be used to determine if an element of any numeric type (i.e. any instance of *Num*) is zero. Note that the type of *isZero* is $(Num\ a, Eq\ a) \Rightarrow a \to Bool$ so that the type represented by a must be an instance of both *Num* and *Eq*. In practice, it is usually reasonable to assume that all instances of *Num* are instances of *Eq*.[1] This can be specified in the definition of the class:

$$\begin{aligned} &\textbf{class}\ Eq\ a \Rightarrow Num\ a\ \textbf{where} \\ &\quad (+) :: a \to a \to a \\ &\quad zero :: a. \end{aligned}$$

The expression $Eq\ a \Rightarrow Num\ a$ should be read as '*Eq* is a *superclass* of *Num*'. With this definition of *Num*, the type of *isZero* simplifies to $Num\ a \Rightarrow a \to Bool$. Simple static

[1] However, such a decision would not be appropriate if we wanted to include some form of exact arithmetic – for example, computable real numbers (Vuillemin, 1988) – as an instance of *Num*.

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

checks must be used to ensure that every instance declaration for the class *Num* has a corresponding declaration for the class *Eq*.

2.5 Applications

To date, the principal uses of type classes have been:

- To enable frequently used polymorphic operators to be defined explicitly, without relying on implementations of these functions as 'primitives' of the language. Typical examples of this include the equality operator ($==$), discussed above, and the *read* and *show* functions used to provide a systematic treatment of parsing and printing of values in Haskell.
- To support the use of overloaded arithmetic operators on a range of numeric types; for example, using ($+$) to denote both addition of integers and addition of floating point numbers.

Such issues are likely to be of most interest to the language implementor. The following work suggests, by means of a particular example, that type classes might also be useful as a general tool in the development of clear and modular programs.

3 Computing with lattices

3.1 A type class for lattices

A lattice is represented by a type a with a partial order ($\sqsubseteq$) represented by a function:

$$(\sqsubseteq) :: a \to a \to Bool,$$

with least (bottom) and greatest (top) elements:

$$\bot, \top :: a,$$

and binary operators which determine the greatest lower bound (meet) and least upper bound (join) of a pair of elements:

$$(\sqcap), (\sqcup) :: a \to a \to a.$$

Using the notation of type classes, we say that such types are instances of the class *Lattice* which might initially be defined using:

class *Lattice* a **where**

$$\begin{array}{ll} (\sqsubseteq) & :: a \to a \to Bool \\ \bot, \top & :: a \\ (\sqcap), (\sqcup) & :: a \to a \to a. \end{array}$$

Given that any partial order is anti-symmetric, it is reasonable to assume that *Eq* is a superclass of *Lattice* with the equality operator ($==$) and the partial order ($\sqsubseteq$) related by the law:

$$(x == y) = (x \sqsubseteq y \wedge y \sqsubseteq x).$$

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

The relationship between the lattice operators $(\sqcap), (\sqcup)$ and the partial order $(\sqsubseteq)$ is described by the laws:

$$x \sqsubseteq y = (x \sqcup y) == y$$
$$x \sqsubseteq y = (x \sqcap y) == x.$$

These observations suggest the use of a simplified definition for the class *Lattice*:

$$\begin{aligned}&\textbf{class } \mathit{Eq}\ a \Rightarrow \mathit{Lattice}\ a \textbf{ where}\\ &\quad \bot, \top \quad :: a\\ &\quad (\sqcap), (\sqcup) :: a \to a \to a.\end{aligned}$$

Either of the laws above can be used to define $(\sqsubseteq)$ as an auxiliary function. For example:

$$\begin{aligned}&(\sqsubseteq) \quad :: \mathit{Lattice}\ a \Rightarrow a \to a \to \mathit{Bool}\\ &x \sqsubseteq y = (x \sqcup y) == y.\end{aligned}$$

Program development is simplified by reducing the number of member functions in a class:

- Fewer function definitions are required in each instance declaration.
- Proof obligations are reduced or simplified in cases where each new instance declaration requires the proof of laws constraining the values of the member functions (see section 8.2).

On the other hand, the use of a general definition rather than an instance-specific definition for functions such as $(\sqsubseteq)$ may carry a runtime cost. In Haskell, this problem is solved by allowing class declarations to include default definitions for member functions. We will therefore use the following definition of the class *Lattice* in place of the previous definitions for the class and for the function $(\sqsubseteq)$:

$$\begin{aligned}&\textbf{class } \mathit{Eq}\ a \Rightarrow \mathit{Lattice}\ a \textbf{ where}\\ &\quad \bot, \top \quad :: a\\ &\quad (\sqcap), (\sqcup) :: a \to a \to a\\ &\quad (\sqsubseteq) \quad :: a \to a \to \mathit{Bool}\\ &\quad x \sqsubseteq y \quad = (x \sqcup y) == y.\end{aligned}$$

This enables individual instance declarations to provide efficient instance-specific versions of the $(\sqsubseteq)$ function, with the original general definition being used when no other implementation is given.

3.2 The two point lattice of boolean values

The simplest lattice that we will consider is the two point lattice of boolean values with $\mathit{False} \sqsubseteq \mathit{True}$ and implemented by the algebraic datatype:

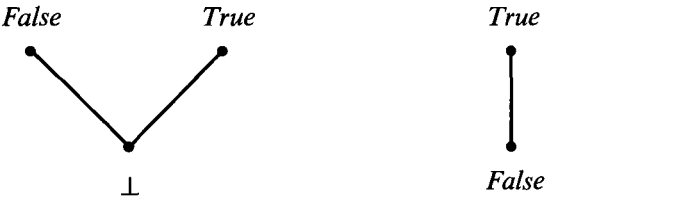
$$\textbf{data } \mathit{Bool} = \mathit{False} \mid \mathit{True}.$$

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

The lattice operations on *Bool* are just the standard boolean operations:

instance *Lattice Bool* **where**
$\bot = False$
$\top = True$
$(\sqcap) = (\wedge)$
$(\sqcup) = (\vee)$.

It is important to distinguish the semantics of the type *Bool* from the lattice structure described by this instance declaration. The former is usually described by the elements of a three point flat domain (i.e. a complete partial order, not a lattice) in which the elements *False* and *True* are incomparable. The latter is a (totally ordered) two point lattice. Calculations involving the semantic bottom element cannot be used in effective computations, and we will only consider proper elements in the definition of instances of *Lattice*. In particular, the bottom element of a lattice is *not* represented by the bottom element in the corresponding semantic domain:



Semantics of *Bool* Structure of *Lattice Bool*

3.3 The least fixed point operator

Suppose that L is a finite lattice and that $f :: L \to L$ is a monotonic function (so that $x \sqsubseteq y \Rightarrow fx \sqsubseteq fy$). A *fixed point* of f is an element x in L such that $fx = x$. It is a standard result that the set of fixed points of f forms a sublattice of L. In particular, f has a least fixed point, denoted fix f, which may be calculated as the limit of the ascending Kleene chain:

$$\bot \sqsubseteq f\bot \sqsubseteq f^2\bot \sqsubseteq f^3\bot \sqsubseteq \ldots \sqsubseteq f^n\bot \sqsubseteq \ldots.$$

Since L is finite, this chain is guaranteed to reach a fixed point after only finitely many steps, and this limit will be *fix f*. Noticing that the list of elements in the chain is produced by the expression *iterate f* $\bot$, we can define the fixed point operator *fix* using:

$$fix :: Lattice\ a \Rightarrow (a \to a) \to a$$
$$fix\ f = firstRepeat\ (iterate\ f\ \bot)$$

where *firstRepeat* determines the first repeated element in an infinite list:

$$firstRepeat :: Eq\ a \Rightarrow [a] \to a$$
$$firstRepeat\ (x{:}xs) = \textbf{if}\ x == head\ xs\ \textbf{then}\ x\ \textbf{else}\ firstRepeat\ xs.$$

(Notice that the type system is powerful enough to detect that *fix* can only be applied

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

to a function from a lattice to itself. On the other hand, it cannot capture the condition that these functions should also be monotonic.)

3.4 Products of lattices

In this section we show how to obtain instances of class *Lattice* for (finite) products of lattices, whose members are tuples of elements.

3.4.1 The implementation of tuples

We begin by describing the representation of tuples, and in particular the representation of pairs using the datatype:

$$\textbf{data}\ \mathit{Pair}\ a\,b = a\,\textbf{Pr}\,b$$

where **Pr** is an infix constructor function. Pairs will normally be written in the form (x, y) which is interpreted as $\mathit{Pair}\ x\,y$ in a type expression, and as $(x\,\textbf{Pr}\,y)$ otherwise.

The pairing constructor could be used to represent arbitrary tuples; for example, the triple (x, y, z) might be represented by the pair $((x, y), z)$. Unfortunately, this representation is not sufficiently powerful to distinguish between tuples of different sizes; for example, the expression $(x, y, z) == (u, v)$ will not necessarily be treated as a type error.

Alternative representations can be used to avoid this problem:

- One simple approach is to provide different datatypes for each different tuple size. For example, 3-tuples and 4-tuples may be represented using the types:

$$\textbf{data}\ \mathit{Tuple3}\ a\,b\,c\ \ = \mathit{Tuple3}\ a\,b\,c$$

$$\textbf{data}\ \mathit{Tuple4}\ a\,b\,c\,d = \mathit{Tuple4}\ a\,b\,c\,d$$

  and we interpret an expression of the form (x, y, z) as $\mathit{Tuple3}\ x\,y\,z$ (in both type and value expressions). This is (essentially) the approach used in Haskell, although the tuple constructors such as *Tuple3* cannot be used explicitly.

- A more unusual approach was suggested in an early draft of the Haskell report (Hudak and Wadler, 1988), and provides an interesting application of type classes in its own right. This representation uses a second pairing constructor:

$$\textbf{data}\ \mathit{Ntuple}\ \ a\,b = a\,\textbf{Tp}\,b,$$

  and we interpret $(x_1, x_2, x_3, \ldots, x_n)$ as $\mathit{Ntuple}\ (\ldots(\mathit{Ntuple}\ (\mathit{Pair}\ x_1, x_2)\,x_3)\ldots)\,x_n$ in type expressions, and as $(x_1\,\textbf{Pr}\,x_2\,\textbf{Tp}\,x_3 \ldots \textbf{Tp}\,x_n)$ otherwise, where **Pr** and **Tp** are treated as left associative infix operators. For example, a triple is represented by an element of type $\mathit{Ntuple}\ (\mathit{Pair}\ a\,b)\,c$ which cannot be unified with the type $\mathit{Pair}\ d\,e$ of an arbitrary pair, and hence the expression $(x, y, z) = (u, v)$ will cause a type error as expected.

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

Using this representation we see that the left argument of the **Tp** constructor should always be a tuple. We can enforce this condition by defining a type class *Tuple* with instances for the type constructors *Pair* and *Ntuple* as follows:

class *Tuple a*

instance *Tuple* (*Pair a b*)

instance *Tuple a* ⇒ *Tuple* (*Ntuple a b*),

and modifying the original definition of *Ntuple* to:

data *Tuple a* ⇒ *Ntuple a b* = *a* **Tp** *b*.

The predicate *Tuple a* on the left hand side restricts the choice of types which can be assigned to the type variable *a* to instances of *Tuple*. In particular, the **Tp** operator is treated as having type *Tuple a* ⇒ *a* → *b* → *Ntuple a b*. This ensures that the expression (1 **Tp** 2) is not well-typed, since *Int* is not an instance of *Tuple*.

The second representation is most convenient for the work here as it enables us to represent all tuples using only two constructed types. As a consequence, overloaded functions that can be used with tuples of arbitrary size can be defined using just two instance declarations, rather than the 'infinite family' that would be required to support arbitrarily large tuples using the first representation.

3.4.2 The lattice structure of products

The lattice structure of a product of lattices is defined by applying the lattice operators on each component of the product separately. The product of two lattices is described by the declaration:

instance (*Lattice a*, *Lattice b*) ⇒ *Lattice* (*a*, *b*) **where**

$$\begin{aligned}
\bot &= (\bot, \bot) \\
\top &= (\top, \top) \\
(x, y) \sqcap (u, v) &= (x \sqcap u, y \sqcap v) \\
(x, y) \sqcup (u, v) &= (x \sqcup u, y \sqcup v).
\end{aligned}$$

A similar instance declaration for the *Ntuple* constructor enables us to describe the lattice structure on arbitrary finite products of lattices:

instance (*Tuple a*, *Lattice a*, *Lattice b*) ⇒ *Lattice* (*Ntuple a b*) **where**

$$\begin{aligned}
\bot &= \bot \,\mathbf{Tp}\, \bot \\
\top &= \top \,\mathbf{Tp}\, \top \\
(x \,\mathbf{Tp}\, y) \sqcap (u \,\mathbf{Tp}\, v) &= (x \sqcap u) \,\mathbf{Tp}\, (y \sqcap v) \\
(x \,\mathbf{Tp}\, y) \sqcup (u \,\mathbf{Tp}\, v) &= (x \sqcup u) \,\mathbf{Tp}\, (y \sqcup v).
\end{aligned}$$

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

This gives the required componentwise definition for arbitrary tuples as illustrated by the following calculation:

$$\begin{aligned}(x, y, z) \sqcup (u, v, w) &= ((x, y)\,\mathbf{Tp}\,z) \sqcup ((u, v)\,\mathbf{Tp}\,w) \\ &= ((x, y) \sqcup (u, v))\,\mathbf{Tp}\,(z \sqcup w) \\ &= (x \sqcup u, y \sqcup v)\,\mathbf{Tp}\,(z \sqcup w) \\ &= (x \sqcup u, y \sqcup v, z \sqcup w).\end{aligned}$$

3.5 Dual lattices

The dual of a lattice a has the same points as a but with the ordering reversed; thus $x \sqsubseteq y$ in the dual lattice if and only if $y \sqsubseteq x$ in the original lattice. In the current framework, the dual of a lattice can be described using the datatype:

data *Dual a* = *Dual a*.

(Note that the symbol *Dual* is used in two distinct ways in this definition; on the left it is the name of a type constructor, on the right it is the name of a (value) constructor function used as a tag so that values of type *Dual a* can be distinguished from those of type *a*.) This representation is not ideal, since we would expect the dual of the dual of a lattice to be the original lattice, whereas the types *a* and *Dual* (*Dual a*) are clearly not equal. However, the two types are at least isomorphic, assuming that we restrict our attention to proper values (i.e. ignoring any value involving the semantic bottom value), as suggested in section 3.2.

The lattice structure of the dual of a lattice a is obtained by swapping the top and bottom elements and the join and meet operators in a as described by the instance declarations:

instance *Lattice a* ⇒ *Lattice* (*Dual a*) **where**

$$\begin{aligned}\bot &= \mathit{Dual}\ \top \\ \top &= \mathit{Dual}\ \bot \\ \mathit{Dual}\ x \sqcup \mathit{Dual}\ y &= \mathit{Dual}\ (x \sqcap y) \\ \mathit{Dual}\ x \sqcap \mathit{Dual}\ y &= \mathit{Dual}\ (x \sqcup y)\end{aligned}$$

instance *Eq a* ⇒ *Eq* (*Dual a*) **where**

$$(\mathit{Dual}\ x == \mathit{Dual}\ y) = (x == y).$$

An alternative formulation of dual lattices is described in section 8.1.

3.6 Powerset lattices

The powerset $\mathscr{P}(X)$ of an arbitrary (fixed) set X is the set of all subsets of X and forms a lattice with respect to subset inclusion. The top element in this lattice is the set X (of which every other set in $\mathscr{P}(X)$ is a subset) and the bottom element is the empty set (since it is a subset of every other set). The meet and join functions are just the

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

familiar intersection and union operators. Combining these facts, we might hope to describe the lattice structure of the powerset with an instance declaration of the form:

$$\begin{aligned}&\textbf{instance } \mathit{Lattice}\ (\mathscr{P}(X))\ \textbf{where}\\ &\quad \bot \ = \varnothing\\ &\quad \top \ = X\\ &\quad (\sqcap) = (\cap)\\ &\quad (\sqcup) = (\cup).\end{aligned}$$

While the meaning of this definition should be fairly obvious, it cannot be used in a valid Haskell program, since the set X is used both as a type (in the first line of the declaration) and as a value (in the definition of $\top$).

One way to provide a concrete implementation of the powerset lattice is to use a representation based on set expressions such as that given by the datatype:

$$\begin{aligned}\textbf{data } \mathit{Powerset}\ a = \ &\mathit{EmptySet}\\ \mid\ &\mathit{Union}\ (\mathit{Powerset}\ a)\ (\mathit{Powerset}\ a)\\ \mid\ &\mathit{Member}\ a\\ \mid\ &\mathit{Intersect}\ (\mathit{Powerset}\ a)\ (\mathit{Powerset}\ a)\\ \mid\ &\mathit{WholeSet}.\end{aligned}$$

A set expression of the form *Member* x represents the singleton set $\{x\}$ and the meaning of the remaining constructors should be clear (if not already) from the instance declaration:

$$\begin{aligned}&\textbf{instance } \mathit{Lattice}\ (\mathit{Powerset}\ a)\ \textbf{where}\\ &\quad \bot \ = \mathit{EmptySet}\\ &\quad \top \ = \mathit{WholeSet}\\ &\quad \mathit{join} \ = \mathit{Union}\\ &\quad \mathit{meet} = \mathit{Intersect}.\end{aligned}$$

The following definition shows how the membership function, written as an infix operator ($\in$), can be described by interpreting each different kind of set expression in the obvious way:

$$\begin{aligned}(\in) \qquad &:: \mathit{Eq}\ a \Rightarrow a \to \mathit{Powerset}\ a \to \mathit{Bool}\\ x \in \mathit{EmptySet} \ &= \mathit{False}\\ x \in \mathit{Union}\ l\ r \ &= x \in l \vee x \in r\\ x \in \mathit{Member}\ y \ &= x == y\\ x \in \mathit{Intersect}\ l\ r &= x \in l \wedge x \in r\\ x \in \mathit{WholeSet} \ &= \mathit{True}.\end{aligned}$$

One of the biggest problems with this representation is the task of finding an appropriate definition of the equality function between two sets which deals correctly with set expressions involving *WholeSet*, but it is beyond the scope of this paper to discuss the issue any further here. An alternative approach, suitable for representing the powersets of a certain class of lattices, will be described in section 5.4, in which

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

the lattice operations are interpreted directly rather than being treated as the constructors for a language of expressions.

4 Lattices of functions

A number of interesting lattices occur in the form of function spaces. It is particularly important to be able to manipulate the elements of these lattices in applications such as strictness analysis, where computable functions are approximated by functions on finite lattices.

4.1 The lattice structure of a function space

We begin by considering a very general case: If L is a lattice and X is an arbitrary non-empty set, then the set of functions from X to L forms a lattice whose structure is determined by pointwise application of the lattice operations on L. We might attempt to describe this by the instance declaration:

instance *Lattice* $b \Rightarrow$ *Lattice* $(a \to b)$ **where**

$$\begin{aligned} \bot x &= \bot \\ \top x &= \top \\ (f \sqcap g)x &= fx \sqcap gx \\ (f \sqcup g)x &= fx \sqcup gx. \end{aligned}$$

However, *Eq* is a superclass of *Lattice* so every instance of *Lattice* must also have an instance in *Eq*. Since there is no sensible way to define an equality operator for functions of type $(a \to b)$, it is not possible to define an instance for *Eq* $(a \to b)$ and hence the preceding instance declaration is unacceptable.

This problem can be solved by working with a concrete representation for functions rather than the functions themselves. For example, if a is a finite type with distinct elements $x_1, \ldots, x_n$ and b is an instance of *Lattice* then the lattice of functions of type $a \to b$ is isomorphic to the n-fold product of b under the correspondence which identifies each function f with the n-tuple $(fx_1, \ldots, fx_n)$. This technique can be used to represent lattices of functions, but is only practical when n is small.

4.2 An introduction to frontiers

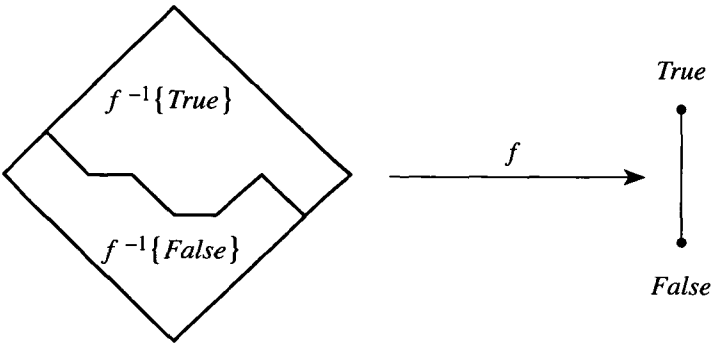
In most work with lattices, it is sufficient to restrict our attention to functions which are monotonic. Note that if a and b are instances of *Lattice*, then the set of monotonic functions of type $(a \to b)$ is a sublattice of the function space $(a \to b)$. For the most part, we consider the special case of monotonic functions of type $(a \to Bool)$ mapping the elements of a lattice a to the two point lattice of boolean values (representations for other kinds of function space are described briefly in section 4.2.6).

4.2.1 Representing functions by sets of points

Any function f of type $(a \to Bool)$ is uniquely determined by the set of points $U = f^{-1}\{True\}$ in the lattice a which it maps to *True*; given any point x in a, either $x \in U$ (in which case $fx = True$) or $x \notin U$ (in which case $fx = False$). Of course, there

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

is nothing here that requires the use of the boolean value *True*, and we could equally well represent the function f by the set of points $L = f^{-1}\{False\}$ which it maps to *False*. Note that $L = C(U)$ and $U = C(L)$, where $C(X)$ is the *complement* of X in the lattice a – in other words, the elements of a which are not members of X:



4.2.2 *Upper sets and minimum frontiers*

Using sets to represent functions as above is still only practical when working with small lattices. In the rest of this section we show how the restriction to monotonic functions can be used to obtain a more compact representation for these sets of points, and hence for the corresponding functions.

The key observation is that if x and y are distinct points in a such that $x \in U$ and $x \sqsubseteq y$, then $y \in U$; by monotonicity of f we know that $fx \sqsubseteq fy$, but $fx = True$ and hence fy must also be *True*. Subsets of a lattice with this property are called *upper sets* and the set of all upper sets in the lattice a is denoted $\mathcal{U}(a)$. Note that the empty set and the set of all elements in a are both upper sets, and that the intersection and union of any two upper sets is again an upper set. Hence $\mathcal{U}(a)$ is a lattice (and a sublattice of $\mathcal{P}(a)$). In fact, $\mathcal{U}(a)$ is isomorphic to the lattice of monotonic functions of type $(a \rightarrow Bool)$ under the correspondence which maps each function f to the upper set $f^{-1}\{True\}$; it is straightforward to show that this is a bijection, and the result follows by establishing the identities:

$$(f \sqcup g)^{-1}\{True\} = f^{-1}\{True\} \cup g^{-1}\{True\}$$
$$(f \sqcap g)^{-1}\{True\} = f^{-1}\{True\} \cap g^{-1}\{True\}$$

and from the fact that $\bot^{-1}\{True\} = \{\}$, the bottom element of $\mathcal{U}(a)$, while $\top^{-1}\{True\}$ is the set of all points in a, the top element in $\mathcal{U}(a)$. Thus upper sets provide a representation for monotonic functions and calculations in that lattice can be described by equivalent calculations in the lattice of upper sets.

The advantage of representing a monotonic function by an upper set is that we do not usually need to store all of the elements of the set. So long as we restrict our attention to finite lattices, it is sufficient to keep just the *minimal* elements of an upper set U, denoted $min(U)$ – i.e. those elements $x \in U$ such that $y \in U$ and $y \sqsubseteq x$ implies that $x = y$. Sets of this kind can be thought of as representing the dividing line between the two subsets $f^{-1}\{True\}$ and $f^{-1}\{False\}$ of the lattice a and are often referred

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

to as *frontiers*, using terminology introduced by Clack and Peyton Jones (1985). Frontier sets can also be characterized directly as the subsets of a lattice whose elements are pairwise incomparable, and we write $\mathscr{F}(a)$ for the set of all frontiers in a lattice a.

With this notation the *min* operator is a bijection:

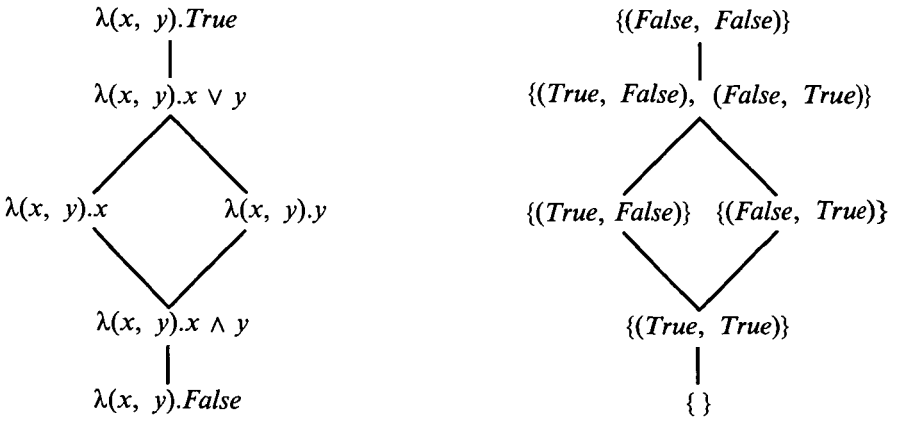
$$min :: \mathscr{U}(a) \to \mathscr{F}(a)$$

and the corresponding inverse is the *upward closure* operator:

$$(\uparrow) :: \mathscr{F}(a) \to \mathscr{U}(a)$$

where $\uparrow F = \{y \mid x \in F, x \sqsubseteq y\}$ is the smallest upper set containing F. Together, these operators enable us to use frontiers to represent upper sets, and hence to represent monotonic functions. In the next section we show how to define a lattice structure on $\mathscr{F}(a)$ which makes *min* and $(\uparrow)$ into lattice isomorphisms, in which case we describe the elements of $\mathscr{F}(a)$ as *minimum frontiers*. The correspondence between the lattice of monotonic functions and the lattice of minimum frontiers enables us to describe calculations in the former by equivalent calculations in the latter.

As a concrete illustration of the use of frontiers, the following diagram shows the lattice of monotonic functions of type $(Bool, Bool) \to Bool$ together with the corresponding lattice of minimum frontiers:



(As an aside, we should mention that both *min* and $(\uparrow)$ can be applied to arbitrary subsets of the lattice a using the definitions given above. It would therefore be possible to treat these operators as having types $\mathscr{P}(a) \to \mathscr{F}(a)$ and $\mathscr{P}(a) \to \mathscr{U}(a)$ respectively, but neither operator is a bijection using these more general types.)

4.2.3 The lattice of minimum frontiers

For the purposes of this paper we choose a representation for minimum frontiers as values of type:

$$\textbf{data}\ Minf\ a = Minf\ [a],$$

using a list to specify the elements in the frontier set. Given an arbitrary list of points

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

in a lattice, we can find the set of minimal elements (represented by a minimum frontier) using the following implementation of *min*:

$$\begin{aligned}
&\mathit{min} && :: \mathit{Lattice}\ a \Rightarrow [a] \to \mathit{Minf}\ a\\
&\mathit{min} && = \mathit{Minf}\,.\,\mathit{minimals}\,.\,\mathit{nub}\\
&\mathit{minimals} && :: \mathit{Lattice}\ a \Rightarrow [a] \to [a]\\
&\mathit{minimals}\ xs && = \mathit{filter}\ (\mathit{null}\,.\,\mathit{pointsBelow})\ xs\\
& && \quad \textbf{where}\ \mathit{pointsBelow}\ x = [y \mid y \leftarrow xs, y \sqsubseteq x, x \neq y].
\end{aligned}$$

There is a unique lattice structure on the set of minimum frontiers such that *min* and $(\uparrow)$ are lattice isomorphisms between $\mathscr{F}(a)$ and $\mathscr{U}(a)$ – each of the lattice operations on frontiers must be defined so that its image under $(\uparrow)$ is the corresponding lattice operation on upper sets:

- The smallest upper set is the empty set – the upward closure of the empty minimum frontier $\bot = \mathit{Minf}\ [\,]$.
- The largest upper set in a lattice a is the lattice itself, which has least element $\bot$, and is therefore represented by the minimum frontier $\top = \mathit{Minf}\ [\bot]$.
- Suppose that x and y are minimum frontiers (representing the upper sets $\uparrow x$ and $\uparrow y$ respectively). To ensure that $(\uparrow)$ is a lattice isomorphism, the meet of x and y must be defined so that $\uparrow(x \sqcap y) = (\uparrow xs) \cap (\uparrow ys)$. Notice that:

$$\begin{aligned}
z \in (\uparrow xs) \cap (\uparrow ys) &\Leftrightarrow (\exists x \in xs\,.\,x \sqsubseteq z) \wedge (\exists y \in ys\,.\,y \sqsubseteq z)\\
&\Leftrightarrow \exists x \in xs\,.\,\exists y \in ys\,.\,x \sqcup y \sqsubseteq z\\
&\Leftrightarrow z \in \uparrow \{x \sqcup y \mid x \in xs, y \in ys\}
\end{aligned}$$

  so that a suitable definition for $(\sqcap)$ is provided by:

$$\mathit{Minf}\ xs \sqcap \mathit{Minf}\ ys = \mathit{min}[x \sqcup y \mid x \leftarrow xs, y \leftarrow ys].$$

- In a similar way, the join of two minimum frontiers x and y must be defined so that $\uparrow(x \sqcup y) = (\uparrow xs) \cup (\uparrow ys)$. It follows from the definition of *min* that the $(\sqcup)$ function can be defined using:

$$\mathit{Minf}\ xs \sqcup \mathit{Minf}\ ys = \mathit{min}\,(xs +\!\!+ ys)$$

  where the list $(xs +\!\!+ ys)$ represents the set of points in the union of xs and ys.

These observations motivate the following instance declaration:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Lattice}\ a \Rightarrow \mathit{Lattice}\ (\mathit{Minf}\ a)\ \textbf{where}\\
&\quad \bot && = \mathit{Minf}\ [\,]\\
&\quad \top && = \mathit{Minf}\ [\bot]\\
&\quad \mathit{Minf}\ xs \sqcap \mathit{Minf}\ ys && = \mathit{min}[x \sqcup y \mid x \leftarrow xs, y \leftarrow ys]\\
&\quad \mathit{Minf}\ xs \sqcup \mathit{Minf}\ ts && = \mathit{min}(xs +\!\!+ ys).
\end{aligned}$$

As before, each instance of *Lattice* must also have an instance in the class *Eq*.

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

Remembering that minimum frontiers represent sets of elements and not simply lists, we define:

$$\textbf{instance } Eq\ a \Rightarrow Eq\ (Minf\ a) \textbf{ where}$$
$$Minf\ xs == Minf\ ys = (xs \subseteq ys) \wedge (ys \subseteq xs).$$

4.2.4 Lower sets and maximum frontiers

The representation of functions by minimum frontiers in the last two sections was motivated by identifying functions with upper sets of the form $f^{-1}\{True\}$. We can also obtain a dual representation motivated by identifying each function f with the set of points $f^{-1}\{False\}$, but otherwise following the same pattern as before.

Assuming that f is monotonic, any set of the form $L = f^{-1}\{False\}$ has the property that if $y \in L$ and $x \sqsubseteq y$, then $x \in L$, and we therefore refer to L as a *lower set*. The set of all lower sets in a lattice a is denoted by $\mathscr{L}(a)$ and is a sublattice of $\mathscr{P}(a)$.

The correspondence which maps any function f to the lower set $f^{-1}\{False\}$ is a bijection from the lattice of monotonic functions of type $(a \to Bool)$ to $\mathscr{L}(a)$, but it is not quite a lattice isomorphism, since it reverses the ordering between elements:

$$f \sqsubseteq g \Leftrightarrow g^{-1}\{False\} \subseteq f^{-1}\{False\}.$$

As a result, joins and meets in the lattice of monotonic functions correspond to meets and joins, respectively, in $\mathscr{L}(a)$:

$$(f \sqcup g)^{-1}\{False\} = f^{-1}\{False\} \cap g^{-1}\{False\}$$
$$(f \sqcap g)^{-1}\{False\} = f^{-1}\{False\} \cup g^{-1}\{False\}.$$

It follows that the lattice of monotonic functions of type $(a \to Bool)$ is isomorphic to the dual of $\mathscr{L}(a)$. We have already shown that this lattice of functions is isomorphic to $\mathscr{U}(a)$, and hence each of the lattices $\mathscr{L}(a)$ and $\mathscr{U}(a)$ is isomorphic to the dual of the other. In both cases, the isomorphism is the complement function C which maps upper sets to lower sets, and *vice versa*.

Just as the upper sets in a finite lattice are uniquely determined by their minimal elements, every lower set in a finite lattice is uniquely determined by its *maximal* elements which can be obtained using the bijection:

$$max :: \mathscr{L}(a) \to \mathscr{F}(a)$$

where max (L) is the set of all elements $x \in L$ such that $y \in L$ and $x \sqsubseteq y$ implies that $x = y$. The inverse function is the *downward closure* operator:

$$(\downarrow) :: \mathscr{F}(a) \to \mathscr{L}(a)$$

where $\downarrow F = \{y \mid x \in F, y \sqsubseteq x\}$ is the smallest lower set containing F.

4.2.5 The lattice of maximum frontiers

As in section 4.2.3, there is a unique lattice structure on $\mathscr{F}(a)$ such that the *max* and $(\downarrow)$ operators introduced above are lattice isomorphisms, and we refer to frontiers used in this way as *maximum frontiers*. It follows from the duality between $\mathscr{U}(a)$ and

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

$\mathscr{L}(a)$ that the lattice of maximum frontiers is isomorphic to the dual of the lattice of minimum frontiers.

In our current framework, maximum frontiers will be represented by elements of type:

$$\textbf{data}\ Maxf\ a = Maxf\ [a].$$

The following implementation for *max* is the obvious dual of the definition of *min* in section 4.2.3:

$$\begin{aligned} max &:: Lattice\ a \Rightarrow [a] \to Maxf\ a \\ max &= Maxf . maximals . nub \\ maximals &:: Lattice\ a \Rightarrow [a] \to [a] \\ maximals\ xs &= filter\ (null . pointsAbove)\ xs \\ &\quad \textbf{where}\ pointsAbove\ x = [y \mid y \leftarrow xs, x \sqsubseteq y, x \neq y]. \end{aligned}$$

In a similar way, the lattice of maximum frontiers can be described by dualizing the definition of the lattice of minimum frontiers:

$$\begin{aligned} &\textbf{instance}\ Lattice\ a \Rightarrow Lattice\ (Maxf\ a)\ \textbf{where} \\ &\quad \bot = Maxf\ [] \\ &\quad \top = Maxf\ [\top] \\ &\quad Maxf\ xs \sqcap Maxf\ ys = max[x \sqcap y \mid x \leftarrow xs, y \leftarrow ys] \\ &\quad Maxf\ xs \sqcup Maxf\ ys = max\ (xs +\!\!+ ys). \end{aligned}$$

A simple definition makes $Maxf\ a$ an instance of *Eq*:

$$\begin{aligned} &\textbf{instance}\ Eq\ a \Rightarrow Eq\ (Maxf\ a)\ \textbf{where} \\ &\quad Maxf\ xs == Maxf\ ys = (xs \sqsubseteq ys) \wedge (ys \sqsubseteq xs). \end{aligned}$$

4.2.6 *Lattices of monotonic functions*

Combining the results of the preceding sections we obtain a representation for (monotonic) functions using either minimum or maximum frontiers described by the bijections:

$$\begin{aligned} minApply &:: Minf\ a \to (a \to Bool) \\ minApply\ (Minf\ xs)\ x &= any\ (\sqsubseteq x)\ xs \\ maxApply &:: Maxf\ a \to (a \to Bool) \\ maxApply\ (Maxf\ xs)\ x &= not\ (any\ (x \sqsubseteq)\ xs). \end{aligned}$$

The corresponding inverse functions can be specified using:

$$\begin{aligned} minFront &:: (a \to Bool) \to Minf\ a \\ minFront\ f &= min(f^{-1}\{True\}) \\ maxFront &:: (a \to Bool) \to Maxf\ a \\ maxFront\ f &= max(f^{-1}\{False\}). \end{aligned}$$

Note that these equations cannot be used as executable definitions, since there is no

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

general method for calculating the inverse image $f^{-1}\{x\}$ of a point x under a function f. However, a concrete implementation of *minFront* and *maxFront* which is derived directly from these specifications will be given in section 5.4. The resulting algorithm is rather inefficient, as it relies on the ability to enumerate the elements of the source lattice for the function concerned. A number of researchers have suggested more efficient algorithms which use the fact that these functions are monotonic to speed up the calculation (Peyton Jones and Clack, 1987; Martin and Hankin 1987; Hunt, 1989). Although beyond the scope of this paper, it is interesting to note that each of the algorithms suggested in these papers can be described, and hence compared, in a uniform manner using the tools provided here (Jones, 1989).

In practice, it is convenient to represent a function of type $(a \to Bool)$ by both its minimum and maximum frontiers using a value of type:

$$\textbf{data}\ Fn\,a = Fn\,(Minf\,a)\,(Maxf\,a).$$

This representation uses redundant information; the values of u and l in an expression of the form $Fn\,u\,l$ should be related by the equations $u = min\,(C(\downarrow l))$ and $l = max\,(C(\uparrow u))$. The reason that we have chosen to keep both is that some functions are easier to describe using minimum frontiers, while others which are easier to describe using maximum frontiers (see the definitions of *succs* and *preds* for values of type $Fn\,a$ given in section 5.1 for an example of this). Furthermore, several of the algorithms mentioned above for obtaining the frontier representation of a function $f :: a \to Bool$ actually produce values for both frontiers using a combined search, and hence there is no additional cost in finding the corresponding element of type $Fn\,a$.

The lattice structure of $Fn\,a$ is easily described in terms of the lattice structures for $Minf\,a$ and (the dual of) $Maxf\,a$:

$$\begin{aligned}
&\textbf{instance}\ Lattice\ a \Rightarrow Lattice\ (Fn\,a)\ \textbf{where}\\
&\bot \qquad\qquad\quad = Fn\,\bot\,\top\\
&\top \qquad\qquad\quad = Fn\,\top\,\bot\\
&Fn\,u\,l \sqcap Fn\,u'\,l' = Fn\,(u \sqcap u')\,(l \sqcup l')\\
&Fn\,u\,l \sqcup Fn\,u'\,l' = Fn\,(u \sqcup u')\,(l \sqcap l').
\end{aligned}$$

So far we have considered only lattices of monotonic functions of type $(a \to Bool)$. Other kinds of function space can often be described in terms of lattices of this form. For example, using the familiar isomorphisms:

$$\begin{aligned}
a \to (b, c) &\cong (a \to b, a \to c)\\
a \to b \to c &\cong (a, b) \to c
\end{aligned}$$

we can represent monotonic functions of type $Bool \to (Bool, Bool)$ and $Bool \to Bool \to Bool$ by values in the lattices $(Fn\,Bool, Fn\,Bool)$ and $Fn\,(Bool, Bool)$, respectively. These techniques can be used to represent the lattice of monotonic functions for any type $(a \to b)$ in which a is an arbitrary instance of *Lattice* and b is constructed from the type *Bool* and an arbitrary combination of product and (monotonic) function space constructors.

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

5 Navigable lattices

5.1 A subclass of lattices

Much of the work in Jones (1989) dealt with algorithms to determine the maximum and minimum frontiers of a function. A typical algorithm begins with an approximation to the frontier that it is required to find, which is then refined by testing the value of the function at nearby points and adjusting the approximation accordingly. This requires some means of 'moving around' a lattice to locate suitable nearby points. In the following sections, we show how this can be described using the functions *succs* and *preds* specified by:

$$succs\ x = min\,(C(\downarrow\{x\}))$$
$$preds\ x = max\,(C(\uparrow\{x\})).$$

These functions can be defined for any lattice, but in practice will not be needed for some instances of *Lattice*. Rather than modifying the definition of *Lattice* to include *preds* and *succs* as member functions, we introduce a new subclass of *Lattice* with the definition:

```
class Lattice a ⇒ Navigable a where
    succs :: a → Minf a
    preds :: a → Maxf a.
```

This approach gives greater modularity and frees us from the need to define *succs* and *preds* except in those instances where these functions will actually be required.

Some instances of the class *Navigable* are given in the following definitions. The definition of the instance for (*Fn a*) is based on work by Hunt (1989). Further details are given in Jones (1989):

```
instance Navigable Bool where
    succs False = Minf [True]
    succs True  = Minf []
    preds False = Maxf []
    preds True  = Maxf [False]

instance (Navigable a, Navigable b) ⇒ Navigable (a, b) where
    succs (x, y) = Minf ([sx, ⊥) | sx ← sxs] ++ [(⊥, sy) | sy ← sys])
                   where Minf sxs = succs x
                         Minf sys = succs y
    preds (x, y) = Maxf ([(px, ⊤) | px ← pxs] ++ [(⊤, py) | py ← pys])
                   where Maxf pxs = preds x
                         Maxf pys = preds y

instance Navigable a ⇒ Navigable (Fn a) where
    succs (Fn u (Maxf ps)) = Minf [Fn (Minf [p]) (preds p) | p ← ps]
    preds (Fn (Minf ps) l) = Maxf [Fn (succs p) (Maxf [p]) | p ← ps].
```

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

5.2 Implementations of complement

As a simple application of the member functions of *Navigable*, we can describe the effect of the complement operator on upper sets (represented by minimum frontiers) and lower sets (represented by maximum frontiers). For example, if X is an upper set then:

$$\begin{aligned}
C(X) &= C(\bigcup\{\uparrow\{x\} \mid x \in X\}) && X \text{ is an upper set}\\
&= \bigcap\{C(\uparrow\{x\}) \mid x \in X\} && \text{de Morgan's law}\\
&= \bigcap\{\downarrow(max(C(\uparrow\{x\}))) \mid x \in X\} && (\downarrow).max \text{ is identity on lower sets}\\
&= \bigcap\{\downarrow(preds\ x) \mid x \in X\} && \text{definition of } preds\\
&= \downarrow(\sqcap\{preds\ x \mid x \in X\}) && \text{lattice structure on maximum frontiers}
\end{aligned}$$

so that if f is a minimum frontier representing X then the maximum frontier representing $C(X)$ is $maxComb\ f$ where:

$$\begin{aligned}
&maxComp :: Navigable\ a \Rightarrow Minf\ a \rightarrow Maxf\ a\\
&maxComp = foldr(\sqcap)\top . map\ preds.
\end{aligned}$$

A similar argument motivates the definition of *minComp* which can be used to calculate the minimum frontier of the complement of a lower set represented by a maximum frontier:

$$\begin{aligned}
&minComp :: Navigable\ a \Rightarrow Maxf\ a \rightarrow Minf\ a\\
&minComp = foldr(\sqcap)\top . map\ succs.
\end{aligned}$$

5.3 An implementation of upward closure

The member functions of *Navigable* can also be used to implement a form of the upward closure operator. This provides a simple illustration of the way in which the member functions of *Navigable* can be used to search a lattice. For simplicity, we will only consider the task of computing the upward closure of a minimum frontier, although this restriction is easily relaxed by composing with *min*.

Suppose that f is a minimum frontier containing elements of some lattice a. If f is empty then so is $\uparrow f$. Otherwise, we can pick some element t in f and enumerate the elements of $\uparrow f$ by outputting t and then enumerating the elements of:

$$\begin{aligned}
(\uparrow f)\backslash\{t\} &= (\uparrow f) \cap C(\{t\}) && \text{standard set theory}\\
&= (\uparrow f) \cap C(\downarrow\{t\}) && t \text{ minimal in } \uparrow f\\
&= (\uparrow f) \cap (\uparrow(min(C(\downarrow\{t\})))) && (\uparrow).min \text{ is identity on upper sets}\\
&= (\uparrow f) \cap (\uparrow(succs\ t)) && \text{definition of } succs\\
&= \uparrow(f \sqcap succs\ t) && \text{lattice structure on minimum frontiers.}
\end{aligned}$$

This description can be translated directly into the following function definition:

$$\begin{aligned}
&(\uparrow) && :: Navigable\ a \Rightarrow Minf\ a \rightarrow [a]\\
&\uparrow(Minf[\,]) && = [\,]\\
&\uparrow(Minf(t:ts)) && = t:\uparrow(Minf(t:ts) \sqcap succs\ t).
\end{aligned}$$

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

5.4 Enumerating the elements of a lattice

The implementation of (↑) in the previous section gives us an easy way to enumerate the elements of an arbitrary navigable lattice; we simply calculate the upward closure of the minimum frontier $\{\bot\}$:

$$\begin{aligned} &elements :: Navigable\ a \Rightarrow [a] \\ &elements = \uparrow[\bot]. \end{aligned}$$

Notice that the type of the expression *elements* does not specify which instance of *Navigable* is to be used in place of the type variable *a*. In Haskell, this ambiguity can be resolved by an expression of the form *elements*::*type* where the required instance is determined by the type expression *type*. For example:

- *elements*::[*Bool*] produces the list [*False*, *True*] of boolean values.
- *elements*::[(*Bool*, *Bool*)] produces a list of the four points in the lattice (*Bool*, *Bool*).
- *elements*::[*Fn* (*Bool*, *Bool*)] produces a list of six values representing the monotonic functions from (*Bool*, *Bool*) to *Bool*.

Those familiar with Stoy (1977) may recall one section which defines a sequence of lattices:

$$\begin{aligned} D_0 &= Bool \\ D_{n+1} &= D_n \to D_n. \end{aligned}$$

Stoy claims that D_3 has 120,549 elements, and suggests that his reader writes a program to verify this. Expanding the definition of D_3 gives:

$$\begin{aligned} D_3 &= ((Bool \to Bool) \to (Bool \to Bool)) \to ((Bool \to Bool) \to (Bool \to Bool)) \\ &\cong ((Bool \to Bool, Bool) \to Bool, Bool \to Bool, Bool) \to Bool \\ &\cong Fn\,(Fn\,(Fn\ Bool, Bool), Fn\ Bool, Bool) \end{aligned}$$

so that a suitable program, using the tools developed in this paper is:

$$length(elements::[D_3]) == 120549.$$

As another simple application of *elements*, the specifications for the functions *minFront* and *maxFront*, given in section 4.2.6, can now be translated almost directly into executable definitions:

$$\begin{aligned} &minFront\ \ :: (a \to Bool) \to Minf\ a \\ &minFront\ f\ = min\,[x \mid x \leftarrow elements, f\,x] \\ &maxFront\ \ :: (a \to Bool) \to Maxf\ a \\ &maxFront\ f = max\,[x \mid x \leftarrow elements, not\ (f\,x)]. \end{aligned}$$

Note, however, that this implementation is very inefficient, and a more sophisticated algorithm would be required in any practical application as described in section 4.2.6.

We can also use *elements* to provide a representation for the powerset of an

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

arbitrary navigable lattice, using lists of values to describe the elements of each subset and the standard definition of set equality:

$$\textbf{data}\ \mathit{Set}\ a = \mathit{Set}\ [a]$$

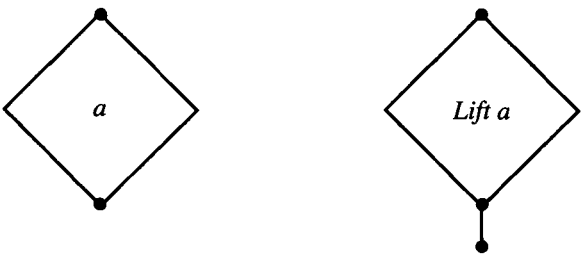
$$\textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ (\mathit{Set}\ a)\ \textbf{where}$$
$$(x == y) = (x \subseteq y) \wedge (y \subseteq x).$$

The powerset lattice can then be described by the instance declaration:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Navigable}\ a \Rightarrow \mathit{Lattice}\ (\mathit{Set}\ a)\ \textbf{where}\\
&\bot \qquad\qquad = \mathit{Set}\ [\,]\\
&\top \qquad\qquad = \mathit{Set}\ \mathit{elements}\\
&\mathit{Set}\ xs \sqcup \mathit{Set}\ ys = \mathit{Set}\ (xs +\!\!+ ys)\\
&\mathit{Set}\ xs \sqcap \mathit{Set}\ ys = \mathit{Set}\ [x \mid x \leftarrow xs,\ x\ \textbf{elem}\ ys].
\end{aligned}$$

6 Adding new instances

The use of type classes makes it easy to extend the framework described in the previous sections to include other kinds of lattice. Suppose, for example, that we want to work with *lifted* lattices:



The lifted version, *Lift a*, of a lattice *a* contains a copy of *a* with an additional element, *Bottom*, below all the members of *a*. The elements of the lifted lattice are represented by elements of type:

$$\textbf{data}\ \mathit{Lift}\ a = \mathit{Bottom} \mid \mathit{Up}\ a$$

The definition of the lattice operations on a lifted lattice is standard and is described by the instance declaration:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Lattice}\ a \Rightarrow \mathit{Lattice}\ (\mathit{Lift}\ a)\ \textbf{where}\\
&\bot \qquad\qquad = \mathit{Bottom}\\
&\top \qquad\qquad = \mathit{Up}\ \top\\
&\mathit{Bottom} \sqcap y \qquad = \mathit{Bottom}\\
&\mathit{Up}\ x \sqcap \mathit{Bottom} = \mathit{Bottom}\\
&\mathit{Up}\ x \sqcap \mathit{Up}\ y \qquad = \mathit{Up}\ (x \sqcap y)\\
&\mathit{Bottom} \sqcup y \qquad = y\\
&\mathit{Up}\ x \sqcup \mathit{Bottom} = \mathit{Up}\ x\\
&\mathit{Up}\ x \sqcup \mathit{Up}\ y \qquad = \mathit{Up}\ (x \sqcup y).
\end{aligned}$$

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

A further declaration makes *Lift a* an instance of *Navigable*; its definition is a simple consequence of the specification of *preds* and *succs*, and the lattice structure of *Lift a*:

```
instance Navigable a ⇒ Navigable (Lift a) where
  succs Bottom          = [Up ⊥]
  succs (Up x)          = [Up y | y ← succs x]
  preds Bottom          = []
  preds (Up x) | x == ⊥ = [Bottom]
               | otherwise = [Up y | y ← preds x].
```

It is easy to extend the framework still further with additional instance declarations; simple examples include separated and coalesced sums, flat domains, and a number of less standard lattice constructions. A remarkable degree of modularity has been achieved in that each of these extensions can be defined and used completely independently of any of the others. The complete program is easier to understand because each instance declaration may be understood separately in its own right; the only link with other instance declarations is the original class declaration.

7 Comparison with previous implementation

This section describes an earlier implementation of the ideas presented in this paper, highlighting a number of deficiencies caused by the use of a standard Hindley/Milner type system. Each of these problems has been eliminated from the version described in this paper by the use of type classes. Although the original implementation used Orwell (Wadler and Miller, 1988), the code fragments given in this section have been rewritten using the notation of Haskell, without the use of type classes.

7.1 Representing the elements of a lattice

An important requirement of the original implementation was that the lattice operators be defined by single functions:

```
(⊑)      :: Element → Element → Bool
(⊓), (⊔) :: Element → Element → Element
```

for some type *Element*. To be able to use these operators with a range of lattices, including lattices of tuples and functions, the use of a Hindley/Milner type system requires that *Element* be a type containing a common instance of a representation of each of these types. This was achieved by defining *Element* as a algebraic datatype:

```
data Element = O | I | Tup [Element] | Fn Minf Maxf
type Frontier = [Element]
type Minf     = Frontier
type Maxf     = Frontier
```

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

where the objects of type *Element* are interpreted as follows:

- The elements O and I represent the points of a two point lattice with $O \sqsubseteq I$, and correspond to the boolean values *False* and *True* in the present implementation.
- An element of the form *Tup xs* represents the tuple of elements in the list *xs*. Lists are used to avoid the need for separate constructor functions for pairs, triples, quadruples, etc.
- As in the current program, functions are represented by a pair of frontiers each of which is represented by a simple list of elements.

The names of the types [*Element*], *Frontier*, *Minf* and *Maxf* were used in type declarations to indicate the intended use of particular function arguments and return values. There is no way to ensure that all of the values in a frontier come from the same lattice with this representation; for example, $[O, Tup\,[O\,I]]$ is a well-typed expression of type [*Element*], but is meaningless as an element of type *Frontier*. To avoid further complication of the definition, there is no representation for individual frontiers as objects of *Element*. This also means that the functions *preds* and *succs* can be defined on each object of *Element* without needing to implement these functions on frontiers. This corresponds to the use of two classes, *Lattice* and *Navigable*, in the present implementation to describe the separation between the definition of a lattice structure on the elements of some type and the implementation of the functions *succs* and *preds* on that lattice structure.

7.2 Implementing the lattice operations

Each of the lattice operations was implemented using pattern matching on objects of type *Element*. For example, the $(\sqcup)$ operation was defined by:

$$
\begin{aligned}
&(\sqcup) && :: Element \to Element \to Element\\
&O \sqcup x && = x\\
&I \sqcup x && = I\\
&(Tup\ xs) \sqcup (Tup\ ys) && = Tup\ (zipWith\ (\sqcup)\ xs\ ys)\\
&(Fn\ lf1\ lf0) \sqcup (Fn\ rf1\ rf0) && = Fn\ (lf1 \sqcup^{mtn} rf1)\ (lf0 \sqcap^{max} rf0).
\end{aligned}
$$

The restriction that x and y be elements of the same lattice in an expression of the form $x \sqcup y$ cannot be enforced by the type checker; each of the expressions $O \sqcup Tup\,[O, I]$ and $Tup\,[O, I] \sqcup O$ is well-typed. Notice that the first expression produces a well-defined value, while the second causes a run-time error because there is no matching case in the definition of $(\sqcup)$.

As the type *Element* does not include a representation for frontiers, the lattice structure of minimum and maximum frontiers cannot be described using the standard lattice operators such as $(\sqcup)$. This forced us to define additional operators, such as $(\sqcup^{mtn})$ and $(\sqcup^{max})$ to implement lattice operations on frontiers.

The implementation of some functions, in particular $\bot$ and $\top$, required the use of an additional type:

$$\textbf{data}\ Lattice = Two \mid Prod\,[Lattice] \mid Func\ Lattice$$

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

whose elements can be used to name each of the lattices whose members are represented by objects of *Element*:

- *Two* is a name for the two point lattice containing *O* and *I*.
- *Prod ls* is a name for the product of the lattices listed by *ls*.
- *Func l* is a name for the lattice of monotonic functions from the lattice named by *l* to the lattice named *Two*.

For example, the bottom element of each *Lattice* can be obtained using the function:

$$\begin{aligned} \bot \quad & :: Lattice \to Element \\ \bot\ Two \quad & = O \\ \bot\ (Prod\ ls) & = Tup\,[\bot\ l \mid l \leftarrow ls] \\ \bot\ (Func\ l) & = Fn\,[\,][\top\ l]. \end{aligned}$$

In this way, the elements of *Lattice* can be thought of as naming the lattices on which a number of functions are defined, i.e. they correspond to names for the instances of a type class.

7.3 Weaknesses due to Hindley/Milner typing

The previous sections have described a number of weaknesses in the original implementation, including:

- The use of 'artificial' type definitions to represent the elements of types which have already been defined by the system. For example, the use of the constructors *O* and *I* to represent boolean values, and of expressions of the form *Tup xs* to represent tuples.
- The inability to detect some kinds of 'type error'. In some cases these errors will not be detected, whilst others will cause runtime errors.
- The use of a number of distinct names (for example $(\sqcup)$, $(\sqcup^{min})$ and $(\sqcup^{max})$) for a single concept.
- The need to define types such as *Lattice* whose elements represent types (or more accurately in the present example, subtypes of *Element*).
- The layout of the program is confusing; the definitions of the lattice operations on the elements of each individual *Lattice* are distributed throughout the program.
- The program is difficult to extend; each additional *Lattice* constructor (for example, lifting) requires a modification of the definition of *Element* and a change to each of the definitions of the main lattice operations, which grow increasingly long and unwieldy.

The source of each of these problems can be traced to the constraints imposed on a programmer by the use of a Hindley/Milner type system. Thanks to the use of type classes, none of these problems occurs in the present implementation.

8 Future work

One of the objectives in this paper has been to describe a framework for computing with the elements of finite lattices. The biggest difficulty with this particular application is the problem of finding a concrete representation for functions. We have

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

described a compact representation for a reasonably large class of monotonic functions based on the use of frontiers, but it would certainly be desirable to extend this to include other kinds of function space.

A second objective has been to explore the use of type classes. In particular, we have shown how this leads to a modular program which avoids some of the problems that occur using a similar program in a language based on the standard Hindley/Milner type system. The remaining sections describe two small extensions to the system of type classes provided in languages such as Haskell which could potentially increase the usefulness of type classes in program development. Both of these ideas were originally suggested in Wadler and Blott (1989), and the examples given here provide further motivation for supporting these features in practical implementations.

8.1 Classes with multiple parameters

A number of researchers have suggested generalizations to the system of type classes in Haskell which allow classes with multiple parameters. A typical application would be to describe duality in lattices, where the dual of a given lattice has the same elements (possibly with a different representation), but the opposite partial order. Another way to describe this is with a *complement* function which is an order isomorphism between each lattice and its dual. With this in mind, a suitable class definition might be:

$$\textbf{class}\ (\mathit{Lattice}\ a, \mathit{Lattice}\ b, \mathit{Dual}\ b\ a) \Rightarrow \mathit{Dual}\ a\ b\ \textbf{where}$$
$$\quad \mathit{comp} :: a \to b.$$

Some simple instances of *Dual* are given by the declarations:

$$\textbf{instance}\ \mathit{Dual}\ \mathit{Bool}\ \mathit{Bool}\ \textbf{where}$$
$$\quad \mathit{comp} = \mathit{not}$$
$$\textbf{instance}\ (\mathit{Dual}\ a\ c, \mathit{Dual}\ b\ d) \Rightarrow \mathit{Dual}\ (a,b)\ (c,d)\ \textbf{where}$$
$$\quad \mathit{comp}\ (x,y) = (\mathit{comp}\ x, \mathit{comp}\ y)$$
$$\textbf{instance}\ \mathit{Navigable}\ a \Rightarrow \mathit{Dual}\ (\mathit{Minf}\ a)\ (\mathit{Maxf}\ a)\ \textbf{where}$$
$$\quad \mathit{comp} = \mathit{maxComp}$$
$$\textbf{instance}\ \mathit{Navigable}\ a \Rightarrow \mathit{Dual}\ (\mathit{Maxf}\ a)\ (\mathit{Minf}\ a)\ \textbf{where}$$
$$\quad \mathit{comp} = \mathit{minComp}.$$

There are no strong theoretical problems which make the treatment of multiple parameter type classes particularly difficult. Whilst useful as a means of describing relationships between types, our experience to date (see Jones, 1991, for example) suggests that practical applications of such classes may be rather limited. To give an indication of the kind of problems which can occur, the principal type of the expression *comp*.*comp* (which we might (fairly reasonably) hope would denote an identity function of type $\mathit{Lattice}\ a \Rightarrow a \to a$) is:

$$(\mathit{Dual}\ a\ b, \mathit{Dual}\ b\ c) \Rightarrow a \to c.$$

Apart from the fact that the domain and range types are not the same, this typing is *ambiguous* (in the sense that the type variable *b* appears in a predicate but not in the

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

main body of the type) and it can then be shown that the expression *comp . comp* does not have a well-defined semantics.

8.2 The use of program laws

Whilst the treatment of computable equality (==) as a built-in primitive has a number of disadvantages (listed in section 2.1), it also has the significant advantage of ensuring that the function has a well-defined behaviour for all argument types. This can be exploited in program development and transformation (assuming of course that the implementation of the primitive is correct).

By contrast, in a system of type classes it is not possible to place any semantic restriction on the definitions given in any particular instance other than ensuring that they yield values of the correct types. Furthermore, in the development of a large program, the instance declarations used to construct the definition of a single overloaded operator may be distributed across a number of separate program modules. This makes it very difficult for a programmer to know what properties can be assumed about overloaded functions, and prevents the use of simple equational reasoning, often cited as one of the most important benefits of using functional languages.

One approach to this problem is to adopt a programming methodology in which:

- Each class declaration is accompanied by a number of algebraic laws constraining the values of its member functions.
- Each instance declaration is accompanied by a proof of the laws in the particular instance being defined.

Such laws can of course be written in the form of comments, but it might be preferable to extend the syntax of the language with a concrete syntax for laws:

- Programmers would be encouraged to state laws formally using a uniform syntax, rather than a variety of *ad hoc* annotations.
- The type checker can be used to ensure that the laws given are type correct, and hence detect some meaningless or erroneous laws.
- It is unlikely that the proofs for each law could be constructed automatically for each instance declaration. On the other hand, machine readable laws in a given program might well be used in conjunction with an automated proof checker, or with the machine assisted tools for program derivation and proof.

The following example illustrates one possible syntax for writing laws about the ($\sqcap$) operator in the class *Lattice*, introducing a name and listing the free variables for each law:

$$\mathit{MeetIdem}\ x \quad \Rightarrow x \sqcap x = x$$
$$\mathit{MeetSymm}\ x\,y \Rightarrow x \sqcap y = y \sqcap x.$$

Unfortunately, the task of choosing appropriate laws in a non-strict programming language is not quite as straightforward as might be hoped. For example, the *MeetSymm* law given above is not valid for the instance declarations given in this

https://doi.org/10.1017/S0956796800000514 Published online by Cambridge University Press

paper unless we restrict the values taken by its free variables x and y to be proper values. This is just as much of a problem when type classes are not involved, and is simply a reflection of the fact that the properties of familiar mathematical functions are not always shared by their computable counterparts; for example, a law of the form $(x == x) = True$ which might be used to assert that $(==)$ is reflexive will not be valid for either the primitive or type class implementations of computable equality.

Acknowledgements

I would like to thank Phil Wadler, Colin Runciman and an anonymous referee for their comments on an earlier version of this paper which, I hope, have enabled me to improve the presentation in this version. This work was carried out while the author was a member of the Programming Research Group, Oxford University Computing Laboratory, UK with financial support from the Science and Engineering Research Council of Great Britain.

References

Clack, C. and Peyton Jones, S. 1985. Strictness analysis – a practical approach. *Proc. Functional Programming Languages and Computer Architecture*, Volume 201 of *Lecture Notes in Computer Science*, Springer-Verlag.

Damas, L. and Milner, R. 1982. Principal type schemes for functional programs. *Proc. 8th ACM Symposium on Principles of Programming Languages*, ACM Press.

Hudak, P., Peyton Jones, S. and Wadler, P. (editors). 1991. Report on the programming language Haskell, a non-strict purely functional language (Version 1.1). Technical Report YALEU/DCS/RR777, Yale University.

Hudak, P. and Wadler, P. (editors). 1988. Report of the functional programming language Haskell: draft proposed standard. Technical report YALEU/DCS/RR-666, Yale University.

Hunt, S. 1989. Frontiers and open sets in abstract interpretation. *Proc. Functional Programming Languages and Computer Architecture*, ACM Press.

Jones, M. 1989. Frontiers and strictness analysis. University of Oxford, unpublished report.

Jones, M. 1991. Overloading in Gofer. Chapter 14 in *An Introduction to Gofer*, user documentation distributed with Gofer version 2.21.

Martin, C. and Hankin, C. 1987. Finding fixed points in finite lattices. *Proc. Functional Programming Languages and Computer Architecture*, Volume 274 of *Lecture Notes in Computer Science*, Springer-Verlag.

Peyton Jones. S. and Clack, C. 1987. Finding fixpoints in abstract interpretation. In S. Abramsky and C. Hankin (editors), *Abstract Interpretation of Declarative languages*, Ellis Horwood.

Stoy, J. 1977. *Denotational Semantics: The Scott–Strachey approach to programming language theory*. MIT Press.

Vuillemin, J. 1988. Exact real computer arithmetic with continued fractions. *Proc. ACM Symposium on Lisp and Functional Programming*, ACM Press.

Wadler, P. and Blott, S. 1989. How to make ad-hoc polymorphism less ad-hoc. *Proc. 16th ACM Symposium on Principles of Programming Languages*, ACM Press.

Wadler, P. and Miller, Q. 1988. Introduction to Orwell 5.00. Programming Reseach Group, University of Oxford.