[72] Joseph E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, volume 1 of *The MIT Press Series in Computer Science*. The MIT Press, 1977.

[73] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

[74] Philip Wadler. A new array operation. In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction*, pages 328–335. Santa Fé, New Mexico, USA, September/October 1986. Lecture Notes in Computer Science, Volume 279, Springer–Verlag.

[75] Philip Wadler. 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, pages 246–265. Ellis Horwood, 1987.

[76] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78. Nice, France, ACM, June 1990.

[77] Philip Wadler. Is there a use for linear logic? In *Proceedings of the Symposium on Partial Evaluation and and Semantics–Based Program Manipulation*, pages 255–273. New Haven, Connecticut, USA, ACM, June 1991. The proceeding appears as *SIGPLAN Notices*, 26(9), September 1991.

[78] Philip Wadler. The essence of functional programming. In *19th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 1–14. Albuquerque, New Mexico, USA, ACM, January 1992.

[79] Paul P. Wang. A special case of second–order strictness analysis. Technical Memo MIT/LCS/TM–383, Massachusetts Institute of Technology, February 1989.

[80] Stephen Warshall. A theorem on boolean matrices. *Journal of the Association for Computing Machinery*, 9(1):11–12, January 1962.

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

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

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

[62] Carl G. Ponder, Patrick C. McGeer, and Antony P.–C. Ng. Are applicative languages inefficient? *SIGPLAN Notices*, 23(6):135–139, June 1988.

[63] Paul Walton Purdom, Jr. and Cynthia A. Brown. *The Analysis of Algorithms*. Holt, Rinehart and Winston, 1985.

[64] Neil Sarnak. *Persistent Data Structures*. PhD thesis, Department of Computer Science, New York University, 1986.

[65] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[66] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[67] J. T. Schwartz. Optimization of very high level languages — i. value transmission and its corollaries. *Computer Languages*, 1(2):161–194, June 1975.

[68] J. T. Schwartz. Optimization of very high level languages — ii. deducing relationships of inclusion and membership. *Computer Languages*, 1(3):197–218, September 1975.

[69] Dana S. Scott. Lectures on a mathematical theory of computation. In Manfred Broy and Gunther Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 145–292. Technical University Münich and the NATO science Committee, D. Reidel, 1982.

[70] Dana S. Scott. Domains for denonational semantics. In Mogens Nielson and Erick Meineche Schmidt, editors, *Automata, Languages and Programming*, pages 577–613. Aarhus, Denmark, July 1982. Lecture Notes in Computer Science, Volume 140, Springer–Verlag.

[71] M. Smyth and G. D. Plotkin. The category–theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–783, November 1982.

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

[51] Alan Mycroft. The theory and practice of transforming call–by–need into call–by–value. In Bernard Robinet, editor, *International Symposium on Programing*, pages 269–281. Paris, France, April 1980. Lecture Notes in Computer Science, Volume 83, Springer–Verlag.

[52] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, Department of Compute Science, University of Edinburg, 1981.

[53] Alan Mycroft and Neil D. Jones. A relational framework for abstract interpretation. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, pages 156–171. Copenhagen, Demark, October 1985. Lecture Notes in Computer Science, Volume 217, Springer–Verlag.

[54] Eugene W. Myers. Efficient applicative data types. In *11th Annual ACM Symposium on Principles of Programming Languages*, pages 66–75. ACM, January 1984. Salt Lake City, Utah, USA.

[55] Flemming Nielson. Two–level semantics and abstract interpretation. *Theoretical Computer Science*, 69(2):117–242, December 1989.

[56] Hanne Riis Nielson and Flemming Nielson. Automatic binding time analysis for a typed λ–calculus. *Science of Computer Programming*, 10(2):139–176, February 1988.

[57] Hann Riis Nielson and Flemming Nielson. Transformations on higher–order functions. In *Functional Programming Languages and Computer Architecture*, pages 129–143. Imperial College, London, UK, ACM/Addison–Wesley, September 1989.

[58] Martin Odersky. How to make destructive updates less destructive. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 25–36. Orlando, Florida, USA, ACM, January 1991.

[59] Martin Odersky. Observers for linear types. In Bernd Krieg–Brückner, editor, *4th European Symposium on Programming*, pages 390–407. Rennes, France, February 1992. Lecture Notes in Computer Science, Volume 582, Springer–Verlag.

[40] Paul Hudak and Jonathan Young. A collecting interpretation of expressions (without powerdomains). In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 107–118. ACM, January 1988. San Diego, California, USA.

[41] Pual Hudak and Jonathan Young. Collecting interpretations of expressions. *ACM Transactions on Programming Languages and Systems*, 13(2):269–290, April 1991.

[42] John Hughes. An efficient implementation of purely functional arrays. Technical report, Department of Computer Sciences, Chalmers University of Technology, 1985.

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

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

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

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

[47] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *20th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 71–84. Charleston, South Carolina, USA, ACM, January 1993.

[48] Benton L. Leong and Joel I. Seiferas. New real–time simulations of multihead tape units. *Journal of the Association for Computing Machinery*, 28(1):166–180, January 1981.

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

[29] K. Gopinath and John L. Hennessy. Copy elimination in functional languages. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 303–314. ACM, January 1989. Austin, Texas, USA.

[30] David Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer–Verlag, 1981.

[31] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing Series. The MIT Press, 1992.

[32] Juan C. Guzmán and Paul Hudak. Single–threaded polymorphic lambda calculus. In *Proceedings of 5th Annual IEEE Symposium on Logic in Computer Science*, pages 333–343. IEEE, June 1990.

[33] Sören Holmström. How to handle large data structures in functional languages. In *Proceedings of the SERC Chalmers Workshop on Declarative Programming Languages.* University College London, 1983.

[34] Robert Hood and Robert Melville. Real–time queue operations in pure Lisp. *Information Processing Letters*, 13(2):50–53, 1981.

[35] Rob R. Hoogerwoord. A symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513, October 1992.

[36] Paul Hudak. Arrays, non–determinism, side–effects, and parallelism: A functional perspective. In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction*, pages 312–327. Saanta Fé, New Mexico, USA, September/October 1986. Lecture Notes in Computer Science, Volume 279, Springer–Verlag.

[37] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *12th Annual ACM Symposium on Principles of Programming Languages*, pages 300–314. ACM, January 1985. New Orleans, Louisiana, USA.

[38] Paul Hudak, Simon Peyton Jones, and Philip Wadler, editors. Report on the programming language Haskell — a non–strict, purely functional language. Version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[39] Paul Hudak and Jonathan Young. Higher–order strictness analysis in untyped lambda calculus. In *13th Annual ACM Symposium on Principles of Programming Languages*, pages 97–109. ACM, January 1986. St. Petersburg Beach, Florida, USA.

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

[20] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, August 1979.

[21] Paul F. Dietz. Fully persistent arrays. In Frank Dehne, Jörg–Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures*, pages 67–74. Ottawa, Canada, August 1989. Lecture Notes in Computer Science, Volume 382, Springer–Verlag.

[22] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 365–372. New York City, New York, USA, ACM, May 1987.

[23] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[24] James R. Driscoll, Daniel D. K. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. In *Proceedings of the Second Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 89–99, January 1991. San Francisco, California, USA.

[25] Joseph H. Fasel and Robert M. Keller, editors. *Graph Reduction*. Santa Fé, New Mexico, U.S.A., September/October 1986. Lecture Notes in Computer Science, Volume 279, Springer–Verlag.

[26] P. C. Fischer, A. R. Meyer, and A. L. Rosenberg. Real–time simulation of multihead tape units. *Journal of the Association for Computing Machinery*, 19(4):590–607, October 1972.

[27] Hania Gajewska and Robert E. Tarjan. Deques with heap order. *Information Processing Letters*, 22(4):197–200, April 1986.

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

[10] Adrienne Gael Bloss. *Path Analysis and the Optimization of Non–Strict Functional Languages*. PhD thesis, Department of Computer Science, Yale University, May 1989. Also appears as report YALEU/DCS/RR–704.

[11] Adrienne Bloss. Update analysis and the efficient implementation of functional aggregates. In *Functional Programming Languages and Computer Architecture*, pages 26–38. ACM/Addison–Wesley, September 1989. Imperial College, London, UK.

[12] Adam L. Buchsbaum. Private communication, 1993.

[13] Adam L. Buchsbaum and Robert E. Tarjan. Confluently persistent deques via data structural bootstrapping. In *Proceedings of the Fourth Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 155–164. Austin, Texas, USA, January 1993.

[14] Geoffrey L. Burn, Chris L. Hankin, and Samson Abramsky. The theory of strictness analysis for higher order functions. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, pages 42–62. Copenhagen, Demark, October 1985. Lecture Notes in Computer Science, Volume 217, Springer–Verlag.

[15] Geoffrey L. Burn, Chris L. Hankin, and Samson Abramsky. Strictness analysis for higher–order functions. *Science of Computer Programming*, 7(3):249–278, 1986.

[16] Tyng–Ruey Chuang. Fully persistent arrays for efficient incremental updates and voluminous reads. In Bernd Krieg–Brückner, editor, *4th European Symposium on Programming*, pages 110–129. Rennes, France, February 1992. Lecture Notes in Computer Science, Volume 582, Springer–Verlag.

[17] Tyng–Ruey Chuang and Benjamin Goldberg. A syntactic approach to fixed point computation on finite domains. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 109–118. San Francisco, California, USA, June 1992. The proceedings aslo appears as *Lisp Pointers*, Volume V, Number 1, January–March 1992.

[18] Tyng–Ruey Chuang and Benjamin Goldberg. Real–time deques, multihead Turing machines, and purely functional programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 289–298. University of Copenhagen, Demark, June 1993. ACM Press.

Bibliography

[1] Annika Aasa, Sören Holmström, and Christina Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28(3):490–503, 1988.

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

[3] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I–structure: Data structutes for parallel computing. In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction*, pages 336–369. Santa Fé, New Mexico, USA, September/October 1986. Lecture Notes in Computer Science, Volume 279, Springer–Verlag.

[4] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I–structures: Data structure for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[5] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[6] Henry G. Baker, Jr. Shallow binding in Lisp 1.5. *Communications of the ACM*, 21(7):565–569, July 1978.

[7] Henry G. Baker. Shallow binding makes functional arrays fast. *SIGPLAN Notices*, 26(8):145–147, August 1991.

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

[9] Amir M. Ben–Amram and Zvi Galil. On pointers versus addresses. *Journal of the Association for Computing Machinery*, 39(3):617–648, July 1992.

```
    fun length (LIST l)      = List.length l
      | length (PAIR (L, R)) = length' L + length' R
end
end
```

```
        then (h, PAIR (NORM (L, l-1), rhs))
        else if l >= 2
             then (h, PAIR (steps 6 (RevS1 (([], 0, L, 2*l-1), L, []),
                                     RevB  (([], 0, R, r-1),   R, [], r-1))))
             else (h, LIST (rev ((op @) (unpack R))))  end
    | pop LHS (PAIR (L, R)) =
      let  val (e, L) = pop' L  in  (e, PAIR (steps 4 (L, R)))  end
    | pop RHS deque         =
      let  val (e, deque) = pop LHS (swap deque)  in  (e, swap deque)  end

  fun push' z (NORM  (a, b))          = NORM  (push z a, b+1)
    | push' z (RevB  (a, b, c, d))    = RevB  (put  z a, b, c, d)
    | push' z (RevS1 (a, b, c))       = RevS1 (put  z a, b, c)
    | push' z (RevS2 (a, b, c, d, e)) = RevS2 (put  z a, b, c, d, e)
    | push' z (COPY  (a, b, c, d))    = COPY  (put  z a, b, c, d)

  fun push LHS e (LIST l) =
      if length l <= 2
         then LIST (e::l)
         else PAIR (NORM (pack [e,              hd l]      [], 2),
                    NORM (pack [hd (tl (tl l)), hd (tl l)] [], 2))
    | push LHS e (PAIR (NORM (L, l), rhs as NORM (R, r))) =
      let  val L = stack.push e L  in
      if 3*r >= l+1
         then PAIR (NORM (L, l+1), rhs)
         else PAIR (steps 6 (RevB  (([], 0, L, l-r),   L, [], l-r),
                             RevS1 (([], 0, R, 2*r+1), R, [])))  end
    | push LHS e (PAIR (L, R)) = PAIR (steps 4 (push' e L, R))
    | push RHS e deque         = swap (push LHS e (swap deque))

  fun length' (NORM  (_, l))                    = l
    | length' (RevB  ((_, a, _, r), _, _, _))   = a + r
    | length' (RevS1 ((_, a, _, r), _, _))      = a + r
    | length' (RevS2 ((_, a, _, r), _, _, _, _)) = a + r
    | length' (COPY  ((_, a, _, r), _, _, _))   = a + r
```

```
    fun ticks (RevB (currentB, Big, auxB, 0), RevS1 (currentS, _, auxS)) =
              (normalize (COPY  (currentB, auxB, [], 0)),
               RevS2 (currentS, auxS, Big, [], 0))
      | ticks (RevS1 (currentS, _, auxS), RevB (currentB, Big, auxB, 0)) =
              (RevS2 (currentS, auxS, Big, [], 0),
               normalize (COPY (currentB, auxB, [], 0)))
      | ticks (lhs, rhs) = (tick lhs, tick rhs)

    fun steps 0 pair = pair
      | steps n pair = steps (n - 1) (ticks pair)
in
    exception Empty

    datatype    Side  = LHS | RHS
    datatype 'a Deque = LIST of 'a list
                      | PAIR of 'a State * 'a State

    val new = LIST []
    fun empty (LIST []) = true
      | empty _         = false

    fun swap (LIST l)          = LIST (rev l)
      | swap (PAIR (lhs, rhs)) = PAIR (rhs, lhs)

    fun pop' (NORM  (a, b))          = (head a, NORM  (tail a, b-1))
      | pop' (RevB  (a, b, c, d))    = (top  a, RevB  (bot  a, b, c, d))
      | pop' (RevS1 (a, b, c))       = (top  a, RevS1 (bot  a, b, c))
      | pop' (RevS2 (a, b, c, d, e)) = (top  a, RevS2 (bot  a, b, c, d, e))
      | pop' (COPY  (a, b, c, d))    = (top  a, COPY  (bot  a, b, c, d))

    fun pop _   (LIST []) = raise Empty
      | pop LHS (LIST l)  = (hd l, LIST (tl l))
      | pop LHS (PAIR (NORM (L, l), rhs as NORM (R, r))) =
        let  val (h, L) = stack.pop L  in
        if 3*(l-1) >= r
```

```
fun head stack = let  val (element, _) = pop stack  in  element  end
fun tail stack = let  val (_, stack)   = pop stack  in  stack    end

fun put e (extra, added, old, remained) = (e::extra, added+1, old, remained)
fun get   ([],    added, old, remained) =
    (head old, ([], added, tail old, remained-1))
  | get   (e::es, added, old, remained) =
    (e, (es, added-1, old, remained))

fun top current = let  val (element, _) = get current  in  element  end
fun bot current = let  val (_, current) = get current  in  current  end

fun normalize (state as COPY ((extra, added, _, remained), _, new, moved)) =
    if moved = remained
       then NORM (pack extra new, added + moved)
       else state
  | normalize state = state

fun tick state =
    case state of
         NORM _ => state
       | RevB (current, Big, auxB, count) =>
         RevB (current, tail Big, (head Big)::auxB, count-1)
       | RevS1 (current, Small, auxS) =>
         if empty Small
            then state
            else RevS1 (current, tail Small, (head Small)::auxS)
       | RevS2 (current, auxS, Big, newS, count) =>
         if empty Big
            then normalize (COPY  (current, auxS, newS, count))
            else RevS2 (current, auxS, tail Big, (head Big)::newS, count+1)
       | COPY (current as (_, _, _, remained), aux, new, moved) =>
         if moved < remained
            then normalize (COPY (current, tl aux, (hd aux)::new, moved+1))
            else normalize state
```

```
      | pop    ([],    [])    = raise Empty

    fun   pack x y  = (x, y)
    fun unpack pair = pair
end
```

C.3 SML Code for Real–Time Deques

In the following SML code, datatype `Current` is used to hold the "current" stack when elements are being transferred between the two stacks of a deque. It has four fields: the list of newly pushed elements and its count, and the old stack and its remaining count. Datatype `State` has five states whose functions are described in section 4. They are `NORM` (when no element is being transferred), `RevB` (for performing procedure **a**), `RevS1` (for procedure **b**), `RevS2` (for procedure **d**), and `COPY` (for procedure **c** or **e**). Functions `put` and `get` perform push and pop operations, respectively, on datatype `Current`. State transitions are carried out by function `tick`, which is called by `ticks`, which is again called by `steps`.

A `Deque` datatype is either a list of (less than 4) elements, or a pair of states representing the lhs and rhs stacks. Function `swap` exchanges the two sides of a deque. As described in section 4, functions `pop` and `push` will initiate the transfer of elements between the two stacks of a deque if and only if the resulting bigger stack is more than three times the size of the resulting smaller stack.

```
functor Deque (stack : STACK) : DEQUE =
struct
local
    open stack

    type 'a Current = 'a list * int * 'a Stack * int

    datatype 'a State = NORM  of 'a Stack * int
                      | RevB  of 'a Current * 'a Stack * 'a list * int
                      | RevS1 of 'a Current * 'a Stack * 'a list
                      | RevS2 of 'a Current * 'a list * 'a Stack * 'a list * int
                      | COPY  of 'a Current * 'a list * 'a list * int
```

```
    val pack   : 'a list -> 'a list -> 'a Stack
    val unpack : 'a Stack -> 'a list * 'a list
end


signature DEQUE =
sig
    exception Empty

    datatype Side = LHS | RHS

    type 'a Deque

    val new    : 'a Deque
    val empty  : 'a Deque -> bool
    val push   : Side -> 'a -> 'a Deque -> 'a Deque
    val pop    : Side -> 'a Deque -> 'a * 'a Deque
    val length : 'a Deque -> int
end
```

C.2 SML Code for Stacks

```
functor Stack () : STACK =
struct
    exception Empty

    type 'a Stack = 'a list * 'a list

    val new = ([], [])
    fun empty ([], []) = true
      | empty _        = false

    fun push e (x,     y)     =     (e::x, y)
    fun pop    (x::xs, ys)    = (x, (xs,   ys))
      | pop    ([],    y::ys) = (y, (ys,   []))
```

Appendix C

SML Code for Real–Time Deques

The SML signatures for the stack and deque structures are described in section C.1. An implementation of the stack structure is described in section C.2, where each operation, including pack and unpack, costs $O(1)$ time. An implementation of real–time deques is described in section C.3. It includes the definition of states according to the transfer procedures described in section 6.4, the functions to manipulate them, and the actual implementations of the user–level operations described by the SML signature of deques.

C.1 SML Signatures of the Stack and Deque Structures

The objective is to implement each of their operations in constant time and constant space. Note that we will implement a stack by a pair of lists, by using the `pack` and `unpack` functions. The reason for this is stated in section 6.4.

```
signature STACK =
sig
    exception Empty

    type 'a Stack

    val new   : 'a Stack
    val empty : 'a Stack -> bool
    val push  : 'a -> 'a Stack -> 'a Stack
    val pop   : 'a Stack -> 'a * 'a Stack
```

```
                val {threshold = t, length = l, root = R} =
                     reroot {depth = d + 1, current = A'}
                val (l, R) = if l >= t
                             then (0,     duplicate R)
                             else (l + 1, R)

                val r     = #array R
                val cache = case !r of
                              ROOT cache => cache
                            | _          => raise Bugging

                val _ = a := !r
                val _ = r := LINK {index = i,
                                   value = cache sub i,
                                   link  = (Array.update (cache, i, v);
                                            A)}
            in
                {threshold = t, length = l, root = A}
            end
    end

    fun read (A as {array = a, ...}) i =
        case !a of
             ROOT cache =>
                  (case cache sub i of
                        FULL v => v
                      | NULL   => raise Null)
           | LINK _ =>
                  let
                      val {root = R, ...} = reroot {depth = 0, current = A}
                  in
                      read R i
                  end

    fun size {size = s, array = _} = s
end
```

```
fun create n =
    if n > 0
       then {size = n, array = ref (ROOT (array (n, NULL)))}
       else raise Subscript

fun update (A as {size = s, ...}) i v =
    {size = s, array = ref (LINK {index = i, value = FULL v, link = A})}

fun reroot {depth = d, current = (A as {size = s, array = a})} =
let
    fun duplicate {size = s, array = a} =
        case !a of
             ROOT cache =>
                  let
                      val cache' = array (s, NULL)
                      val count  = ref (s - 1)
                      val _      =
                          while !count >= 0 do
                          (Array.update (cache', !count, cache sub !count);
                           count := !count - 1)
                  in
                      {size = s, array = ref (ROOT cache')}
                  end
           | _ => raise Bugging
in
    case !a of
         ROOT cache =>
              let
                  val t = if d > 2 * s
                             then (d - 1) div ((d - 1) div s + 1) + 1
                             else d
              in
                  {threshold = t, length = 0, root = A}
              end
       | LINK {index = i, value = v, link = A'} =>
              let
```

```
    (* See the reference manual of the Standard ML of New Jersey for details. *)

    val create : int -> '1a Array
    val update : '3a Array -> int -> '3a -> '3a Array
    val read   : '2a Array -> int -> '2a
    val size   : '1a Array -> int
end
```

B.2 SML Code for Fully Persistent Arrays

The `array` data type used in the definition of datatype `Node` is built–in (*i.e.*, the `array` in `...ROOT of 'a Cell array ...`). So is the `array` construction in the function definition `create` (*i.e.*, the `array` in `...ROOT (array (n, NULL)) ...`). The array subscription function `sub` is built–in, so is the update operation `Array.update`. They are used in the function definition of `reroot`. Other array operations — `create`, `update`, `reroot`, `read`, and `size` — are defined on the fully persistent data type `Array` defined in the functor. Notice that the definition of the fully persistent `update` operation is purely functional, and that `read` operations often perform `reroot` before the actual reads.

```
functor Array () : ARRAY =
struct
    datatype 'a Cell = NULL
                     | FULL of 'a
    datatype 'a Node = ROOT of 'a Cell array
                     | LINK of {index : int,
                                value : 'a Cell,
                                link  : {size  : int,
                                         array : 'a Node ref}}

    type 'a Array = {size  : int,
                     array : 'a Node ref}

    exception Null
    exception Subscript = Array.Subscript
    exception Bugging
```

Appendix B

SML Code for Fully Persistent Arrays

The ephemeral (*i.e.*, side–effecting) array data structure provided by SML of New Jersey, version 0.66, is used to implement fully persistent arrays, also for SML of New Jersey. The SML signature for persistent arrays is described in section B.1. The implementation, based on fragmented shallow binding, is described in section B.2 as an SML functor `Array`. Notice that the ephemeral array operations of SML/NJ (version 0.66) are used in functor `Array` to implement fully persistent array operations in the same functor. This may cause confusion. Therefore, in section B.2, we will point out in the code those ephemeral array operations which are built–in in SML /NJ (version 0.66).

B.1 SML Signature of Fully Persistent Arrays

```
signature ARRAY =
sig
    exception Null
    exception Subscript

    type 'a Array

    (* Some type variables are weak polymorphic (e.g., '3a, '2a, and '1a)     *)
    (*    because reference is used to implement functional arrays.          *)
```

and

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

This shows the confluence of the first three cases in the above list. It remains to show that that last case is confluent with the first three. But the confluence of $\bigsqcap_{i\in I}\bigsqcup_{j\in J_i} M'_{i,j}$ with $\bigsqcap_{i\in I'}\bigsqcup_{j\in J_i} M_{i,j}$ and $\bigsqcap_{i\in I}\bigsqcup_{j\in J'_i} M_{i,j}$ are covered in the cases for $\sqcap$ and $\sqcup$, respectively. Finally, it can be shown that both $\bigsqcup_{p\in\prod_{i\in I} J_i}\bigsqcap_{i\in I} M_{i,p|i}$ and $\bigsqcap_{i\in I}\bigsqcup_{j\in J_i} M'_{i,j}$ can be reduced to $\bigsqcup_{p\in\prod_{i\in I} J_i}\bigsqcap_{i\in I} M'_{i,p|i}$. This completes the proof. ◇

We now prove the main result.

Proposition A.0.59 $\boldsymbol{\beta}\sqcup\sqcap\boldsymbol{d}$ is Church–Rosser. □

PROOF. Since $\boldsymbol{\beta}\sqcup\sqcap\boldsymbol{d}$ is strongly normalizing (Proposition 3.2.33) and locally confluent (Lemma A.0.58), it follows that it is confluent, which implies Church–Rosser. ◇

contains other redexes. The cases for $\sqcap$ and $\boldsymbol{\beta}$ are not shown here because the former is similar to the case for $\sqcup$, and the latter is straightforward.

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

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

The first two cases are easy because $\bigsqcup_{i\in I-J} M_i \simeq \bigsqcup_{i\in I-J'} M_i$; hence, by Lemma A.0.57, the two terms can be reduced to the same term. It suffices to show that $\bigsqcup_{i\in I-J} M_i$ and $\bigsqcup_{i\in I} M'_i$ reduce to a common term to complete the case for the $\sqcup$–redex. There are two sub–cases, depending whether k is in the set $I - J$ or not, when $M_k \to M'_k$ in term $\bigsqcup_{i\in I} M'_i$. Suppose that $k \in I - J$. Then for all $j \in J$ such that $M_j \preceq M_k$, let $M_j \to^* M''_j$ and $M''_j \preceq M'_k$ (Lemma A.0.56). Otherwise, let $M''_i \equiv M'_i$ for all other $i \in I$. It follows that both $\bigsqcup_{i\in I-J} M_i \to^* \bigsqcup_{i\in I-J} M''_i$ and $\bigsqcup_{i\in I} M'_i \to^* \bigsqcup_{i\in I-J} M''_i$. On the other hand, suppose $k \in J$. Then for $i \in I - J$ such that $M_k \preceq M_i$, let $M_i \to^* M''_i$ and $M'_k \preceq M''_i$ (Lemma A.0.56). Furthermore, for those $j \in J$ and $M_j \preceq M_i$, also let $M_j \to^* M''_j$ and $M''_j \preceq M''_i$. Otherwise, let $M''_i \equiv M'_i$ for all other $i \in I$. It follows that both $\bigsqcup_{i\in I-J} M_i$ and $\bigsqcup_{i\in I} M'_i$ reduce to $\bigsqcup_{i\in I-J} M''_i$.

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

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

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

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

which implies

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

By Lemma 3.2.31, the above predicate proves relationship (A.1).

This completes the proof. ◇

Lemma A.0.57 Let $M, N \in \Lambda$. If $M \simeq N$, then there exists a term $L \in \Lambda$ such that both $M \rightarrow^* L$ and $N \rightarrow^* L$. □

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

By Lemma 3.2.31, it suffices to show that, if $\bigsqcup_{i\in I} M_i \simeq \bigsqcup_{j\in J} N_j$ then they can be reduced to a common term, and if $\bigsqcap_{i\in I} M_i \simeq \bigsqcap_{j\in J} N_j$ then they can also be reduced to a common term, where I, J are index sets and $M_i, N_j \in \Lambda_{\mathcal{Z}}$. We show the case for disjunction. The case for conjunction is similar.

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

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

Lemma A.0.58 $\boldsymbol{\beta \sqcap \sqcup d}$ is locally confluent. □

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

We show local confluence for reductions on overlapping redexes using a case–by–case analysis. We show two cases, where a $\sqcup$–redex contains other redexes, and where a $\boldsymbol{d}$–redex

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

$$(\forall k \in K)(\forall j \in J_{f(k)})(M_{f(k),j} \preceq N_{k,g_{f(k),k}(j)}). \tag{A.3}$$

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

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

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

By predicate (A.3) and Lemma 3.2.31, it follows that for all $i \in I$ and for all $j \in J_i$,

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

because

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

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

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

because

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

for all $i \in I$. Furthermore,

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

because f is total. It follows that

$$\bigsqcap_{i \in I} M_{i,p|i} \quad \preceq \quad \bigsqcap_{k \in K} N_{k,g_{f(k),k}(p|f(k))} \tag{A.4}$$

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

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

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

Function m is well defined because f, g, and $|$ are all well defined. From (A.4) and m, we have for all $p \in \prod_{i \in I} J_i$,

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

Appendix A

Proofs

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

Lemma A.0.56 Let $M, N \in \Lambda$ and $M \preceq N$. If $M \to M'$, then there exists a term N' such that $N \to^* N'$ and $M' \preceq N'$. Conversely, if $N \to N'$, then there exists a term M' such that $M \to^* M'$ and $M' \preceq N'$. □

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

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

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

$$\bigsqcup_{p \in \prod_{i \in I} J_i} \bigsqcap_{i \in I} M_{i,p|i} \quad \preceq \quad \bigsqcup_{q \in \prod_{k \in K} L_k} \bigsqcap_{k \in K} N_{k,q|k} \tag{A.1}$$

if

$$\bigsqcap_{i \in I} \bigsqcup_{j \in J_i} M_{i,j} \quad \preceq \quad \bigsqcap_{k \in K} \bigsqcup_{l \in L_k} N_{k,l}. \tag{A.2}$$

By Lemma 3.2.31, relationship (A.2) implies

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

scenarios. The two themes of the value–representation distinction and amortization complexity analysis in fact interact closely in our development of functional arrays and deques.

With respect to future work, it will be very valuable to have an efficient implementation of fully persistent arrays in which no reference locality is assumed. Our implementation assumes that read sequences are voluminous. Perhaps the results on pointer machines versus random–access machines of Ben–Amram and Galil [9] can be used for future development. On fully persistent deques, it is open whether deque concatenation can also be implemented in constant real–time, in addition to the real–time pop and push operations. It would be interesting to see if a purely functional implementation of deques exists such that each concatenation, pop, and push operation costs only constant worst–case time.

in representation does not affect the integrity of all arrays. Although fragmenting along a long rerooting path takes much time, it pays off in future array operations, which will operate at smaller scale representations of the arrays.

The same distinction between value and representation is also shown in the real–time implementation of functional deques. In our real–time deque implementation, a normal deque is represented as two stacks, with pop/push operations at one end go to one stack, and pop/push operations at the other end go the the other stack. The sizes of the two stacks are kept balanced. If the two stacks grow unbalanced and some elements have to be moved from one stack to the other to regain the balance, the representation of the deque is changed. An "in–transition" deque is then represented in fact by two versions of the same deque. One version is the two stack representation which can become increasingly unbalanced. However, pop and push operations can be performed in real–time on this representation. The other version residing in an in–transition deque is a new and balanced two–stack representation of the same deque, which is incomplete and is being incrementally constructed. Until it is completely rebuilt, this version of the deque cannot be used for pop and push operations. So, a user level pop/push operation on an in–transition deque in fact operates concurrently on both versions. It performs the physical pop/push operation on the unbalanced version, but also contributes a few steps in the incremental rebuilding of the balanced version, which will replace the unbalanced version when it is completely built.

The technique of making the distinction between value and representation can be fruitful when designing other functional aggregate data structures. It remains to be explored. The other common characteristic shared by both the array and deque implementations is the algorithmic technique they use. They both use amortization complexity analysis to show that the proposed implementations offer good performance for any worst–case sequence of operations. (The real–time deque implementation in Section 6.4 has worst–case constant overhead, but it is evolved from the preceding design in Section 6.3, which has amortized performance.) The reason we need an amortization analysis is that although changing the representation of an aggregate from one to another (but keeping the same functionality) takes much time, the cost can be amortized by faster access to the aggregate in the future. An analysis is then used to show the amortization works for all possible

7.2 Efficient Implementations of Functional Aggregates

We have shown that arrays can be made fully persistent by using a fragmented shallow binding scheme. It is shown that the resulting data structure can support multi–threaded accesses to arrays with low overhead, assuming some reference locality. Side–effects are used in the implementation. We also show that deques can be implemented efficiently for functional programs. Every access to either end of any version of a deque costs only constant real–time. The implementation is purely functional. The above two implementations differ from each other in several ways: Side–effects are used in the array case, but the deque case is purely functional. The array implementation offers only amortized performance, the deque implementation is guaranteed to be real–time. What common lessons do we learn from these two case studies?

One common characteristic shared by the two implementations is that they all make clear the distinction between the value of an aggregate and the underlying representation of the aggregate. An aggregate can have many different representations in an implementation, as long as they all have the same functionality. What we must devise, however, is an access mechanism that will always produce the same outcome from the various underlying representations of an aggregate.

In the array case we see that, in the shallow binding scheme, an updated array is represented as a cache (with constant access time) while its previous version is represented by a node recording the previous value of the updated entry and a parent link to the cache. A rerooting operation on the previous version of the array, however, will exchanges the representations of the two arrays. The previous version of the array is now a cache, while the updated version is a child node pointing to the cache. Baker's shallow binding scheme improves the deep binding scheme in that his scheme always reroots the node to be read/updated, which makes easy the task of read/update at the designated node because it is now represented as a cache. Our fragmented shallow binding scheme improves Baker's original scheme in that ours scheme does not reroot when performing updates. If there is no read following an update, it makes no sense to spend the time in making the updated array the root node. Moreover, if a read operation incurs a long rerooting, then we further divide the version tree, along the rerooting path, into a forest in which each smaller tree has its own root. Here, we change the representations of the arrays again, not only by rerooting, but also by breaking a long rerooting path. We also make sure that the change

Chapter 7

Conclusion and Future Work

We recapitulate in this chapter the results developed in this dissertation. We also contemplate several possible directions of future work.

7.1 A Syntactic Method for Least Fixed Point Finding

We have developed a syntactic approach to finding the least fixed points of higher–order functions over finite domains. The proposed syntactic method is shown to be sound and complete with respect to the semantics. It is shown that the method is quite effective, especially in contrast to the usual frontier method which may be very inefficient even for simple problems.

The syntactic method works on the two–element domain $\mathit{2}$, and the monotonic function spaces generated by $\mathit{2}$. For finite domains not of this nature, they must be encoded into some monotonic function spaces generated by $\mathit{2}$. For instance, in Section 3.5 we embed domains $\mathit{3}, \mathit{4}, \ldots$ respectively into domains $\mathit{2} \to \mathit{2}, (\mathit{2} \to \mathit{2}) \to \mathit{2}, \ldots$. Least fixed point computation over these domains can then be carried out in the proposed framework. It will be interesting to develop syntactic method that can be applied to all finite domains of reasonable shapes, without the explicit encoding. However, this will change much of the language Λ we currently use. The completeness result enjoyed by the sub–language Λ^0 of Λ would have to be re-established in the new language (see discussion in Section 3.4.3).

design of fully persistent real–time deques is not purely functional because side–effects are used. His scheme is also more complicated than ours. Hoogerwoord's design is purely functional, but not real–time, and is not suitable for multi–threaded applications. While not aware of the above two works, we start with the result of Hood and Melville [34] on purely functional real–time queues, and extend it to purely functional real–time deques. The underlying idea, that a deque is represented as two balanced stacks, is the same.

An open problem is whether deque concatenation can also be implemented in real–time in a purely functional way, in addition to the real–time pop and push operations. Buchsbaum and Tarjan [13] recently show that deques can be made confluently persistent (*i.e.*, concatenateable) with constant cost for each concatenation and push operation, but at the cost of $O(\log^* n)$ for each pop operation, where n is the deque size.[1] All costs are amortized. This improves a previous result of Driscoll, Sleator, and Tarjan [24]. Using our purely functional implementation of real–time deques, their implementation can be made purely functional, with amortized complexity improved to worst–case complexity for each operation [12]. It would be interesting to see if a purely functional implementation of deques exists such that each concatenation, pop, and push operation costs only constant worst–case time.

[1] The function $\log^*$ is defined for every integer $n \geq 0$ such that

$$\begin{cases} \log^* n = 0 & \text{if } n = 0, \\ \log^* n = 1 + \log^* \lceil \log n \rceil & \text{otherwise.} \end{cases}$$

Intuitively, $\log^* n$ is the number of repeated logarithm operations to take n to 0. For example, $\log^* 1 = 1, \log^* 2 = 2$, and $\log^* 2^{2^{2^2}} = 5$.

Are the data structures described in this chapter still attractive if implemented in a non–strict language or implemented within a system that does not provide real–time garbage collection? Even though the resulting program will not have real–time properties, it is still important to have efficient implementations of aggregate data structures, such as deques, which are suitable for multi–threaded applications. Using a non–strict functional language, the performance of the deques described in section 6.4 will degrade to $O(1)$ amortized time, instead of $O(1)$ real–time, per deque operation in a multi–threaded program. This is better than the naive implementation of deques described in section 6.3, which achieves $O(1)$ amortized complexity only in single–threaded programs.

6.7 Related Work and Future Work

A real–time simulation of a multihead Turing machine in a purely functional language does not come as a big surprise. Fischer, Meyer, and Rosenberg [26] and Leong and Seiferas [48] has shown that a multihead Turing machine can be simulated in real–time by a multitape Turing machine with only one head per tape. A single tape, single head Turing machine can be simulated in real–time in a purely functional language, by two stacks and a finite control. Thus, it is clear that a multihead Turing machine can be simulated in real–time in a purely functional language. Nevertheless, the simulations of [26] and [48] are complicated, and their resulting simulations of a multihead Turing machine in a purely functional language will probably not be as simple as ours. Furthermore, our objective is to demonstrate that purely functional languages can be used to implement non–trivial aggregate data structures, such as deques and multihead Turing machines, efficiently and straightforwardly, which is different from the objectives of [26] and [48].

In our purely functional implementation of real–time deques, two techniques are used: a deque is represented as two balanced stacks, and incremental method is used to balance the two stacks. Incremental methods have been used in many contexts to improve algorithms of amortized performance to worst–case performance. Baker's algorithm for real–time garbage collection is a good example [5]. Baker took a copying garbage collection algorithm which had $O(1)$ amortized cost per storage allocation (this cost includes the high cost of the stop–and–copy collection) and modified it to be $O(1)$ worst–case cost.

Both Sarnak [64] and Hoogerwoord [35] use two stacks to represent a deque. Sarnak's

is clear that every move of a multihead Turing machine (printing a symbol under a head, moving a head left or right one cell), can be accomplished in real–time by performing the operations at the corresponding deques.

Given the next move function δ from $Q \times \Sigma^k$ to $Q \times K \times (\Sigma \cup \{L, R\})$ and the representation of the tape described above, we can implement in a purely functional way a next move function Δ from $Q \times \mathit{Tape}$ to $Q \times \mathit{Tape}$ such that each function application of Δ costs $O(k)$ time and space. *Tape* is the data type that describes exactly the current tape configuration of the machine, including the the contents of the tape, the positions of the heads, and the permutation function f. The permutation function f is needed because if two heads cross each other on the tape, their ordering on the tape will change and the permutation function f must be changed accordingly. This results in a real–time simulation in a purely functional language of a multihead Turing machine, where each move of the machine is accomplished in $O(k)$ time and space. Note that the complexity of the simulation, $O(k)$ per move, depends only on k, the number of heads, not on n, the number of symbols on the tape. Also note that every configuration of the execution history of a multihead Turing machine is equally accessible in our real–time simulation. This means that our real–time simulation can be multi–threaded, which is very useful if we want to simulate a nondeterministic multihead Turing machine where there may be several eligible next moves at any given machine configuration.

6.6 Remarks

Throughout our discussion in this chapter, we implicitly assume that the purely functional language used to implement the deques is a strict language rather than non–strict language in which the evaluation of function arguments are delayed until needed. A strict language is required because the incremental nature of the algorithm requires that elements be transferred between stacks well in advance of the use of those elements. A non–strict language would delay most of the transfer steps when a deque operation occurs. However, a subsequent deque operation (such as a popping an element and performing a strict operation on the element) would cause many of the delayed transfer steps to execute. Naturally, this would destroy the real–time nature of the algorithm.

left or right one cell. Let Q be the finite set of states in the machine, Σ be the finite set of tape symbols, and K be the finite set of heads $\{h_1, h_2, \ldots, h_k\}$. The next move function δ is a mapping from $Q \times \Sigma^k$ to $Q \times K \times (\Sigma \cup \{L, R\})$, where L and R will move the head left and right one cell respectively. Notice that the machine can move or write using only one of its heads at a time. The machine starts with an initial state and an initial sequence of symbols on the tape. It proceeds according to the next move function, and halts if it reaches a final state. The resulting sequence of symbols on the tape is the output of the machine.

We are interested in multihead Turing machines because they provide a model for limited random access memory. A k–head Turing machine can be viewed as a linear memory machine with multiple program counters. These program counters differ from the program counter in a usual random access machine in that they cannot be arbitrarily reset to point to any memory locations. Only increment, decrement, read, and write instructions are available to manipulate these program counters. To compensate the loss of the arbitrary jump instruction, k program counters are made available such that the machine can memorize k different locations in the memory. It would be interesting to see how well the purely functional model (which assumes non–linear memory and admits no side–effects) can simulate this limited model of random access memory (which assumes linear memory and admits side–effects).

We now describe how to simulate in real–time a multihead Turing machine in a purely functional language. The contents of the tape and the positions of the heads can be implemented by a sequence of deques with each deque representing a segment of the tape delimited by its two surrounding heads. Let the entire contents of the tape be described by a sequence of symbols $S = (s_0, s_2, \ldots, s_n)$ with the k heads located at symbol $s_{l_i}, 1 \leq i \leq k$. Without loss of generality, we assume that $0 \leq l_i \leq l_{i+1} \leq n$ for all $1 \leq i \leq k-1$. Recall that $S^\diamond$ denotes the deque containing exactly the elements of sequence S, with elements s_0 and s_n at the two ends of the deque. The tape, along with the head positions, can be represented by a sequence of deques $(D_0, D_1, \ldots, D_k)$, where $D_0 = (s_0, s_1, \ldots s_{l_1-1})^\diamond$, $D_i = (s_{l_i}, s_{l_i+1}, \ldots s_{l_{i+1}-1})^\diamond$ for $1 \leq i \leq k-1$, and $D_k = (s_{l_k}, s_{l_k+1}, \ldots s_n)^\diamond$. Because several heads may be positioned over the same cell on the tape, some of the deques may be empty. Also, head h_i, the ith head in K, need not be positioned over symbol s_{l_i}. Rather, there is a permutation function f over $\{1, 2, \ldots k\}$ such that head h_i will position at symbol $s_{l_{f(i)}}$. It

resulting smaller stack will have m elements, and the resulting bigger stack will have size $2m + 1$. Since $3 \cdot m \geq 2m + 1$ for all $m \geq 1$, the invariant (6.1) has been maintained.

6.5 Applications

Being able to add real–time deques to a purely functional language can lead to some interesting results. We show in this section two applications. The first is some implications in purely functional list processing. The other is a real–time simulation of a multihead Turing machine in a purely functional language.

6.5.1 Purely Functional List Processing

Here is a quiz: How much time will it take to reverse a list of n elements in a (sequential) purely functional language? A conventional method will take $O(n)$ time. But a list can be implemented, in a purely functional way, as a real–time deque without losing its functionality or its constant time performance. Furthermore, we can reverse a deque in constant time because of its symmetric nature. It turns out that a list of an arbitrary number of elements can be reversed in constant worst–case time if it is implemented as a real–time deque and an additional orientation tag is attached to the deque.

This gives us a new way to concatenate two lists. The usual method will take $O(|X|)$ time to concatenate a list Y to the rear of a list X. However, if lists X and Y are implemented as real–time deques with the lhs of the deque as the front of the list and the rhs as the rear, then their concatenation can be implemented in the following way: If $|Y| < |X|$, then pop elements from the lhs of Y and push them to the rhs of X; otherwise, pop elements from the rhs of X and push them to the lhs of Y. This takes $O(min(|X|, |Y|))$ worst–case time, which is better than the usual $O(|X|)$ time.

6.5.2 Real–Time Simulation of a Multihead Turing Machine in a Purely Functional Language

A multihead Turing machine is a Turing machine with a single two–way linear tape and multiple read/write heads upon the tape. In one move the multihead Turing machine, depending on the symbols scanned by the heads and the state of the finite control, changes its state and either prints a symbol on the cell scanned by one head or moves one head

larger than the rhs stack, then initiate the transfer process.

- dq is a pair of stacks, and is transferring elements between them $\Longrightarrow$ Push e into the lhs *extra* list, then perform 4 incremental steps of the transfer process. If the transfer process is now complete, then pair–up the *extra* list with the *new* list to form a new stack, both for the lhs and the rhs.

- *pop lhs dq*: Depending on the representation of dq, perform one of the following actions,

  - dq is a list of (less than 4) elements $\Longrightarrow$ Return the "car" and "cdr" of the list.
  - dq is a pair of stacks, but is not currently transferring elements between the two stacks $\Longrightarrow$ Pop the lhs stack. If it results in an empty lhs stack, then the deque is now represented as a list of the elements in the rhs stack. If the rhs stack is three times larger than the resulting lhs stack, then initiate the transfer process.
  - dq is a pair of stacks, and is transferring elements between them $\Longrightarrow$ Pop the lhs *extra* list if it is not empty, otherwise pop the old lhs stack. Perform 4 incremental steps of the transfer process. If the transfer process is now complete, then pair–up the *extra* list with the *new* list to form a new stack, both for the lhs and the rhs.

- *new*: Return an empty list.

- *empty dq*: Return true if dq is an empty list, otherwise, return false.

The actual code, written in the purely functional subset of SML (of New Jersey), along with some annotations, is included in Appendix C.

Before we conclude this section, it remains to be shown that the invariant (6.1) is maintained after the transfer process. Suppose for the moment that the m deque operations that carry out the entire transfer process do not interfere with the rebuilding of the two new stacks. In such a case, the resulting deque will have one stack of size $2m + 1$ and the other stack of size $2m + k - 1$ in its representation, where k may be $1, 2$, or 3. The sizes of the two stacks will become most unbalanced if the m deque operations that carry out the transfer process all happen to be pop operations aimed at the smaller of the above two stacks. This occurs when $k = 1$ and the stack of size $2m + k - 1$ are popped m times. The

can be carried out concurrently, which takes no more than $2m + 3$ steps. A step costs two units of time and space by moving one element from B to $auxB$ and moving one element from S to $auxS$. Similarly, procedure **c** can be carried out concurrently with **d** and **e**, taking a total of at most $2m + 3$ steps. In total, $4m + 6$ steps are sufficient to complete the transfer process. Since the transfer process will be distributed evenly over the next m deque operations, we allocate 6 steps to the deque operation that violates the invariant, and 4 steps to each of the m deque operations that follow.

During the course of the transfer, pop operations can be performed by the user on the original stacks S and B. Therefore, we must take care not to copy those discarded elements back onto the new stack $newS$ and $newB$ from the auxiliary stacks $auxS$ and $auxB$. A counter is used to keep track of the number of remaining elements in the original stack. Push operations during the course of transfer also cause problems. The elements pushed to stack S, for example, must be kept in a separate place $extraS$, to be annexed with $newS$ at the end of the transfer process to become the real new stack S. We cannot simply append $newS$ to $extraS$ for this will ruin the real–time performance at the very last step. Instead, we use the pair $\langle ExtraS, newS \rangle$ as the representation of the stack S when the transfer process is completed. This complicates pop and push operations on stacks. Pop and push operations on stack S have to examine $ExtraS$ first to see if it is empty. If it is not empty, then the pop or push operation is performed on it, otherwise the operation is performed on $newS$. But, in exchange, we maintain real–time performance. This implies that, in the normal representation of a deque as an lhs stack and an rhs stack, each of the two stacks is in fact a pair of lists in our implementation.

To summarize, we now describe below the high–level implementation of the deque operations. Only the lhs operations are described; the rhs cases are symmetrical to the lhs cases.

- *push lhs e dq*: Depending on the representation of *dq*, perform one of the following actions,
  - *dq* is a list of (less than 4) elements $\Longrightarrow$ "*Cons e dq*". If the resulting list has 4 elements, then split it in half into an lhs stack and an rhs stack.
  - *dq* is a pair of stacks, but is not currently transferring elements between the two stacks $\Longrightarrow$ Push *e* into the lhs stack. If the resulting lhs stack is three times

is replaced by a list of its elements. This list will contain at most three elements, and all push and pop operations afterward will be handled in an ad hoc way by looking at the head and tail of the list. If the size of the list grows to four by subsequent push operations, we then break the list into equally sized lhs and rhs stacks and usual representation is subsequently used. It is clear that each of the above ad hoc treatments consumes only a constant amount of time and space.

The other case in which the invariant can be violated is by a pop operation on the smaller stack, or by a push operation on the bigger stack, such that the size of the resulting bigger stack is larger than three times the size of the resulting smaller stack. Let S be the smaller stack, and B be the bigger stack, after the violating pop or push operation. If the invariant has been maintained all along and $|S| = m$, then $|B| = 3m + k$, where $m \geq 1$ and k is either $1, 2$, or 3. Without loss of generality, we assume that the resulting deque has the configuration $\langle S, B \rangle$, with $S = (p_1, p_2, \ldots, p_m)^{\triangleleft}$ and $B = (q_1, q_2, \ldots, q_{3m+k})^{\triangleright}$, where k may be $1, 2$, or 3. We then transfer the bottommost $m + 1$ elements of stack B to the bottom of stack S such that, after the transfer, we have two new stacks, $newS = (p_1, p_2, \ldots, p_m, q_1, q_2, \ldots, q_{m+1})^{\triangleleft}$ and $newB = (q_{m+2}, q_{m+3}, \ldots, q_{3m+k})^{\triangleright}$, as the representation of the deque. It is clear that that transfer needs at most $O(m)$ time and space. We then distribute the transfer process evenly over the next m deque operations. Because each of the following m deque operations need only $O(m)/m = O(1)$ extra cost to rebuild the new stacks, each of of deque operations takes only constant time and consumes only constant space.

We now describe the details. The construction of the new stacks $newS$ and $newB$ can be accomplished by the following procedures:

a. Pop and reverse the topmost $2m + k - 1$ elements of stack B into an auxiliary stack $auxB$ such that $B = (q_1, q_2, \ldots, q_{m+1})^{\triangleright}$ and $auxB = (q_{m+2}, q_{m+3}, \ldots, q_{3m+k})^{\triangleleft}$.

b. Reverse stack S into an auxiliary stack $auxS$ such that $auxS = (p_1, p_2, \ldots, p_m)^{\triangleright}$.

c. Reverse stack $auxB$ into a new stack $newB$ such that $newB = (q_{m+2}, q_{m+3}, \ldots, q_{3m+k})^{\triangleright}$.

d. Reverse stack B into a new stack $newS$ such that $newS = (q_1, q_2, \ldots, q_{m+1})^{\triangleleft}$.

e. Reverse stack $auxS$ onto stack $newS$ such that $newS = (p_1, p_2, \ldots, p_m, q_1, q_2, \ldots, q_{m+1})^{\triangleleft}$.

At the end of the transfer, stacks $newS$ and $newB$ will replace the roles of stack S and B, respectively, in the representation of the deque. Note that procedures **a** and **b** above

but the bank account balance is zero. In such a case, the lhs pop operation will decrease the size difference by one. However, it will not withdraw and discard one credit from the bank account this time, as would be required in the usual situations. That is, the bank account balance again reflects exactly the difference between the stack sizes after the lhs pop operation. A symmetrical analysis also applies to the case of rhs pop operations.

We conclude that $O(1)$ amortized time suffices for each deque operation in a single–threaded sequenced of operations. However, this implementation suffers the same drawbacks of the naive amortized implementation for FIFO queues in section 6.2. It is not real–time and does not suit multi–threaded applications. We will address these problems in the next section.

6.4 Functional Deques with Good Real–Time Performance

For an implementation of deques to be real–time and functional, the implementation must insure that each of the push or pop operations on either end of a deque is completed in $O(1)$ time in the worst case, and that the push or pop operation returns a new version of the deque without destroying the old version. This section describes such an implementation written in a purely functional language.

As in section 6.3, an lhs stack and an rhs stack together are used to represent a deque. The lhs pop or push operations are performed on the lhs stack, and the rhs pop or push operations are performed on the rhs stack. Problems arise when a pop operation encounters an empty stack. Our strategy is to make sure that such problems never arise. To able to do so, we maintain the following invariant between the two stacks:

$$|B| \geq |S| \geq 1, \text{and } 3|S| \geq |B| \tag{6.1}$$

where B is the bigger stack of two stacks and S is the smaller stack. That is, each of the two stacks has at least one element, and the size of the bigger stack is never larger than three times the size of the smaller stack.

The above invariant can be violated in two ways. The first is if one of the stacks becomes empty due to a pop operation. However, if the invariant has been maintained all along, we know that the other stack has at most three elements. Thus the deque as a whole has at most three elements. At that point, the usual representation of the deque

and vice versa.

We solve this problem by transferring the bottom half of the rhs stack to the empty lhs stack, with the bottommost elements of the rhs stack becoming the topmost elements in the lhs stack after the transfer. That is, for an lhs pop operation on queue $\langle ()^{\triangleleft}, R^{\triangleright} \rangle$, we first transform the configuration of the queue from $\langle ()^{\triangleleft}, R^{\triangleright} \rangle$ to $\langle {R_1}^{\triangleleft}, {R_2}^{\triangleright} \rangle$, where $R = R_1 R_2$ and $|R_1| = \lceil |R|/2 \rceil$. Then we perform an lhs pop operation on $\langle {R_1}^{\triangleleft}, {R_2}^{\triangleright} \rangle$. A symmetrical treatment also applies to an rhs pop operation when the rhs stack is empty but the lhs stack is not.

Does the above strategy provide a good implementation? If a sequence of deque operations is single–threaded and starts with an empty deque, then, by a simple credit–debit argument [73], it can be shown that each operation takes $O(1)$ amortized time and space. For the argument, we associate with a deque an imaginary "bank account" of units of time, whose balance reflects exactly the difference between the sizes of the two stacks in the the deque. That is, the balance will be $| \, |L^{\triangleleft}| - |R^{\triangleright}| \, |$ for a configuration of $\langle L^{\triangleleft}, R^{\triangleright} \rangle$. We also allocate two units of time for each deque operation, which will be the amortized cost of the operation.

A pop or push operation will change the difference between the sizes of the two stacks by one if it does not initiate the transfer process. For such an operation, one unit of time is used to pop/push an element from/to its corresponding stack, and the other unit of amortized time is deposited into the bank account if the pop or push operation increases the size difference. If the size difference is decreased by the push or pop operation, then one unit of time is withdrawn from the account and is discarded along with the other unit of time allocated for the deque operation. By doing so, we make sure that the bank account is consistent with the difference between the sizes of the two stacks.

On the other hand, an lhs pop operation on the deque $\langle ()^{\triangleleft}, R^{\triangleright} \rangle$ will have to transfer the bottom half of the rhs stack $R^{\triangleright}$ to the empty lhs stack, before the pop operation can be performed. The transfer will require $|R^{\triangleright}|$ units of time, which are withdrawn from the bank account. The bank account will have contained exactly $|R^{\triangleright}|$ units before the transfer. After the transfer, the difference of the stack sizes is at most one (with the size of the lhs stack larger than or equal to the size of the rhs stack) and the bank account has a balance of zero. The lhs pop operation then proceeds as if the transfer had never happened. Inconsistency occurs after the transfer process if the stack size difference is one

initiates the transfer, and allocate 3 steps to each of the n queue operations that follows. Since each queue operation performs at most 4 extra steps to aid the construction of the new output stack, each queue operation takes $O(1)$ time and $O(1)$ space.

In the actual implementation, a normal queue is represented by two stacks, one for pop operations and one for push operations. However, an in–transition queue is represented by the old output stack (for pop operations), a new input stack (for receiving elements from push operations), and additional intermediate stacks for incremental construction of the new output stack. Once the new output stack is completely rebuilt, it is paired with the new input stack to represent a normal queue. The old output stack and those intermediate stacks are discarded. A normal queue enters the in–transition mode whenever its output stack becomes smaller than its input stack.

There are some remaining details in Hood and Melville's algorithm, though. In the construction of a new output stack, we need not copy all the elements in the original output stack $O^{\triangleleft}$ because some of elements may have been popped during the course of the transfer. A counter is used to insure that we will not over–copy. Also, it can be shown that the queue is never empty during its transfer of elements, and after the transfer of elements, the invariant $|O^{\triangleleft}| \geq |I^{\triangleright}|$ still holds for the new input stack $I^{\triangleright}$ and the new output stack $O^{\triangleleft}$.

6.3 Functional Deques with Good Amortized Performance

A deque is a linear buffer where push and pop operations can be performed on either end of the buffer. We can modify the naive implementation of FIFO queues described in the above section to get a purely functional implementation of deque with good amortized performance. Similar schemes are also described by Sarnak [64] and Hoogerwoord [35].

A pair of stacks, $\langle L^{\triangleleft}, R^{\triangleright} \rangle$, is used to represent a deque, similar to that for a queue. The difference is that pop or push operations can be performed on stack $L^{\triangleleft}$ or stack $R^{\triangleright}$. We call $L^{\triangleleft}$ the the left–hand–side (lhs) stack and $R^{\triangleright}$ the right–hand–side (rhs) stack, instead of using the terms output stack and input stack previously used. Push and pop operations on the left side of the deque are performed on the lhs stack, and operations on the right side of the deque are performed on the rhs stack. The problem is that an lhs pop operation may be executed while the lhs stack is empty and the rhs stack contains some elements,

transferred at most once from the input stack to the output stack after it is pushed onto the queue. Since there are at most s push operations in a sequence of s queue operations, the total cost of transfer is bounded by s. This results in $O(s)/s = O(1)$ amortized cost per queue operation for the transfer, in addition to the $O(1)$ actual cost per queue operation for implementing the push and pop operations on the input and out stacks respectively.

However, the above implementation has two drawbacks. It is not real–time, and, more severely, does not suit multi–threaded applications, where pop or push operations may be performed not only on the newest version of a queue, but on previous versions as well. For example, let queue $P = \langle ()^{\triangleleft}, (p_1, p_2, \ldots, p_m)^{\triangleright} \rangle$ be a queue formed by a sequence of m push operations. A single pop operation on P will take $O(m)$ time to get element p_1 and form a new queue $Q = \langle (p_2, p_3, \ldots, p_m)^{\triangleleft}, ()^{\triangleright} \rangle$. Suppose that the next operation in the sequence is again a pop operation on P rather than on the newer version, Q. Then $O(m)$ time has to be spent again to get element p_1 and to form a queue identical to Q. Note that in general we have no way to tell whether or not two pop operations will be performed on the same queue.

Hood and Melville improve the above naive implementation by a simple idea, that the transfer of elements from the input stack to the output stack need not be carried out all at once when the output stack becomes empty. The transfer of elements can be carried out *incrementally* over a sequence of queue operations whenever a substantial number of elements have been accumulated in the input stack, even when the output stack is not empty.

Suppose that a queue has the configuration $\langle O^{\triangleleft}, I^{\triangleright} \rangle$ and is to be transformed into the configuration $\langle (OI)^{\triangleleft}, ()^{\triangleright} \rangle$. Suppose also that the transfer of elements from the input stack $I^{\triangleright}$ to the output stack $O^{\triangleleft}$ is initiated only when the invariant $|O^{\triangleleft}| \geq |I^{\triangleright}|$ is violated. At the beginning of the transfer, we can assume $|O^{\triangleleft}| = n$ and $|I^{\triangleright}| = n + 1$ for some $n \geq 0$. The new output stack $(OI)^{\triangleleft}$ can be constructed in three stages: reverse stack $I^{\triangleright}$ creating a stack $I^{\triangleleft}$, reverse stack $O^{\triangleleft}$ creating a stack $O^{\triangleright}$, and then pop all the elements from stack $O^{\triangleright}$ and push them onto stack $I^{\triangleleft}$ one by one. The whole process will take $3n + 4$ steps of time, where each step either moves the topmost element from one stack to another, or is a test (at the bottom of the recursion) to see if a stack is empty. Because the original output stack $O^{\triangleleft}$ will become empty after n pop operations, we must accomplish the $3n + 4$ steps in n regular queue operations. We then allocate 4 steps to the pop or push operation that

Hoogerwoord. Our purely functional implementation of deques with real–time performance is presented in section 6.4. Section 6.5 describes its implications in purely functional list processing. It also shows how to simulate in real–time, in a purely functional language, a multihead Turing machine by using multiple real–time deques. Section 6.6 contains some remarks on real–time processing in a purely functional setting. Section 6.7 discusses related work and future work.

In this chapter, we will use the notation $P = (p_1, p_2, \ldots, p_m)$ to describe a sequence P of m elements $p_1, p_2, \ldots, p_m$. The concatenation of sequences P and Q is denoted by PQ. The notation $P^{\triangleleft}$ describes a stack consisting of the sequence P, with p_1 at the top of the stack and p_m at the bottom. Similarly, $P^{\triangleright}$ is also a stack consisting of the sequence P, but with p_m at the top of the stack and p_1 at the bottom. A deque consisting of the sequence P is denoted by $P^{\diamond}$. Notice that $(p_1, p_2, \ldots, p_m)^{\triangleleft} = (p_m, p_{m-1}, \ldots, p_1)^{\triangleright}$. The size of a sequence P is denoted by $|P|$. Similarly, $|P^{\triangleleft}|, |P^{\triangleright}|$, and $|P^{\diamond}|$ are respectively the sizes of stacks $P^{\triangleleft}, P^{\triangleright}$, and deque $P^{\diamond}$.

6.2 Functional Queues with Good Real–Time Performance

Hood and Melville [34] describe a pure LISP implementation of FIFO queues with good amortized performance. They then modify it to get an implementation, also in pure LISP, with good real–time (*i.e.* worst–case) performance. Their idea is to represent a queue by two disjoint stacks, where the input stack $I^{\triangleright}$ is used to receive the sequence of elements being pushed and the output stack $O^{\triangleleft}$ is used to store the sequence of elements to be popped. We will denote the configuration of a queue as $\langle O^{\triangleleft}, I^{\triangleright} \rangle$. The sequence OI represents the entire sequence of elements of the queue. When the output stack becomes empty and a pop operation is executed, the sequence of elements in the input stack are all transferred to the output stack by reversing the sequence in the input stack to form a new output stack. The input stack is then replaced by an empty stack. That is, the configuration of the queue is transformed from $\langle ()^{\triangleleft}, I^{\triangleright} \rangle$ into $\langle I^{\triangleleft}, ()^{\triangleright} \rangle$. The transfer takes time linear to size of the input stack and can be implemented in a purely functional way.

Starting from an empty queue (with both the input and output stacks empty), a single–threaded sequence of s queue operations will transfer at most s elements from the input stack to the output stack. This is based on the observation that an element will be

example, suppose that the queue is implemented in a side–effecting language as a single–linked list with access pointers to both ends and the standard technique of [23] is used to make the single–linked list fully persistent. Each of the pop or push operations to any version of a queue will need $O(1)$ *amortized* time and space.

We extend in this chapter the result of Hood and Melville such that using a purely functional language, each of the push or pop operations on either end of any version of a deque costs only $O(1)$ time in the worst case. Efficient implementation of deques in a purely functional language was considered both by Gajewska and Tarjan [27] and by Ponder, McGeer, and Ng [62], but without giving a solution. Our results show that it can be done.

The techniques we use are not new. We will use two stacks to represent a deque, with each stack top corresponding to one of the two open ends of the deque. The two stacks are balanced at all time in that the bigger stack is never more than three times the size of the smaller stack. A pop operation on a non–empty deque is performed directly on one of the stacks. The challenge is to make the design as simple as possible — such that a purely functional implementation becomes straightforward — and to get its complexity analysis right. The two–stacks representation is used by Gries [30, pages 250–251] to design purely functional queues with constant amortized cost per operation. Hoogerwoord [35] shows that the representation can be extended to purely functional deques with constant amortized cost per operation. However, these two designs are not suitable for multi–threaded applications because the constant cost for each operation is amortized along a single thread of operations. Hood and Melville [34] show that, by incrementally copying elements from one stack to another, real–time queues can be implemented in a purely functional way. A design for fully persistent deques with constant worst–case time per operation is described in Sarnak's Ph.D. thesis [64]. However, his design is not purely functional because side–effects are used in the implementation. His method, and the associated complexity analysis, is also more complicated than ours. Nevertheless, the basic idea is similar in his and our design: a deque is represented as two stacks and they are kept balanced by incrementally moving elements between them.

In section 6.2, we outline Hood and Melville's purely function method for real–time FIFO queues. We then describe in section 6.3 a purely functional implementation of deques with good amortized performance. This design is also described by both Sarnak and

language without losing their efficiency. For an advocate of functional programming, the challenge is to demonstrate that certain aggregates can be implemented efficiently in purely functional languages. We describe in this chapter a purely functional implementation of double–end queues (deques) such that each push or pop operation on either end of any version of a deque is accomplished in $O(1)$ time in the worst case. We then show that the implementation has some interesting implications, and can be used to give a real–time simulation of a multihead Turing machine in a purely functional language.

We will be discussing and comparing several fully persistent implementations of queues and deques in this chapter. When we are doing that, it is helpful that we ask the following questions regarding the implementations:

- Is the implementation a *purely functional* one? Or, must it use *side–effects* and be implemented in a side–effecting language?
- Is the stated cost of each aggregate operation a *worst–case* cost or an *amortized* one?
- Is the implementation good for *multi–threaded* applications? Or is it only suitable for *single–threaded* applications?

An implementation is said to be *real–time* if each operation costs only constant overhead in the worst case. Notice that a purely functional implementation of an aggregate data structure automatically makes it fully persistent because no side–effect is used in the implementation and previous versions of an aggregate are always accessible. The problem is that the implementation may be inefficient, even under the assumption that it will only be used for single–threaded applications. Also, an implementation (either in purely functional or side–effecting languages) with good amortized performance does not necessarily mean it is good for multi–threaded applications, because the amortized cost may be measured over a single–threaded sequence of operations.

Hood and Melville [34] show that a FIFO queue can be implemented in pure Lisp (where no side–effect is allowed) such that, in the worst case, push or pop operations on the queue cost only $O(1)$ time and consume $O(1)$ space. This is an interesting result for several reasons. First of all, the $O(1)$ complexity applies not only to the newest version of a queue, but to previous versions as well. This makes their implementation suitable for multi–threaded applications. Secondly, it had not previously been shown, even for languages allowing side-effects, how a fully persistent real–time queue could be implemented. For

Chapter 6

Purely Functional, Real–Time Deques

This chapter answers the following question: Can a deque (double–ended queue) be implemented in a purely functional language such that each push or pop operation on either end of a deque is accomplished in $O(1)$ time in the worst case? The answer is yes, thus solving a problem posted by Gajewska and Tarjan [27] and by Ponder, McGeer, and Ng [62], and refining results of Sarnak [64] and Hoogerwoord [35].

We term such a deque *real–time*, since its constant worst–case behavior might be useful in real time programs (assuming real–time garbage collection [5], *etc.*). Furthermore, we show that no restriction of the functional language is necessary, and that push and pop operations on *previous* versions of a deque can also be achieved in constant time.

We present a purely functional implementation of real–time deques and its complexity analysis. We then show that the implementation has some interesting implications, and can be used to give a real–time simulation of a multihead Turing machine in a purely functional language.

6.1 Introduction and Survey

The general techniques described by Driscoll, Sarnak, Sleator, and Tarjan [23]. for making aggregate data structures fully persistent often rely on side–effects to achieve good time and space performance. It is not clear how they can be implemented in a purely functional

we view this as a common problem faced by sharing aggregate data structures in a parallel system, not a problem caused by the fragmented shallow binding scheme *per se.* On the other hand, the fragmented shallow binding scheme also has its advantage because its data structure is more likely to be fragmented and to be distributed well.

of these multi–threaded read patterns are voluminous.

We can take the full histogram problem as an example in the multi–threaded case. The full histogram problem is to classify a sequence of incoming events into a fixed set of categories, and to support queries about either the distribution of events among the categories in a certain past time step, or to query the accumulation of the events of a particular category in a certain sequence of past time steps. The queries can occur while events are still occurring. It is common to represent the fixed set of categories as an array and have each entry of the array store the number of events which have occurred so far in the given category. To answer a query about distribution of events, all entries of the array have to be read. Also, the accumulation of events of a category can be accessed by a straight–line sequence of reads to the arrays in the version tree. Both the distribution query and the evolution query are implemented by voluminous reads.

We give empirical results collected from the execution of four sample programs to support our argument. The programs are for the multiplication of two matrices (each of size 100×100), all the safe positions for 8 queens, the transitive closures of a graph of 100 nodes, and the simulation of a histogram involving 10,000 events in 100 categories. The Warshall [80] algorithm is used in the transitive closures program, and all intermediate closures (which are those considering only paths through node 0 to node $i, 0 \leq i \leq 99$) are preserved.

Except for the matrix multiplication program, all programs are multi–threaded. After the results are computed, each program also reads every entry of the resulting arrays. We do this because otherwise the fragmented shallow binding scheme will have a clear advantage over other implementation schemes. This is because much of the work in the fragmented shallow binding scheme only occurs when reads are performed to the resulted arrays. Also, no effort is spent in partitioning the *Read* operations into voluminous sequences in the fragmented shallow binding implementation; the ordering of *Read* operations is left as it is specified by the program.

Table 5.1 shows the total number of *Create*, *Update*, and *Read* operations being performed; the total number of root nodes and non–root nodes in the resulted data structure; and the total number of rerootings and rotations being performed. We can see that

(the number of non–root nodes) + (the cache size) · (the number of root nodes),

Corollary 5.4.55 A sequence of m *Read* and *Update* operations, which starts from a single *Create* n operation and includes u *Update* operations, can be implemented in $O(n+m)$ time and $O(n+u)$ space by the fragmented shallow binding scheme if all the *Read* operations in the sequence can be partitioned into disjoint voluminous subsequences. It does not matter if *Update* operations are mixed in the voluminous sequences of *Read*. □

PROOF. Since an *Update* operation does not change the weight of the region to which it is applied, *Update* does not change the amortized cost analysis for *VRead* in the proof of Lemma 5.4.53. However, the amortized cost of the *Update* operation is now incorporated in the summarization of the amortized cost for the voluminous *Read* operations. But this does not change the total amortized cost of the entire m operations either. The proof then follows immediately from theorem 5.4.54. ◇

Note that disjoint voluminous sequences of reads often arise naturally in many functional array applications due to the dense locality of their accesses to arrays. Compile–time partitioning of *Read* operations into voluminous sequences may not be necessary in most cases to achieve good running time. Also the fragmented data structure resulting from the fragmented shallow binding scheme is likely to make voluminous sequences of reads disjoint. This is because there are more caches in the version tree, and a subsequence of a previous non–voluminous read sequence may becomes voluminous if the subsequence has an easy access to a local cache. The empirical results in section 5.5 will show that the above observations are valid.

5.5 Examples and Empirical Results

Is it practical to use the fragmented shallow binding scheme to implement functional arrays? Before answering this question, we have to argue first that functional arrays are often used in an incremental updates/voluminous reads style. In single–threaded accesses to functional arrays, the entire read sequence is voluminous because each read is applied to the newly updated array and the total number of rotations needed in all the read operations is bounded by the the total number of updates being performed. For multi–threaded applications of functional arrays, we observe that many of them will read every entry of an array if there is ever a need to read the array. If a long sequence of reads is not directed solely to a single array, it is usually directed to a sequence of closely related arrays. Both

The first operation in the sequence is a *Create* n operation. It needs $n+1$ units of actual time, and space, to allocate and initialize both a cache of size n and a root node. Before the *Create* n operation, we have $\Phi_0 = 0$, and, after the operation, $\Phi_n = -3n$. Therefore, the total amortized time for the first n individual operations is described by

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} t_i + (\Phi_n - \Phi_0) = (n+1) - 3n = -2n + 1.$$

Suppose that the ith individual operation, $n < i \leq m$, is an *Update* operation. It will need 1 unit of actual time and 1 unit of space to complete the operation. The *Update* operation also increase the potential by 3 by adding a new non–root node. The amortized time a_i is then

$$a_i = t_i + (\Phi_i - \Phi_{i-1}) = 1 + 3 = 4.$$

Suppose that the ith to $(i+l-1)$th individual operations constitute a *VRead* operation of volume l. By lemma 5.4.53, the total amortized time for the *VRead* is $O(l)$. That is, it is less than $C_i \cdot l$ for some constant $C_i \geq 0$.

The total amortized cost for the m individual operations is then

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{n} a_i + \sum_{i=n+1}^{m} a_i \leq (-2n+1) + C(m-n),$$

for some constant $C = max\{4, C_{i_1}, C_{i_2}, \ldots, C_{i_j}\}$. Constants $C_{i_1}, C_{i_2}, \ldots, C_{i_j}$ are derived from the j *VRead* operations in the whole sequence.

If the final configuration of the data structure has $k > 1$ regions, then, by Lemma 5.3.50, each region has at least $\lfloor \frac{2n+1}{3} \rfloor$ non–root nodes. Therefore, we have its potential as

$$\Phi_m \geq 9k \left\lfloor \frac{2n+1}{3} \right\rfloor - 3kn + 0 \geq 9k \frac{2n-1}{3} - 3kn + 0 = 3k(n-1) \geq 0,$$

when the array size $n \geq 1$. If the final configuration has only one region, then its potential is

$$\Phi_m \geq 9 \cdot 0 - 3n + 0 = -3n.$$

Recall that $\Phi_0 = 0$, so the total actual time for the entire m individual operations is

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i - (\Phi_m - \Phi_0) \leq (-2n + 1 + C(m-n)) + 3n.$$

The total actual time is $O(m)$ because $m \geq n$. It is clear that the data structure uses only $O(n+u)$ space, where u is the total number of *Update* operations in the sequence.$\diamond$

In the above, we assume that region s is of depth greater than w, the deepest region ever crossed. Suppose that $s = w$ is crossed as in case 4. Then the above analysis still holds except that all the crossing to w may be of case 4. In such a case, we can combine them into a single case 4 crossing. The combined amortized cost is $2\sum_{j\in J} d_j + 2q$, for some subset $J \subseteq \{1, 2, \ldots, k-2\}$. Furthermore, since the rerooting does not involve any cutting, we have $q \leq 2n$.

Summarizing the total amortized cost for $s_j, 1 \leq j \leq k-2$, we have

$$\sum_{j=1}^{k-2} a_j = 2q + 2\sum_{j=1}^{k-2} d_j \leq 4n + 2\sum_{j=1}^{k-2} d_j.$$

It is not difficult to see that crossing the first region, s_0, and crossing the last region, s_{k-1}, each cost at most $4n + 2d_0$ and $4n + 2d_{k-1}$ amortized time, respectively. Therefore, we have the total amortized time

$$\sum_{j=0}^{k-1} a_j = (a_0 + a_{k-1}) + \sum_{j=1}^{k-2} a_j \leq 12n + 2d.$$

Adding both the cost for read (which is l) and rotation (which is at most $12n + 2d$), the total amortized time for a *VRead* of volume l is at most $l + 12n + 2d$, which is $O(l)$ because l is $\Omega(n)$ and d is $O(l)$. ◇

We then can show the following main result.

Theorem 5.4.54 Let a *VRead* operation of volume l be counted as l individual operations, a *Create* n operation be counted as n individual operations, and an *Update* operation be counted as one individual operation. Then a sequence of *Create, Update*, and *VRead* operations will cost $O(m)$ time and $O(n + u)$ space by the fragmented shallow binding scheme if the sequence has m individual operations, starts from a single *Create* n operation, and has u *Update* operations. □

PROOF. Let a_i and t_i denote respectively the amortized and actual time for the ith *individual* operation in the sequence, and Φ_{i-1} and Φ_i respectively be the potential of the data structure before and after the ith individual operation. Recall that the potential of a configuration D of the data structure is defined as

$$\Phi(D) = 9 \cdot N(D) - 3n \cdot R(D) + \sum_{S\in D} W(S).$$

Note that $W(s'_j) \leq d_j$ because the weight of region s_j is 0 when the crossing starts at its b–gate and each rotation only increases the weight of s_j at most by one.

- Case 2. Assume that, as pictured, the rerooting first reduces the weight of s_j from $p+r$ to r, then increases it to $q+r$. The actual cost for the rerooting is then $p+q$. For the rotations, the actual cost is d_j. It is clear that $0 \leq p, 0 \leq q, 0 \leq r$, and $r+q \leq d_j$. Then, we have amortized cost

$$a_j = t_j + (\Phi'_j - \Phi_j) = (p+q+d_j) + (0-(p+r)) = d_j + (q-r) \leq 2d_j.$$

- Case 3. Similar to case 1, except that the weight of s_j after the crossing is 0. We then have

$$a_j = t_j + (\Phi'_j - \Phi_j) = (W(s_j) + d_j) + (0 - W(s_j)) = d_j.$$

- Case 4. Similar to case 2, except that the weight of s_j after the crossing may be as large as $q+r+d_j$. We then have

$$a_j = t_j + (\Phi'_j - \Phi_j) \leq (p+q+d_j) + ((q+r+d_j) - (p+r)) = 2d_j + 2q.$$

Except for case 4, the amortized cost for crossing region s_j is bounded by $2d_j$.

Let w denote the region which has the least depth among the crossed regions. By proposition 5.4.52, such a region is unique. Suppose that a region s of depth greater than w is crossed as in case 4. Let region s be one of the regions s_j for some $1 \leq j \leq k-2$. Then either region s was crossed as $s_{j'}, j' < j$, by case 1 in order to exit from region w, or s will be crossed as $s_{j'}, j' > j$, by case 2 in order to enter region w. Furthermore, by proposition 5.4.52, the two crossings s_j and $s_{j'}$ will use the same f–gate as the exit/entrance gate. Therefore, no rerooting is necessary for the transition from $s_{j'}$ to s_j (or from s_j to $s_{j'}$).

We can then combine s_j and $s_{j'}$ as they are a single crossing in region s. The combined crossing will be of case 1 or 2 depending of what case $s_{j'}$ is. The combined amortized cost is then less than $2(d_j + d_{j'})$. It is possible that between s_j and $s_{j'}$, there are other case 4 crossings on the same region. If so, we can combine those case 4 crossings into a single case 4 crossing. As an example, in figure 5.3, we can combine the crossings of regions S, T, and W as a single case 2 crossing. (Note that regions S, T, and W in fact are the same region.)

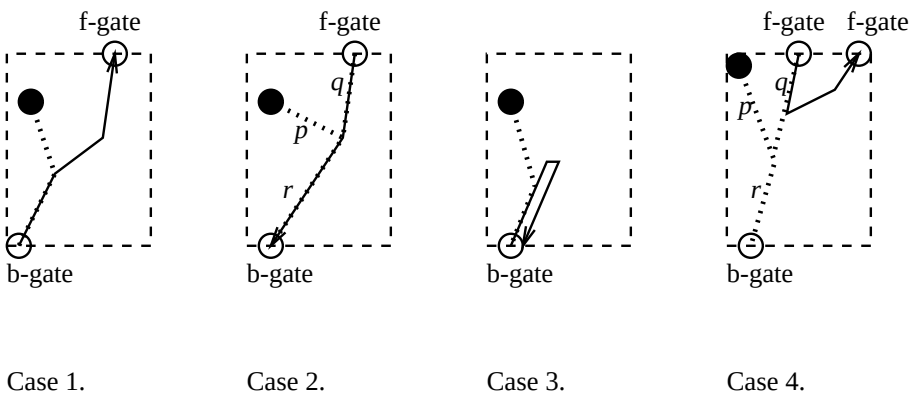


Figure 5.4: Four ways to cross a region.
Note. A dashed box denotes a region. Each black node represents the root node before the crossing. The direction of each crossing is depicted by the solid line with an arrow head. The dotted line is used to illustrate the changing weight of the region.

Assume that the walk goes through, in order, regions $s_0, s_1, \ldots, s_{k-1}$. Also assume that d_i rotations are needed by the walk to cross region $s_j, 0 \leq j \leq k-1$. We then have $\sum_{j=0}^{k-1} d_i = d$. For region $s_j, 1 \leq j \leq k-2$, there are four possible ways for a walk may cross the region. They are shown in figure 5.4. The amortized cost for each case will be analyzed. Let us assume that a region s_j has weight $W(s_j)$ and potential Φ_j before the crossing, and weight $W(s'_j)$ and potential Φ'_j afterward.

The actual cost for crossing a region s_j includes two parts: the cost for rerooting, which makes the entrance gate of the region the root node, and the cost for d_j rotations. Recall that we have shown in lemma 5.4.51 that if a rerooting involves cutting then its amortized cost is less than or equal to 0. Therefore, it suffices to analyze the situation where the rerooting does not involve any cutting.

The amortized costs for crossing each of the four cases in figure 5.4 are the following.

- Case 1. The actual cost t_j equals $W(s_j) + d_j$ because the crossing needs $W(s_j)$ time, exactly the weight of region s_j, to move the root node to the b–gate, and needs additional d_j time to rotate the root from the b–gate to the f–gate. We then have amortized cost

$$a_j = t_j + (\Phi'_j - \Phi_j) = (W(s_j) + d_j) + (W(s'_j) - W(s_j)) = d_j + W(s'_j) \leq 2d_j.$$

region denoted by S, T, and W being the region of nodes E and F, and the region denoted by V being the region of node G. See also figure 5.2.

However, the implementation of *VRead* as described in section 5.3 does not use such a walk to accomplish a *VRead* operation. A *VRead* need not to move the root node of a region to the entrance gate whenever the next read operation enters the region from another region. Instead, it only moves the root node to the first node in the region to be read. Neither does a *VRead* need to move the root node of a region to the exit gate whenever the next read operation leaves the region to another region. Instead, the root node remains at the last node in the region being read. As a result, a *VRead* will use less rotations than the walk sequence described above does. For example, the implementation described in section 5.3 only uses the following operations to accomplish the same *VRead* operation.

> G (0 rotation, 1 read), E (1 rotation, 1 read), G (0 rotation, 1 read), F (1 rotation, 1 read), G (0 rotation, 1 read), F (0 rotation, 2 read), A (1 rotation, 1 read), B (1 rotation, 1 read), C (1 rotation, 1 read), D (1 rotation, 1 read).

It uses only 6 rotations instead of 7 rotations required by a walk. The concept of a walk is introduced to prove the following crucial lemma.

Lemma 5.4.53 A *VRead* operation of volume l costs $O(l)$ amortized time. □

PROOF. By the definition of *VRead*, the sequence of arrays being read forms a path of length $d = O(l)$ in the region tree. In the worst cases, the *VRead* operation must be accomplished by a walk of d steps in the region tree. For each step during the walk, two tasks are performed: a rotation between the current root node and the next node in the path (plus bridge–crossing if the two nodes reside in different regions), and possibly several reads in the new root node. In the following, we will count the read cost and the rotation cost in a *VRead* operation separately.

However, there is one complication: at the beginning of a *VRead*, a rerooting may be needed to make the first node in the walk the root node, and, if the walk crosses several regions, a rerooting may be needed every time the walk crosses a bridge between two regions (which is to make the walk's first node in the destination region the root node).

It is clear that a read to a root node costs one unit of time and the operation changes no potential of the data structure. Therefore, without counting the rotation cost, the total read cost for a *VRead* of volume l is l. It remains to count the cost for rotations.

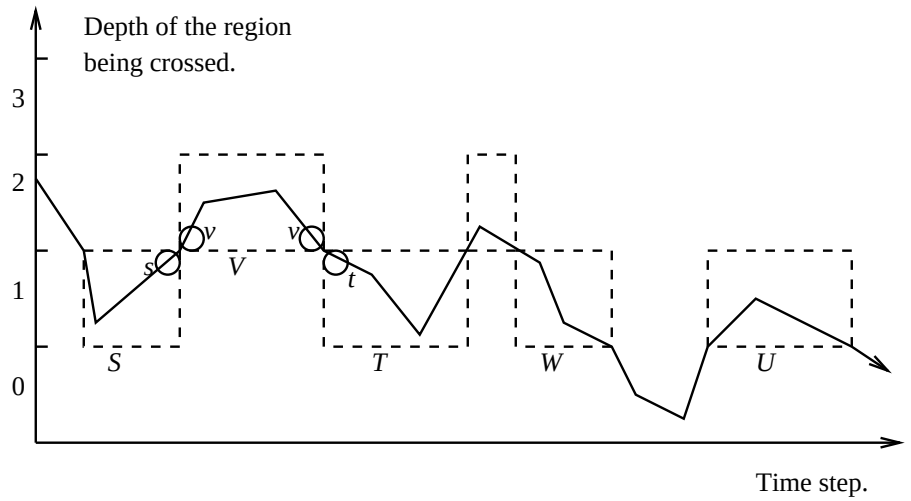


Figure 5.3: A walk in a region tree.
Note. A dashed box denotes a region. Regions S, T, W, and U have depth 1, and region V has depth 2. Notice that S, T, and W are in fact the same region, and s and t the same f–gate.

Let us give an example. Suppose that the following *VRead* operation,

$$VRead\ [G, E, G, F, G, F, F, A, B, C, D]\ [i_0, i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8, i_9, i_{10}],$$

where $0 \leq i_k \leq 1$ for each array index i_k, is issued to the final configuration in figure 5.2. The length of path formed by the above *VRead* is 13. (2 for G to E, 2 for E to G, 1 for G to F, 1 for F to G, 1 for G to F, 0 for F to F, 3 for F to A, 1 for A to B, 1 for B to C, and 1 for C to D.) Therefore, the *VRead* can be accomplished as a walk of 13 steps in the region tree. Each step of the walk first makes the current visited node the root node, then performs a *Read* operation if required. For example, the following 13–step walk accomplishes the above *VRead* operation.

> G (0 rotation, 1 read), F (0 rotation, 0 read), E (1 rotation, 1 read), F (1 rotation, 0 read), G (0 rotation, 1 read), F (0 rotation, 1 read), G (0 rotation, 1 read), F (0 rotation, 2 read), E (1 rotation, 0 read), B (0 rotation, 0 read), A (1 rotation, 1 read), B (1 rotation, 1 read), C (1 rotation, 1 read), D (1 rotation, 1 read).

Each of the above steps performs the operations in the parenthesis on the currently visited node. The rotations are used to make the current node the root node, and the reads fetch the required data. The course of the above walk is roughly pictured by figure 5.3, with the

Before establishing the amortized time complexity for other operations, we state in Proposition 5.4.52 two simple observations. We define a *walk* in a region tree as a sequence of either rotation between adjacent nodes in a region or, if the walk crosses several regions, the crossing of bridges between the regions (where each bridge consists of a b–gate/f–gate pair of nodes). We also define the *depth* of a region in the region tree as the number of bridges between itself and the root region. For example, in the final configuration in figure 5.2, the region including A, B, C, and D has depth 0, while the region including E and F has depth 1 and the region including G has depth 2.

Proposition 5.4.52

1. If a walk leaves a region s via a b–gate/f–gate g, then the same walk will enter region s only via gate g during its first return to s (if the walk does return to s).
2. During a walk, if the current position is in a region of depth k, then the walk cannot lead to another different region also of depth k without first going to a region of depth $k-1$.

□

Proof Outline.

1. If a walk leaves a region s via a b–gate/f–gate g, and then first re–enters region s by a bridge other than g, then there is region outside s which has two different paths leading to s. This is contradictory to that facts that all the region nodes forms a region tree.
2. The same argument as the above. If the walk can lead to another region also of depth k without going to a region of depth $k-1$, then there are two ways from the current region to its parent region. This contradicts to the fact that all the regions nodes form a region tree.

◇

We can draw the course of a walk on a plane with the time step as the X–axis and the depth of the current visited region as the Y–axis, as in figure 5.3. According to Proposition 5.4.52, in the specific walk illustrated in figure 5.3, regions S, T, and W are the same region, but they may differ from region U. Furthermore, the f–gates s and t are the same node because region V has only one b–gate, v, and v is bridged to both s and t.

as

$$\Phi(D) = 9 \cdot N(D) - 3n \cdot R(D) + \sum_{S \in D} W(S)$$

where $N(D)$ is the total number of non–root nodes in D, $R(D)$ is the total number of root nodes in D, n is the size of array, $S \in D$ is used to denote S being a region of D, and $W(S)$ is the weight of a region S. The weight of a region is defined as the distance between the root node and the b–gate in the region. It is clear that the minimal weight of a region is 0 and the maximal weight can be as large as the total number of non–root nodes in the region. Also note that a region's weight stores the exact amount of energy to move the root node of the region to the region's b–gate.

The intuition and justification behind the definition of the above potential function is that the more root nodes (*i.e.*, the more regions, hence, the more caches) a configuration has, the less potential it keeps; and, for each region, the more distant its cache is from its b–gate, the more potential it has. The constants 3 and 9 in the potential function Φ may vary if we change the way cutting is performed during a long rerooting. The constant 3 is chosen here because a path is cut only when its length is roughly 3 times the size of the array. It also makes the following lemma true.

Lemma 5.4.51 If a reroot operation involves cutting the data structure, then the amortized time of the rerooting is less than or equal to 0. □

PROOF. If cutting occurs during a rerooting, then the distance d between the old root node and new root node in the original region must be greater then $2n$, where n is the array size. Let $d = (k+3)n - r$, where $0 \leq k$ and $0 \leq r \leq n-1$. After the rerooting, the original region is cut into $k+3$ regions, and the difference between the total weight of the $k+3$ regions and the weight of the original region is less than or equal to d. That is, $\sum_{S \in D'} W(S) - \sum_{S \in D} W(S) \leq d$, where D and D' are, respectively, the configurations of the data structure before and after the rerooting. It follows that $\Phi(D') - \Phi(D) \leq -3n(k+2) + d$.

Let a be the amortized time of the rerooting and t be the actual time. It is clear that d units of actual time suffices to perform the rerooting. We then have the amortized time

$$a = t + (\Phi(D') - \Phi(D)) \leq d + (-3(k+2)n + d) = -kn - 2r \leq 0.$$

◇

PROOF. A *Read* A_j i operation cuts the version tree if and only if the distance d between the node p pointed to by A_j and the root node of the tree in which p resides is larger than $2n$. Let $d = kn - r$, where $3 \leq k$ and $0 \leq r \leq n-1$. The path is cut into $\lceil \frac{d}{n} \rceil = k$ segments with each segments containing at least $\lfloor \frac{d}{k} \rfloor$ nodes. All these nodes are non–root nodes.

But $\lfloor \frac{d}{k} \rfloor = \lfloor \frac{kn-r}{k} \rfloor = \lfloor n - \frac{r}{k} \rfloor \geq \lfloor n - \frac{n-1}{3} \rfloor = \lfloor \frac{2n+1}{3} \rfloor$. ◇

5.4 Time and Space Complexity of the Fragmented Shallow Binding Scheme

We will use the potential method described by Tarjan [73] to analyze the amortized time complexity of a sequence of *Update* and *VRead* operations, starting from a single *Create* operation. A potential function Φ maps any configuration D of a data structure into its potential $\Phi(D)$, which can be viewed as the amount of energy stored in the configuration. The *amortized time* of an operation, a, is defined by

$$a = t + \Phi(D') - \Phi(D),$$

where t is the actual time needed by the operation, and D and D' are the configurations of the data structure before and after the operation, respectively. We can regard amortized time as a fixed amount of cost. If this fixed amount is larger than the actual need, then the remaining amount is stored in the data structure for future use. If the fixed amount is less than the actual need, then the energy released from the data structure will compensate the difference. The potential function Φ describes how much energy a data structure has.

With this definition, the total actual time for a sequence of m operations can be written as

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (a_i - (\Phi_i - \Phi_{i-1})) = \sum_{i=1}^{m} a_i - (\Phi_m - \Phi_0),$$

where t_i and a_i are the actual time and amortized time of the ith operation, respectively, and Φ_i is a shorthand for $\Phi(D_i)$. Φ_0 is the initial configuration of the data structure, and Φ_m is the final one. If the difference between Φ_m and Φ_0 remains positive, then the total amortized time is an upper bound of the total actual time.

We define the potential of a configuration D in the fragmented shallow binding scheme

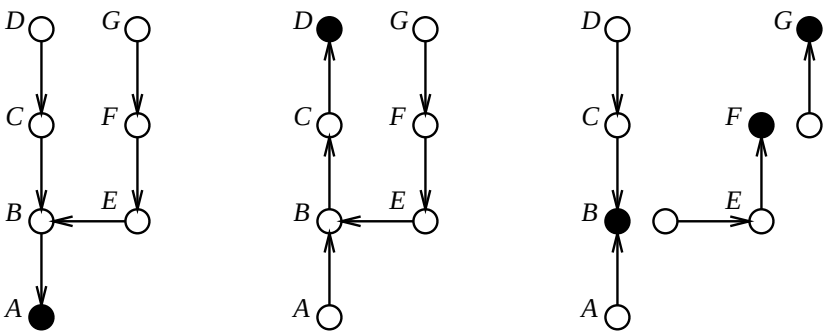


Figure 5.2: The fragmented shallow binding scheme, as illustrated by configurations of a version tree after two rerootings.
Note. The array size is 2. Black nodes denote root nodes and white nodes denote non–root nodes. Each of the root nodes points to a cache of size 2, which is omitted in the picture.

to see that the resulting region graph is also a tree.[2] Let us call it a *region tree*. Note that the root region in the region tree has no b–gate and the leaf regions have no f–gates. To simplify the complexity analysis, let us assign the b–gate of the root region to the original root node of the version tree (as if no rerooting had ever occurred). Also note that the locations of b–gates and f–gates in a region will not change once they are created, regardless of further operations being performed on the data structure. For example, in the final configuration in figure 5.2, the root region including nodes A, B, C, and D has A as b–gate and B as f–gate. A is assigned to be a b–gate because it is the original root node in the root region. The region including nodes E and F has E as b–gate and F as f–gate. The region including node G has G as b–gate but has no f–gate.

Lemma 5.3.50 If a *Read* operation involves cutting the version tree, then each resulting region contains at least $\lfloor \frac{2n+1}{3} \rfloor$ non–root nodes, where n is the array size. □

[2]Suppose it were not true. Then a non–tree–like graph can be constructed by connecting all the regions, which are trees, by restoring all the links between them that have been cut. But this graph is the original version tree, a contradiction.

Call these segments $s_0, s_1, \ldots, s_{k-1}$. Let H_i and T_i respectively be the first node and the last node of segment $s_i, 0 \leq i \leq k-1$. Note that H_0 is the root node and T_{k-1} is node p.

Do a rerooting at node T_0, duplicate the cache pointed to by T_0, allocate a new root node and have it point to the new cache. Make H_1 point to this newly allocated root node. Repeat the above procedure for T_1 and H_2, T_2 and $H_3, \ldots, T_{k-2}$ and H_{k-1}, and T_{k-1}. When all rerooting is done, we have cut the tree in which node p originally resides into k disjoint trees such that each tree has its own root node (which points to its own cache).

After the cutting and rerooting, A_j points to root node T_{k-1} which again points to a cache. Return the ith entry in the cache.

- *VRead* $[A_{j_0}, A_{j_1}, \ldots, A_{j_{l-1}}]$ $[i_0, i_1, \ldots, i_{l-1}]$:
  Perform (in their specified ordering) the following operations: v_0 = *Read* A_{j_0} i_0; v_1 = *Read* A_{j_1} i_1; ...; and v_{l-1} = *Read* $A_{j_{l-1}}$ i_{l-1}. Return $[v_0, v_1, \ldots, v_{l-1}]$.

Figure 5.2 illustrates what a version tree of an array of size 2 will look like after two rerootings under the fragmented shallow binding scheme. The three configurations in figure 5.2 can be thought as resulting from the following three successive sequences of operations,

- A = *Create* 2; B = *Update* A i_b b; C = *Update* B i_c c; D = *Update* C i_d d; E = *Update* B i_e e; F = *Update* E i_f f; G = *Update* F i_g g;
- *Read* D i; and
- *Read* G j;

where $i_b, i_c, i_d, i_e, i_f, i_g, i$, and j are array indices.

In order to analyze the time and space complexity of the above implementation, let us first introduce some terminology. Since a rerooting may cut a tree into a set of disjoint trees, the resulting data structure is more likely to be a forest rather than a tree. Among the many nodes in a particular tree of the forest, we will use the term *backward gate* (*b–gate* for short) for the node whose link between itself and its parent has been cut. Similarly, a *forward gate* (*f–gate* for short) is a node whose link to one of its children has been cut.

Suppose that each tree in the forest is viewed as a *region* and an imaginary bridge is used to connect those regions that were connected previously. Then it is not difficult

a long rerooting, but at the same time keep the total numbers of entries in all caches proportional to the total numbers of updates being performed. By performing this trick, we make sure that the large cost of a long rerooting can be paid off by the small costs of future shorter rerootings. By keeping the total number of entries in caches proportional to the total number of updates, we make sure that only linear space (with respect to the total number of updates) is used.

Before describing the details of the new implementation scheme for efficient incremental updates and voluminous reads, let us first define *VRead*, the voluminous read operation.

- *VRead* $[A_{j_0}, A_{j_1}, \ldots, A_{j_{l-1}}]$ $[i_0, i_1, \ldots, i_{l-1}]$: Return a list of l elements, where the list's kth element has value $A_{j_k}(i_k)$. It is required that l be $\Omega(n)$ and the path consisting of $A_{j_0}, A_{j_1}, \ldots, A_{j_{l-1}}$ in the version tree be of length $O(l)$, where n is the array size.[1]

The volume of the above *VRead* operation is defined to be l, and the length of the path consisting of $A_{j_0}, A_{j_1}, \ldots, A_{j_{l-1}}$ in the version tree is simply the total number of links from A_{j_0} to A_{j_1}, from A_{j_1} to A_{j_2}, ..., and from $A_{j_{l-2}}$ to $A_{j_{l-1}}$ in the version tree. Notice that *Read* will also be supported by the new data structure. However, as we will see, an individual *Read* will cost $O(n)$ amortized time in the worst cases.

We now describe the *fragmented shallow binding scheme*, which supports *Create, Update, Read*, and *VRead*. The implementation of *Create* and *Update* are the same as before. *Read* and *VRead* are implemented in the following way.

- *Read* A_j i:
  Let d be the distance between the node p pointed to by A_j and the root node of the tree in which p resides.
  - If $d \leq 2n$, do a rerooting starting from node p and then return the ith entry in the cache pointed to by p. Note that node p is the root node after the rerooting.
  - If $d > 2n$, then cut the path between node p and the root node equally into $k = \lceil \frac{d}{n} \rceil$ segments such that each segment has at least $\lfloor \frac{d}{k} \rfloor$ nodes of the path.

[1] $O(f(x))$ is defined as the set of all functions $g(x)$ such that there exist positive constants C and x_0 with $|g(x)| \leq C\ f(x)$ for all $x \geq x_0$. $\Omega(f(x))$ is the set of all functions $g(x)$ such that there exist positive constants C and x_0 with $g(x) \geq C\ f(x)$ for all $x \geq x_0$ (see [63], for example). In the definition of $VRead$, it simply says that the volume of the read sequence is at least the magnitude of the size of the array, and the length of the path in the version tree for the read sequence is at most the magnitude of the volume of the read sequence.

- *Read* A_j i:
  Do a rerooting starting from the node pointed to by A_j. After the rerooting, A_j points to the root node which again points to a cache. Return the ith entry in the cache.

For example, the three configurations in figure 5.1 can be thought as being resulted from the following three successive sequences of operations,

- $A =$ *Create* 2; $B =$ *Update* A i_x 0; $C =$ *Update* A i_y 1; $D =$ *Update* C i_x 2;
- *Read* B i; and
- *Read* D j;

where i_x, i_y, i, and j are array indices.

We now use the term *version tree* to refer to the context tree in the shallow binding implementation of functional arrays. It is helpful to emphasize that an update operation just adds a node pointing to the node it is updating. Nothing destructive is performed. Because an update sequence need not occur single–threadedly (such that each operation in the sequence only updates the array resulted from the previous operation in the sequence), the resulting version configuration is more likely to be a tree (and not necessarily to be a singlely–linked list).

It is easy to verify that a *Create* n operation takes $O(n)$ time and uses $O(n)$ space. Each *Update* takes $O(1)$ time and additional $O(1)$ space. A *Read* uses no space but takes time proportional to the distance between the new root node and the old root node in the version tree, which can be as large as the total number of updates being performed. However, after a *Read* A_j i operation, each additional read to either array A_j, or one of A_j's children in the version tree, will cost only additional $O(1)$ time. This is because, after the *Read* A_j i operation, A_j points to the root node and, in case the read is directed to one of A_j's children, the rotation between A_j and one of its children only costs $O(1)$ time.

It is not difficult to see why we insist on voluminous reads. After the initial read of a voluminous sequence, each read of the sequence will cost only additional $O(1)$ time. However, there is still one problem. The distance between the new root node and the old root node may be far greater than the volume of the reads, which makes rerooting expensive. The trick is to allocate multiple caches and disperse them as needed during

rotations is called *rerooting*. A rotation is performed between the current environment, which is at the root node in the context tree, and one of its children. Suppose that the current environment E binds variable v to value a in its cache, and its child node n binds v to b. Then a rotation between node E and node n will make n the current environment and make E n's child. Node n will inherit node E's cache and will change variable v's value to b. Node E will only record the binding of variable v to value a; it will not possess the cache anymore. Instead, node E will point to node n in which the cache resides. A rerooting is a sequence of rotations to make the desired environment current.

Figure 5.1 pictures how rotation and rerooting work under the shallow binding scheme. We will show how to use, and to modify, the shallow binding scheme to implement functional arrays in the next section.

5.3 Functional Arrays by Shallow Binding and Fragmented Shallow Binding

A data structure for functional arrays must support the following operations.

- *Create* n: Return an array of size n. Each entry of the array is not initialized.
- *Update* A_j i v: Return an array $A_{j'}$ which is functionally identical to array A_j except $A_{j'}(i) = v$. Array A_j is not destroyed and can be accessed further.
- *Read* A_j i: Return $A_j(i)$.

We can use the shallow binding scheme to implement *Create*, *Update*, and *Read* as the following.

- *Create* n:
  Allocate a cache of size n. Allocate a node of one field and have this field point to the cache. This node is the *root node*. Return the address of the root node.
- *Update* A_j i v:
  Allocate a node of three fields and have it store i, v, and A_j (note that A_j is an address). Return the address of this newly allocated node.

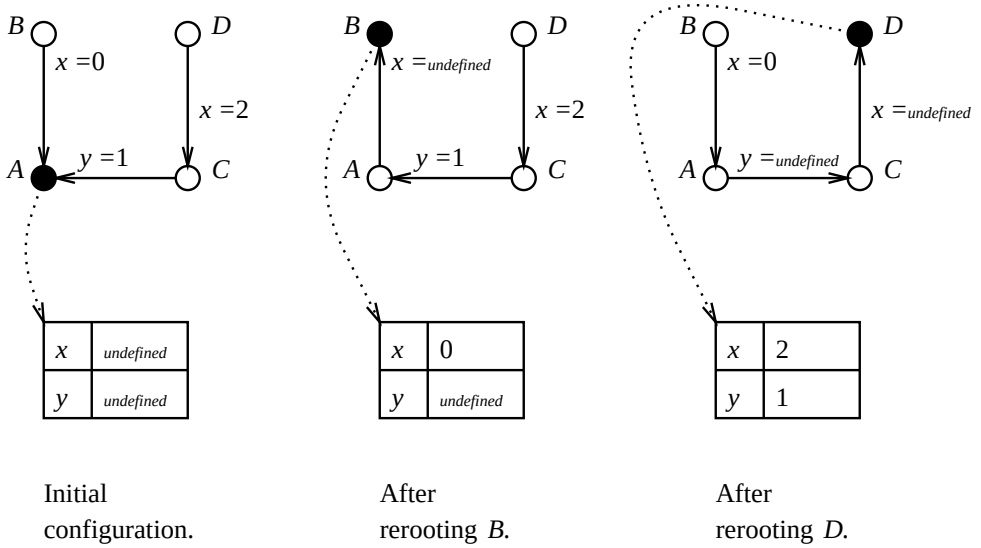


Figure 5.1: The shallow binding scheme, as illustrated by configurations of a context tree after two rerootings.
Note. The first rerooting consists of a single rotation from B to A, and the second rerooting consists of three rotations from D to B. A, B, C, and D are environments, and black nodes denote the current environments. Only the current environment points to the cache. The bindings of variables in each of the four environments do not change between rerooting. What is changed is the implementations of the environments.

value bound to v. Under the deep binding scheme, a context switch between two given environments costs constant time, but a variable lookup in an given environment may cost time linear to the length of its binding path.

In order to improve the performance of variable lookup, Baker [6] developed the *shallow binding* scheme where a *cache* is introduced to record the bindings of variables in the *current environment*. Variable lookup in the current environment is performed by a constant–time reference to the cache. The current environment is also made the root of the context tree. However, only one environment, the current environment, possesses the cache. Other non–current environments in the context tree are still represented as binding paths leading to the current environment at the root.

A context switch between two given environments in the shallow binding scheme will involve a sequence of *rotations* in the context tree. The entire process for a sequence of

those programs always access the newest version of an array. However, they are bad for multi–threaded programs because accesses to old versions are costly.

Previous attempts to make arrays fully persistent do not work well because they use too little random access memory (RAM) to represent multiple versions of an arrays. Instead, they depend mostly on indirect reference by pointers. In fact, previous data structures use at most one set of RAM for multiple versions of an array. An array's random accessibility is then lost once there have been many updates being performed to the array.

We propose in this chapter a variation of Baker's shallow binding scheme which holds multiple sets of RAM. We call this method the *fragmented shallow binding scheme*. This scheme employs multiple sets of RAM, which we will later call *caches*, to improve the efficiency of those programs which access arrays in a multi–threaded way. Although there is still a restriction — reads have to be voluminous — it seems that a major portion of multi–threaded applications do fit the incremental updates/voluminous reads restriction.

The outline of this chapter is the following. After a short introduction to the shallow binding scheme in section 5.2, we describe in section 5.3 how to implement fully persistent arrays by the fragmented shallow binding scheme. Section 5.4 contains detailed complexity analysis of the proposed scheme. Empirical results are presented in section 5.5. Section 5.6 is a short discussion of related issues and future work.

5.2 Deep Binding and Shallow Binding

In a higher–order functional language, where functions can be passed as arguments and returned as results, there may be more than one accessible environment at a given point during a program's execution. In an implementation of a higher–order functional language, it is essentially important to design an efficient data structure to support the following two operations: variable lookup in the current environment and context switch between environments.

In the *deep binding* scheme, the environments are represented as a *context tree* where each node in the tree introduces a new binding of a value to a variable and has a pointer to its parent node. An environment is represented as a path from its most recent binding to the initial binding, at the root node, in the context tree. The lookup for a variable v in an environment E is performed by searching the path of E, looking for the most recent

is assigned a "time–stamp," with the restriction that an update operation with a smaller time–stamp occurs before update operations with greater time–stamps. Entry $B[i]$ of the array B is implemented as a balanced search tree, with each node of the tree containing two fields, t and v, where t is the time–stamp of the update operation and v is the value assigned to entry $A[i]$ by the update operation. The balanced trees use time–stamps as search keys.

An update operation $A[i] := v$ can be implemented as follows: fetch the current time–stamp t, use it as the key to store value v in the balanced search tree at entry $B[i]$ of the array B, and return t as a reference to the newly updated array. Similarly, given a time–stamp t, which denotes the version of the array A at time t, a read operation $A_t[i]$ will search the balanced tree at entry $B[i]$ of the array B for the node of largest time–stamp less than or equal to t, and return the value v stored in the node. A new array is created by returning an array B with each of its entries initialized to be an empty search tree. Let m_i be the total number of update operations being applied to entry $A[i]$ so far. Then this implementation of partially persistent arrays will cost $O(\log m_i)$ time for an update or a read operation to entry $A[i]$ of the array A. Creating a new array will cost $O(n)$ time, where n is the array size.

Based on algorithms for maintaining order in a list [22] and other techniques, Dietz [21] proposes a sophisticated improvement to the above implementation such that it can be used for fully persistent arrays, and it costs $O(\log \log n)$ amortized time per array access.

A common drawback of the above balanced search tree schemes is that much of an array's constant–time accessibility is lost, even if the array is used in a single–threaded manner. Some techniques, based on *reversible difference lists* or *trailers*, have been used to implement fully persistent arrays in mostly single–threaded applications. They can be found, for examples, in the works of Holmstrom [33], Hughes [42], Aasa, Holmstrom & Nilsson [1] and Bloss [10,11]. These techniques are variations of the shallow binding scheme devised by Baker [6,7]. Under these techniques, which will be fully explored in section 5.3, an access to the newest version will take $O(1)$ time, but an access to an old version will take time linear to the number of modifications between the old version and the current version. Any given version of an array can be made the newest version too, also at a cost linear to the number of modifications between the given version and the current newest version of the array. These techniques are good for single–threaded programs because

Previous methods for implementing fully persistent arrays are briefly surveyed. We then give the motivation and the intuition behind our method, and give a detailed analysis of its space and time complexity. Empirical results have also been collected to measure the effectiveness of the proposed data structure.

5.1 Survey and Motivation

There has been much effort to make arrays persistent such that, during a series of modifications, the old versions, as well as the newest version, of an array can still be accessed. As shown by Myers [54], an array of size n can be easily made fully persistent if it is represented as a balanced search tree of n leaves, with array indices as search keys. A new array is created by simply returning a empty search tree. An array read operation is implemented as a search on the designated array index to find its associated value in the balanced tree. An array update operation is then a search followed by a change to the associated value of the designated array index. Because an update operation must also retain the old array, it returns a search tree that only differs from the old search tree in the branch leading from the root node to the leaf node at which the update operation modifies. The old search tree is left undisturbed, and it shares with the new tree all the other branches. All of the above operations can be implemented in a purely functional language such that creating a new array costs constant time and consumes constant space, a read operation costs $O(\log n)$ time and consumes no additional space, and an update operation costs $O(\log n)$ time and consumes $O(\log n)$ space.

If the above balanced search tree scheme is implemented in a side–effecting language, and the technique of Driscoll, Sarnak, Sleator, and Tarjan [23] is used, then an update operation can be implemented in $O(1)$ amortized space, instead of $O(\log n)$ worst–case space. The time and space complexities all other operations remain unchanged, though.

There are other methods to make arrays fully persistent, also based on balanced search trees. An array A can be made *partially* persistent by using an array B of balanced search trees, with each entry $B[i]$ of array B a balanced search tree recording all the update operations ever occurring to entry $A[i]$ of array A. Because only partial persistence is allowed for the array operations, we can assume that all the update operations (to all entries of the array A) occur on the most recent versions of A. Each update operation

Chapter 5

Fully Persistent Arrays for Functional Programs

The array update problem in a purely functional language is the following: once an array is updated, both the original array and the newly updated one must be preserved to maintain referential transparency. We devise a very simple, fully persistent data structure to tackle this problem such that

- each incremental update costs $O(1)$ worst–case time,
- a *voluminous* sequence of r reads cost in total $O(r)$ amortized time, and
- the data structure use $O(n + u)$ space, where n is the size of the array and u is the total number of updates.

A sequence of r reads is voluminous if r is $\Omega(n)$ and the sequence of the arrays being read forms a path of length $O(r)$. A voluminous sequence of reads may be mixed with updates without affecting either the performance of reads or updates.

An immediate consequence of the above result is that if a functional program is single–threaded, then the data structure provides a simple and efficient implementation of functional arrays. This result is not new. What is new is that many multi–threaded functional array applications also exhibit the incremental updates/voluminous reads execution pattern. Those applications can also be efficiently implemented by the proposed data structure.

an implementation of fully persistent arrays which is suitable for multi–threaded application. Assuming some common locality of array accesses, an update operation to any version of array is shown to cost $O(1)$ time in the worst cases, and a read operation in a sufficiently long sequence is shown to cost $O(1)$ amortized time. The implementation, however, uses side–effects to achieve good performance. Hence, functional arrays must be built into the implementation of a purely functional language in order to provide good efficiency. We cannot use a purely functional language which has no built–in efficient implementation of functional arrays to efficiently implement fully persistent arrays.

In Chapter 6, however, we show that real–time deques can be efficiently implemented in purely functional languages, such that every operation to any version of a deque costs only $O(1)$ time in the worst cases. The technique of Driscoll, Sarnak, Sleator and Tarjan [23], instead, will need an imperative language, and can only achieve $O(1)$ amortized time per operation. We also show that this implementation of real–time deques has some interesting consequences, and leads to a real–time simulation of a multihead Turing machine in a purely functional language. An execution step of a k head Turing machine will take only $O(k)$ steps in a purely functional language to simulate, regardless of the total number of the printed symbols on the tape of the Turing machine.

Before leaving this chapter, however, we shall also mention the disadvantage of solving the aggregate update problem by implementing fully persistent aggregates. A common drawback is the overhead for maintaining multiple versions of an aggregate data structure, even when only the newest version is needed. The storage occupied by inaccessible versions of an aggregate will have to be reclaimed. The fully persistent aggregate approach is better seen as a complement to the approaches described in the two previous sections. If a compile time analysis or run time reference counting has detected places where updates can be performed safely in place, then we are free to do so. If all the previous approaches fail to detect single–threaded accesses to an aggregate, or if it becomes too expensive to perform the detection, and multiple copies of an aggregate have to be made available to preserve the value semantics of the functional program, then efficient implementations of fully persistent aggregates shall be used to avoid naive duplications.

We can describe the general technique in [23] as follows. Each node in the ephemeral linked data structure is enlarged in its fully persistent implementation such that each field in an ephemeral node has a constant number of "spare" fields in its corresponding fully persistent node. A modification to a field in the ephemeral node will be implemented by storing the new field value in one of those blank spare fields, without destroying the value in the old field. A version number is accompanied by each modification and is also recorded in the newly written field. This number will latter be used in navigating the fully persistent structure to access the correct version of the aggregate. If all the spare fields in a node are used and there is another update operation to the node, a new node (whose spare fields are all blank) is created, the value of the new field is stored in the new node, and the parent nodes are updated to point to this new node. The update to a parent node is just another modification to its fields which is again implemented by the above mentioned procedure. If all the spare fields of a parent node are used, then a new parent node is created and the grandparent node is modified accordingly, and so on.

This is a very general technique and has wide application. However, there are some restrictions. One restriction is that the technique only applies to linked data structures with bounded in–degree. If the linked data structure has unbounded in–degree, then it can have unbounded number of parent nodes. A modification to a node whose spare fields are all used will incur modification to all of its parent nodes, whose number is unbounded, and will require much time. For aggregates such as arrays, which are not linked structures, the technique is not applicable either. The technique also uses side–effects to make an aggregate fully persistent. This means that the technique is not applicable if we want to use a purely functional language to efficiently implement fully persistent aggregates.[1] Finally, the technique has amortized constant overhead per operation. It remains open whether the technique can be improved to worst–case constant overhead per operation, which makes it applicable to real–time applications.

We devote the next two chapters to the implementations of full persistent arrays and functional real–time deques, where the technique of Driscoll, Sarnak, Sleator and Tarjan [23] does not apply. Efficient implementation of fully persistent arrays has been an active research topic in the functional programming language community. We show in Chapter 5

[1] For an advocate of purely functional programming, it is always an important, if not obsessive, issue to do in a purely functional language whatever can be done in an imperative language, and do it with the same efficiency.

4.3 Efficient Implementations of Persistent Aggregates

The approaches described in the above two sections — using a compile time analysis or run time reference counting to detect safe in–place updates, or using language restrictions or programming styles to enforce single–threaded accesses to aggregates — have a common drawback. They are unable to deal with programs which mostly access aggregates in a multi–threaded manner. In those programs, old versions of an aggregate must be kept around for possible future accesses, and few destructive updates can be performed. How would the program in Figure 4.1 be implemented by the approaches described by the above two sections, for example? Both of the above two approaches fail because array A still has to be duplicated to produce a correct implementation.

A general approach to solve the problems caused by multi–threaded accesses to an aggregate is to make the aggregate persistent such that, during a series of modifications, the old versions, as well as the newest version, of the aggregate can still be accessed. Copying the whole aggregate upon an update operation is one way to make the aggregate persistent, but is too expensive. We are more interested in efficient implementations of persistent aggregates.

Using the terminology of Driscoll, Sarnak, Sleator and Tarjan [23], ordinary data structures are *ephemeral* in the sense that making a change to the structure destroys the old version, leaving only the new one. A data structure is *partially persistent* if all versions can be read but only the newest version can be updated. A data structure is *fully persistent* if every version can be both read and updated. Sometimes, we simply call fully persistent aggregates *functional* because accessing such an aggregate will not have any effect on the values of its older versions.

The techniques developed by Driscoll, Sarnak, Sleator and Tarjan [23] use a side–effecting programming language to make fully persistent any linked data structure whose nodes all have bounded in–degree. (The in–degree of a node is the number of pointers pointing to it.) An operation on the ephemeral aggregate can be implemented on the fully persistent aggregate by an additional constant overhead. The constant overhead per operation is an amortized one in the following sense: an individual operation to the fully persistent aggregate may incur large overhead, but dividing the total cost of any worst–case sequence of operations by the number of the operations in the sequence always results in constant overhead per operation.

is initialized, it becomes read–only and will not admit any additional modification, a feature called *array comprehension* is provided in Haskell for constructing and initializing an entire array all at once, and, during the initialization, hiding all the intermediate versions of the array from the programmer. Since the programmer has no access to those intermediate versions of the array, the array comprehension primitive can be implemented in such a way that all updates within the primitive are done destructively.

Another approach is taken in the dataflow language Id [4]. In Id, multiple initialization to an array entry is not allowed, and the execution of a read operation is always delayed until the designated entry in the array is initialized. These restrictions rule out the read–write and write–write race conditions; hence guarantee deterministic results for all array operations. These language features for efficient construction of functional arrays, interestingly enough, are reported at the same time by Arvind, Nikhil, and Pingali [3], Hudak [36], and Wadler [74] at a workshop on graph reduction [25].

Recently, an imperative functional programming style has been advocated by Wadler [76,78] and by Peyton Jones and Wadler [47]. In this programming styles, a set of functions is provided to manipulate aggregates, and these functions are the only mechanisms for accessing the aggregate. Each of the provided function is either single–threaded, or is given a single–threaded implementation as its definition. Furthermore, if there are two unrelated sequences of single–threaded accesses to an identical aggregate, a sequencing mechanism is used to combine them such that the execution order of the two sequences is always made clear. The sequencing primitive then passes the aggregate side–effected by the first thread to the second thread. For example, the program in Figure 4.1 contains two independent threads: one thread computes the left operand and the other computes the right operand. Such a program would not be allowed in the imperative functional programming style because the two threads are not sequentialized. The program can be rewritten in the imperative functional style such that the left thread is executed first, and the array side–effected by the update operation of the left thread is passed to the right thread. This new program will yield result different from one produced by the original in Figure 4.1. However, this programming style always gives a deterministic result for any sequence of accesses to an aggregate (which is always updated in place), and it retains a mostly functional programming style at the same time.

accesses is part of its definition. The challenge is then to incorporate the concept of single–threadedness but at the same time without losing the referential transparency property of the functional language. If two code fragments have the same semantics, then replacing one with the other in any context shall not change the result of the program, even if one code fragment is specified (and will be implemented) in a single–threaded manner but the other is not.

This approach has been taken, for example, in the works of Guzmán and Hudak [32], Wadler [77], and Odersky [58,59], in which they discuss the designs of single–threaded functional languages. A common characteristic of their designs is that the usage of a bound variable (such as whether it is read only, or used only once, or single–threaded, *etc.*) is specified as a part of its type. A set of typing rules is then devised to check whether all the usages of each variable in a program are consistent with the type of the variable. Because we want to know whether a program is well–typed or not before its execution, the type checking or type reconstruction algorithm must terminate for all programs at compile time. Of the programs accepted by the language's type scheme, variables of the single–threaded type can be implemented accordingly at run time. On the other hand, programs rejected by the type scheme will not compile.

There is one problem though. These single–threadedness type schemes are incomplete in the sense that there always exists some programs that can be implemented single–threadedly but are rejected by the compiler according to the typing rules. This is analogous to ML type inference, which will not necessarily accept every program which does not result in a run time type error. Furthermore, the proposed single–threadedness type schemes can be quite complicated, and, without a good understanding of its working, a programmer can be puzzled by why a program is accepted by the compiler but a small modification to the same program results in a rejection from the compiler.

A less restrictive approach is to make available a set of single–threaded primitives in a functional language, such that expressions composed only of these primitives are always single–threaded and can be implemented accordingly. Single–threadedness will not be enforced globally by a type mechanism, but is introduced locally by this set of primitive functions. No compile time analysis is necessary.

Such a language feature can be found in the functional language Haskell [38], where it is used to create monolithic arrays. Because an array is often used in such a way that, once it

because no overhead is spent in maintaining multiple versions of an aggregate data structure. The disadvantage is that the analysis is incomplete, as are all interesting semantic analyses, in the sense that there may exist single–threaded accesses that are not detected by the analyses. In addition, such analyses are expensive (they have exponential time complexity with respect to program size) and not universal (they currently apply to first–order languages only).

The task of detecting single–threaded accesses can also be performed at run time, instead of at compile time. We can attach to each aggregate data structure a field which records the number of pointers currently have access to it, that is, a reference count. Prior to an update operation, if the reference count is one, which means that the update operation has exclusive access to the aggregate, then we can perform the update in–place because there will be no access to the old aggregate afterward. In doing so, we trade the expensive compile time analysis for the run time overhead of maintaining reference counts. The overhead can be reduced if the reference count can be implemented by the underlining machine hardware such that updates to a multi–threaded aggregate data structure (which has reference count larger than one) will invoke a "trap" routine to perform the duplication of the aggregate and the modification of the new copy. Updates to a single–threaded aggregate (which has reference count of one) are always performed in–place and never invoke the trapping routine. The reference counts can also be used to aid garbage collection.

4.2 Language Restrictions or Programming Styles

In the above section, we rely a compile time analysis or a run time reference counting to detect places in a functional program where update operations can be performed in place. That means that a programmer usually has no control over how (efficiently) an aggregate data structure will be implemented. It is subject to the implementation of a particular compile time analysis or a particular reference counting scheme. Nevertheless, sometimes a programmer has a clear idea of how an aggregate will be used and knows which parts of the program are single–threaded. The compile time analysis may fail to detect these parts, and the run time checking may spend time in maintaining reference counts that are never necessary. Why not let the programmers have a say in how the aggregates will be used?

A functional language can be so designed such that specification of single–threaded

$$((Read\ (Update\ A\ i\ u)\ m) + (Read\ A\ j))\ *\qquad (4.1)$$
$$((Read\ (Update\ A\ k\ v)\ n) + (Read\ A\ l))$$
$$\quad where\quad A = \ldots\text{definition for array } A\ldots$$
$$\qquad\quad i = \ldots\text{definition for index } i\ldots$$
$$\qquad\quad \ldots$$

$$(f\ A\ i\ u\ m\ j) * (f\ A\ k\ v\ n\ l)\qquad (4.2)$$
$$where\quad f\ A\ w\ x\ y\ z = (Read\ (Update\ A\ w\ x)\ y) + (Read\ A\ z),$$
$$\qquad A = \ldots\text{definition for array } A\ldots$$
$$\qquad i = \ldots\text{definition for index } i\ldots$$
$$\qquad \ldots$$

Figure 4.3: Two functional programs which illustrate that single–threadedness is a global property of a program, not a local one.

program (4.1), accesses to array A in each of the two operands of the multiplication can be made single–threaded. However, accesses to array A are not single–threaded in the whole expression. We cannot use just one copy of the array A to implement program (4.1). Furthermore, program (4.1) can be unfolded from program (4.2). In program (4.2), although we can infer that array A in the body of function f can be made single–threaded in an isolated context, we cannot implement function f single–threadedly in all of its occurrences in the whole program. We need an interprocedural analysis in order to do that.

This approach — compile time analysis to detect single–threaded accesses to aggregate data structures and generate code for in–place updates whenever it is safe to do so — has been reported in the works of Schwartz [67,68], Hudak and Bloss [37], Gopinath and Hennessy [29], and Bloss [10,11]. Hudak and Young [41] propose a collecting interpretation of functional programs, which can be used as a basis for interprocedural analysis of functional programs.

The advantage of this approach is that it is possible to generate very efficient code

$$(Read\ (Update\ A\ i\ u)\ m) + (Read\ A\ j)$$
$$where \quad A = \ldots \text{definition for array } A \ldots$$
$$i = \ldots \text{definition for index } i \ldots$$
$$\ldots$$

Figure 4.2: A purely functional program that can be implemented single–threadedly.

results of accessing the original aggregate. Given a specification for a functional program, there may be many ways to implement the program. Thus, one can choose a program in which aggregates are accessed single–threadedly. This is likely to result in a more efficient implementation because the update operations can be performed in–place.

Consider the functional program in Figure 4.2. Because of the commutativity of addition, we can either first compute the left operand $(Read\ (Update\ A\ i\ u)\ m)$ then compute the right operand $(Read\ A\ j)$; or we can do it the other way around. Both will yield the correct result. If we compute the right operand first, then the update operation in the left operand can be performed destructively because the only remaining array access will read the newly modified array. However, if we compute the left operand first, then we cannot perform the update destructively because the read operation in the right operand will demand the old array which will be gone.

A non–standard semantics can be defined to capture the single–threadedness property of bound variables in a functional program. An analysis based on this non–standard semantics can then be applied to approximate the single–threadedness of the program's bound variables, and to generate code for in–place modification for update operations to single–threaded variables. Because the denotational semantics of a functional program often underspecifies the evaluation order of an expression, as exemplified by the program in Figure 4.2, ideally the analysis should also choose one evaluation ordering, among all the correct ones, that exhibits most single–threadedness. Code is then generated based on this most single–threaded ordering.

The analysis can be further complicated by the fact that single–threadedness is a global property, not a local one. Consider, for example, the two programs in Figure 4.3. For

One approach to solving the aggregate update problem is to exploit a program's execution behavior such that updates to aggregate data structures are done destructively whenever it is safe to do so. This approach, which is based on a notion of *single–thread* access to aggregates, is described in section 4.1. We can also enforce a single–threading programming style by means of language design, such that all updates in a program can be done destructively without losing the value semantics. This approach is summarized in section 4.2. A general approach to the aggregate update problem is to design efficient run time data structures such that multiple versions of an aggregate value can be easily accessed. We outline this approach in section 4.3, and devote the next two chapters to efficient implementations of purely functional arrays and real–time deques.

4.1 Safe In–Place Modifications

Let us use the term *access* to include both read and update operations to an aggregate data structure. Schmidt [65] defines a notion of *single–threaded* accesses when implementing a simple imperative language based on its denotational semantics. A sequence of accesses to an aggregate data structure is single–threaded if each access in the sequence refers only to the newest version of the aggregate. That is, each operation accesses the aggregate resulting from the previous operation. If a sequence of access is not single–threaded, it is *multi–threaded*. An aggregate is single–threaded if all accesses to the aggregate during the entire program execution is single–threaded. By the very definition of single–threadedness, an update operation to a single–threaded aggregate can be perform destructively because the previous version of the aggregate will never be used afterward.

Single–threadedness is a notion concerning implementation, and often is not described by a language's denotational semantics. In an imperative program like C, the program store (which is a finite mapping of variable names to their associated values) is understood to be single–threaded. However, the denotational semantics of an imperative language usually does not specify that all accesses to a program's store are single–threaded. That is the reason why Schmidt proposes the notion of single–threadedness to get an efficient implementation of an imperative language, based on its denotational semantics.

In a functional language, because of its value semantics, accesses to aggregate data structures are multi–threaded in nature. A modification to an aggregate must not alter the

$$(Read\ (Update\ A\ i\ u)\ m) + (Read\ (Update\ A\ j\ v)\ n)$$
$$where \quad A = \ldots \text{definition for array } A \ldots$$
$$i = \ldots \text{definition for index } i \ldots$$
$$\ldots$$

Figure 4.1: A purely functional program with two independent threads updating the same array.

time each in the RAM model.

How would a functional program implement arrays on a RAM machine? Consider the functional program in Figure 4.1. In the program, *Update* A i u returns an array identical to A except in the ith entry which now stores value u, and *Read* (*Update* A i u) m retrieves the mth entry of the newly modified array. After the array A is defined in the *where* clause, it is updated in two separate subexpressions in the program. The array is then read once in each of the two subexpressions in the program. If (*Update* A i u) in the first subexpression were implemented by a destructive modification to the array bound to variable A, then, after computing this subexpression, variable A would be bound to an array different from the original definition in the *where* clause; so would be the variable A in the subexpression (*Update* A j v). Value semantics of the program is no longer observed, and referential transparency is lost.

We can solve this problem by making a copy of the whole array whenever it is updated, modifying the new copy, and returning the modified copy as the result. This way, the value semantics of the program is enforced, and referential transparency is retained. However, making a new copy whenever updating an entry of an array is generally considered too expensive in most cases.

This problem is not limited to array updates, but applies to many situations where modifications to aggregate data structures are needed. It is usually called the *aggregate update problem*. Depending on different perspectives to this problem, there have been various approaches to solving it. We classify in this chapter three popular approaches to this problem, and compare their advantages and disadvantages.

Chapter 4

Aggregate Data Structures in Functional Programs

A functional program computes a value according to the function definitions that constitute the program. A function can compute a value of a simple data type, such as an integer, a character, or a boolean value, for example. Functions can be defined to compute values of larger structures as well, like pairs of integers, lists of real numbers, character–to–integer functions, or double–ended queues of some events. When implementing a functional program, it is important that the values it computes can be represented and manipulated efficiently by the underlining machine. This efficiency consideration is shared among all programming languages, and is not unique to functional languages. What makes a purely functional language different from other languages is that its value semantics also restricts it from changing the binding of a variable to its associated value when computing other values. This may cause problems when computing aggregate data structures, where a configuration of an aggregate data structure is computed most easily by a destructive modification to some existing configuration, which may already be bound to other variables.

Take the array data structure as an example. An array is a mapping from a finite index set to a set of data whose sizes are equal. It can be implemented in a Random Access Machine (RAM) by laying out the array in a successive blocks of memory. Reading a given entry in the array is accomplished by reading the contents of the corresponding memory location, and an update to an entry of the array is implemented by storing the new value, thus destroying the old value, in the corresponding memory location. Both cost constant

Jensen [45,46]. Their work also provides a junction between semantics and logics for functional programming languages. Their work is mostly concerned with the dual relationship between domain theory and its axiomatic logics; ours is concerned with least fixed points on finite domains and their corresponding calculus. While their work usually provides a decidable theory without giving an explicit proof strategy, the augmented, simply typed λ–calculus in our approach provides a simple way to compute the desired results.

$$cat^{\#}\ xss^{\#} \;=\; foldr^{\#}\ append^{\#}\ xss^{\#}\ 3_4 \tag{3.5}$$

$$cat^{\#} = \lambda x_6 . \lambda y_3 . \ \bigsqcup\{\ \bigsqcap\{x\ 4_5,\ y\ 0_2\},\ \bigsqcap\{x\ 1_5,\ y\ 1_2\},\ x\ 0_5\} \tag{3.6}$$

Figure 3.5: Strictness analysis for function `cat`.
Note. Equation (3.5) describe the strictness property of `cat`, according to its program text. The strictness of `cat` is then calculated by first substituting the definition of $append^{\#}$ (in Figure 3.3), $foldr^{\#}$ (in Figure 3.4), and 3_4 (in Table 3.4) into Equation (3.5), then by normalizing the resulting term. The result is Equation (3.6), which is the strictness property of function `cat`.

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

Table 3.5: The results of applying $cat^{\#}$ to all the elements in the abstract domain 6.
Note. The results are obtained by calculating the normal forms of $cat^{\#}\ xss^{\#}$, where the definition of $cat^{\#}$ is from Equation (3.6) in Figure 3.5, and the definition of $0_6, 1_6, \ldots, 5_6$ is from Table 3.4.
The results in the above table are interpreted in the following way: If the argument to the `cat` function is an undefined list ($xss^{\#} = 0_6$), then the result is an undefined list. If the argument is a list with an undefined tail, or is a finite list with one of its element either being an undefined list or being a list with an undefined tail ($xss^{\#} = 1_6, 2_6$, or 3_6), then the result is (at most) a list with an undefined tail. If the argument is a finite list in which all elements are finite lists but one of them contains an undefined element ($xss^{\#} = 4_6$), then the result is (at most) a finite list with one of its elements undefined. If the argument is a finite list in which all elements are finite lists with all elements defined ($xss^{\#} = 5_6$), then the result is (at most) a finite list with all its elements defined.

$$
\begin{aligned}
foldr^{\#} = \lambda\, f_{4\to 4\to 4}\,.\,\lambda\, x_6\,.\,\lambda\, y_4\,.\,\lambda\, z_3\,.\\
\bigsqcup\{\ & \textstyle\prod\{x\,4_5,\ z\,0_2,\ f\,3_4\,0_4\,2_3\},\\
& \textstyle\prod\{x\,4_5,\ z\,1_2,\ f\,3_4\,1_4\,1_3,\ f\,3_4\,0_4\,2_3\},\\
& \textstyle\prod\{x\,4_5,\ f\,3_4\,2_4\,0_3,\ f\,3_4\,1_4\,1_3,\ f\,3_4\,0_4\,2_3\},\\
& \textstyle\prod\{x\,3_5,\ f\,0_4\,2_4\,2_3,\ z\,0_2,\ y\,1_3\},\\
& \textstyle\prod\{x\,3_5,\ f\,0_4\,3_4\,2_3,\ z\,0_2,\ f\,3_4\,2_4\,0_3,\ y\,1_3\},\\
& \textstyle\prod\{x\,3_5,\ f\,0_4\,3_4\,1_3,\ z\,1_2,\ y\,0_3\},\\
& \textstyle\prod\{x\,3_5,\ f\,0_4\,3_4\,2_3,\ z\,0_2,\ y\,0_3\},\\
& \textstyle\prod\{x\,3_5,\ z\,0_2,\ f\,0_4\,1_4\,2_3,\ y\,2_3\},\\
& \textstyle\prod\{x\,3_5,\ z\,1_2,\ f\,0_4\,2_4\,2_3,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \textstyle\prod\{x\,3_5,\ z\,1_2,\ f\,3_4\,1_4\,1_3,\ f\,0_4\,3_4\,2_3,\ y\,0_3\},\\
& \textstyle\prod\{x\,3_5,\ z\,1_2,\ f\,0_4\,2_4\,1_3,\ y\,1_3\},\\
& \textstyle\prod\{x\,3_5,\ f\,0_4\,3_4\,2_3,\ f\,3_4\,2_4\,0_3,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \textstyle\prod\{x\,3_5,\ f\,0_4\,3_4\,1_3,\ f\,3_4\,2_4\,0_3,\ y\,1_3\},\\
& \textstyle\prod\{x\,3_5,\ f\,0_4\,3_4\,0_3,\ y\,0_3\},\\
& \textstyle\prod\{x\,2_5,\ f\,1_4\,2_4\,2_3,\ z\,0_2,\ y\,1_3\},\\
& \textstyle\prod\{x\,2_5,\ f\,1_4\,3_4\,2_3,\ z\,0_2,\ f\,3_4\,2_4\,0_3,\ y\,1_3\},\\
& \textstyle\prod\{x\,2_5,\ f\,1_4\,3_4\,1_3,\ z\,1_2,\ y\,0_3\},\\
& \textstyle\prod\{x\,2_5,\ f\,1_4\,3_4\,2_3,\ z\,0_2,\ y\,0_3\},\\
& \textstyle\prod\{x\,2_5,\ z\,0_2,\ f\,1_4\,1_4\,2_3,\ y\,2_3\},\\
& \textstyle\prod\{x\,2_5,\ z\,1_2,\ f\,1_4\,2_4\,2_3,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \textstyle\prod\{x\,2_5,\ z\,1_2,\ f\,3_4\,1_4\,1_3,\ f\,1_4\,3_4\,2_3,\ y\,0_3\},\\
& \textstyle\prod\{x\,2_5,\ z\,1_2,\ f\,1_4\,2_4\,1_3,\ y\,1_3\},\\
& \textstyle\prod\{x\,2_5,\ f\,1_4\,3_4\,2_3,\ f\,3_4\,2_4\,0_3,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \textstyle\prod\{x\,2_5,\ f\,1_4\,3_4\,1_3,\ f\,3_4\,2_4\,0_3,\ y\,1_3\},\\
& \textstyle\prod\{x\,2_5,\ f\,1_4\,3_4\,0_3,\ y\,0_3\},\\
& \textstyle\prod\{x\,1_5,\ f\,2_4\,2_4\,2_3,\ z\,0_2,\ y\,1_3\},\\
& \textstyle\prod\{x\,1_5,\ f\,2_4\,3_4\,2_3,\ z\,0_2,\ f\,3_4\,2_4\,0_3,\ y\,1_3\},\\
& \textstyle\prod\{x\,1_5,\ f\,2_4\,3_4\,1_3,\ z\,1_2,\ y\,0_3\},\\
& \textstyle\prod\{x\,1_5,\ f\,2_4\,3_4\,2_3,\ z\,0_2,\ y\,0_3\},\\
& \textstyle\prod\{x\,1_5,\ z\,0_2,\ f\,2_4\,1_4\,2_3,\ y\,2_3\},\\
& \textstyle\prod\{x\,1_5,\ z\,1_2,\ f\,2_4\,2_4\,2_3,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \textstyle\prod\{x\,1_5,\ z\,1_2,\ f\,3_4\,1_4\,1_3,\ f\,2_4\,3_4\,2_3,\ y\,0_3\},\\
& \textstyle\prod\{x\,1_5,\ z\,1_2,\ f\,2_4\,2_4\,1_3,\ y\,1_3\},\\
& \textstyle\prod\{x\,1_5,\ f\,2_4\,3_4\,2_3,\ f\,3_4\,2_4\,0_3,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \textstyle\prod\{x\,1_5,\ f\,2_4\,3_4\,1_3,\ f\,3_4\,2_4\,0_3,\ y\,1_3\},\\
& \textstyle\prod\{x\,1_5,\ f\,2_4\,3_4\,0_3,\ y\,0_3\},\\
& \textstyle\prod\{x\,0_5,\ z\,1_2,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \textstyle\prod\{x\,0_5,\ f\,3_4\,2_4\,0_3,\ f\,3_4\,1_4\,1_3,\ y\,2_3\},\\
& \textstyle\prod\{x\,0_5,\ f\,3_4\,2_4\,0_3,\ y\,1_3\},\\
& \textstyle\prod\{x\,0_5,\ y\,0_3\},\\
& \textstyle\prod\{x\,0_5,\ y\,1_3,\ z\,1_2\},\\
& \textstyle\prod\{x\,0_5,\ y\,2_3,\ z\,0_2\}\}
\end{aligned}
$$

Figure 3.4: The strictness property of function `foldr`, described in language Λ^0.

by a simple translation from Equation (3.2) by substituting $cons^{\#}$ by its definition from Table 3.2, expanding the *if* statements (*i.e.*, the *step* functions in Proposition 3.3.40), and normalization. The least fixed point of this Λ^1 term can be calculated by an approximation sequence starting from $\lambda x_4 . \lambda y_4 . \lambda z_3 . 0_2$, the least defined term in language $\Lambda^1_{4\to 4\to 4}$. The resulting least fixed point will be a Λ^1 term. Equation (3.4) in Figure 3.3 is its translation in the Λ^0 language. It is the strictness property of the `append` function. We need not translate the final result into a Λ^0 term, but we do here it for completeness reason.

The same process can apply to the `foldr` function, resulting in the Λ^0 term in Figure 3.4 as its strictness property. Substituting the definition of $append^{\#}$ and $foldr^{\#}$ in the definition of $cat^{\#}$ and normalizing the result (Equation (3.5) in Figure 3.5) yields the strictness property of `cat` (Equation (3.6) in Figure 3.5). To see how accurate $cat^{\#}$ is, we can apply it to terms $0_6, 1_6, 2_6, 3_6, 4_6$, and 5_6, resulting Table 3.5. The result is as good as we can hope for, showing that Wadler's abstraction mechanism for list domains is quite accurate for this particular example.

All these results are calculated by a bare–bones Standard ML program which is used to normalize a given Λ^1 term, and to evaluate the least fixed point of a given Λ^1 term by a syntactic approximation iteration. The strictness property of program `foldr`, as described in Figure 3.4, takes about 20 minutes to calculate when running the least fixed point finding program under SML of New Jersey (version 0.66) on a Sun 4/290 with 32 MB physical memory. We also wrote a bare–bones SML program to calculate, using the frontier method this time, the strictness property of `foldr`. We are unable to get the least fixed point after running the frontier method, under the same machine configuration described above, for 16 hours.

3.6 Conclusion and Comparison to Other Work

We have shown how to develop a syntactic approach for least fixed point finding over finite domains. The syntactic method is sound and complete with with respect to the semantics of least fixed point computation on finite domains, and bears close relationship to the simply typed λ–calculus.

It is interesting to compare the development here with the work of Abramsky [2] and

$$
\begin{array}{lll}
\multicolumn{3}{l}{append^{\#}_{4\to 4\to 4}\ x^{\#}_{4}\ y^{\#}_{4} =} \\
 & \mathit{if} & 3_4 \sqsubseteq x^{\#} \\
 & \mathit{then} & \bigsqcup\{y^{\#},\ \ cons^{\#}\ 1_2\ (append^{\#}\ 3_4\ y^{\#})\} \\
\mathit{else} & \mathit{if} & 2_4 \sqsubseteq x^{\#} \\
 & \mathit{then} & \bigsqcup\{cons^{\#}\ 0_2\ (append^{\#}\ 3_4\ y^{\#}),\ \ cons^{\#}\ 1_2\ (append^{\#}\ 2_4\ y^{\#})\} \\
\mathit{else} & \mathit{if} & 1_4 \sqsubseteq x^{\#} \\
 & \mathit{then} & cons^{\#}\ 1_2\ (append^{\#}\ 1_4\ y^{\#}) \\
\mathit{else} & & 0_4
\end{array}
\tag{3.2}
$$

$$
\begin{array}{l}
append^{\#} = fix\ (\lambda\, f_{4\to 4\to 4}\,.\,\lambda\, x_4\,.\,\lambda\, y_4\,.\,\lambda\, z_3\,. \\
\quad \bigsqcup\{\ \ \bigsqcap\{x\,2_3,\ f\,1_4\,y\,0_3\}, \\
\qquad\ \ \bigsqcap\{x\,2_3,\ f\,1_4\,y\,1_3,\ z\,1_2\}, \\
\qquad\ \ \bigsqcap\{x\,2_3,\ z\,0_2\}, \\
\qquad\ \ \bigsqcap\{x\,1_3,\ f\,3_4\,y\,1_3,\ z\,1_2\}, \\
\qquad\ \ \bigsqcap\{x\,1_3,\ f\,2_4\,y\,0_3\}, \\
\qquad\ \ \bigsqcap\{x\,0_3,\ f\,3_4\,y\,0_3\}, \\
\qquad\ \ \bigsqcap\{x\,0_3,\ y\,z\}\})
\end{array}
\tag{3.3}
$$

$$
\begin{array}{l}
append^{\#} = \lambda\, x_4\,.\,\lambda\, y_4\,.\,\lambda\, z_3\,. \\
\quad \bigsqcup\{\ \ \bigsqcap\{x\,2_3,\ z\,0_2\}, \\
\qquad\ \ \bigsqcap\{x\,1_3,\ y\,1_3,\ z\,1_2\}, \\
\qquad\ \ \bigsqcap\{x\,0_3,\ y\,0_3\}\}
\end{array}
\tag{3.4}
$$

Figure 3.3: Strictness analysis for function `append`.

Note. Equation (3.2) describes the strictness property of `append`, according to its program text, in which Wadler's abstraction mechanism is used. Equation (3.3) expresses `append`'s strictness property as the least fixed point of a Λ^1 term. The least fixed point is then calculated by an approximation sequence by using the syntactic calculus starting from term $\lambda\, x_4\,.\,\lambda\, y_4\,.\,\lambda\, z_3\,.\,0_2$, the least element in domain $4 \to 4 \to 4$. The result is then translated into a Λ^0 term in Equation (3.4).

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

(a)

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

(b)

Table 3.4: The definitions in language Λ^0 of some semantic elements over the finite domains $2, 3, 4, 5$, and 6.

Note. Part (a) shows the definitions of the elements in domains $2, 3, 4, 5$, and 6 as terms in language Λ^0. Note that the languages and the terms are defined inductively on the following types: $3 \equiv 2 \to 2$, $4 \equiv 3 \to 2$, $5 \equiv 4 \to 2$, and $6 \equiv 5 \to 2$. All the constants in the languages are subscripted by their types to avoid possible confusion. If written in full, for example, the definition $3_6 \equiv \lambda\, x_5\, .\, x\, 2_4 \equiv \lambda\, x_{((2\to 2)\to 2)\to 2}\, .\, x\, (\lambda\, y_{2\to 2}\, .\, y\, 1_2)$.

The strictness properties of functions `hd`, `tl`, and `cons` are described in (b). Note that a primitive function has different strictness properties over different abstract domains. Here 4 is the abstract domain for lists, and 6 is the abstract domain for lists of lists.

Note that not only is domain $\mathit{4}$ a distributive lattice, but it is also isomorphic to domain $(\mathit{2} \to \mathit{2}) \to \mathit{2}$. This means that we can express the four elements in domain $\mathit{4}$ as the following terms in language $\Lambda^0_{(\mathit{2}\to\mathit{2})\to\mathit{2}}$:

$$\begin{aligned} 0_{\mathit{4}} &\equiv \lambda\, x_{\mathit{2}\to\mathit{2}}\,.\, 0_{\mathit{2}}, \\ 1_{\mathit{4}} &\equiv \lambda\, x_{\mathit{2}\to\mathit{2}}\,.\, x\; 0_{\mathit{2}}, \\ 2_{\mathit{4}} &\equiv \lambda\, x_{\mathit{2}\to\mathit{2}}\,.\, x\; 1_{\mathit{2}}, \\ 3_{\mathit{4}} &\equiv \lambda\, x_{\mathit{2}\to\mathit{2}}\,.\, 1_{\mathit{2}}. \end{aligned}$$

Furthermore, recall that, by completeness (Proposition 3.3.42), the syntactically weaker relationship $\preceq$ in language $\Lambda^0_{(\mathit{2}\to\mathit{2})\to\mathit{2}}$ captures exactly the semantically weaker relationship $\sqsubseteq$ in domain $\mathit{4}$. The strictness properties of the primitive functions `hd`, `tl`, and `cons` can be defined in language Λ^0 too. The same property also applies to domain $\mathit{6}$, which can be embedded in domain $(((\mathit{2} \to \mathit{2}) \to \mathit{2}) \to \mathit{2}) \to \mathit{2}$. The definitions of these semantic elements in language Λ^0 are summarized in Table 3.4.

Given the definitions in Table 3.4, we then can compute the strictness properties of functions `append`, `foldr`, and `cat`. Wadler's technique takes pattern–matching on input arguments into account when analyzing a functional program. Take function `append` as an example. Suppose that the first argument to `append` is only as defined as 2_{4} (*i.e.*, $2_{\mathit{4}} \sqsubseteq (x :: xs)^{\#}$ but $3_{\mathit{4}} \not\sqsubseteq (x :: xs)^{\#}$). What will be the strictness properties for its head and tail? By consulting the valuation table for $\mathit{cons}^{\#}$ in Table 3.2 for entries 2_{4}, we know that it suffices to consider either the case of $2_{\mathit{4}} = \mathit{cons}^{\#}\ 0_{\mathit{2}}\ 3_{\mathit{4}}$, or the case of $2_{\mathit{4}} = \mathit{cons}^{\#}\ 1_{\mathit{2}}\ 2_{\mathit{4}}$. (The case for $2_{\mathit{4}} = \mathit{cons}^{\#}\ 0_{\mathit{2}}\ 2_{\mathit{4}}$ is covered by both of the above two cases.) Therefore, the analysis must take both cases into consideration. That is, the final result should be the least upper bound of the two cases. (See the if–clause with condition $2_{\mathit{4}} \sqsubseteq x^{\#}$ in Equation (3.2) in Figure 3.3.) Note that this yields more information than simply evaluating $\mathit{hd}^{\#}\ 2_{\mathit{4}} = 1_{\mathit{2}}$ and $\mathit{tl}^{\#}\ 2_{\mathit{4}} = 3_{\mathit{4}}$ to get the strictness of its head and tail, which tells us nothing at all in this particular case. The strictness of *append* can then be expressed as Equation (3.2) in Figure 3.3.

If a semantic method is to be used, then the least fixed point semantics for $\mathit{append}^{\#}$ can now be evaluated by an approximation iteration starting with the least element in domain $\mathit{4} \to \mathit{4} \to \mathit{4}$. Since we are interested in the syntactic method, instead we write $\mathit{append}^{\#}$ as the least fixed point of a Λ^1 term, as Equation (3.3) in Figure 3.3. The Λ^1 terms is obtained

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

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

cat xss  =  foldr append xss []
```

Figure 3.2: The definitions of functions `append`, `foldr`, and `cat`.

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

Table 3.2: The abstraction of the primitive functions `hd`, `tl`, and `cons` over domain *4*.

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

Table 3.3: The abstraction of the primitive functions `hd`, `tl`, and `cons` over domain *6*.

Abramsky define an abstraction of a domain as a finite lattice of its non–empty Scott–closed subsets, where the Scott–closed subset are ordered by subset inclusion in the lattice. For strictness analysis on a basic domain D, D is usually abstracted to the abstract domain with only two Scott–closed subsets: $\{\bot_D\}$ and D. This abstract domain is usually written as $\mathcal{2}$. Its two elements are defined as $0_{\mathcal{2}} = \{\bot_D\}$ and $1_{\mathcal{2}} = D$, with the ordering $0_{\mathcal{2}} \sqsubseteq 1_{\mathcal{2}}$. If an element $d \in D$ has abstraction $d^{\#} = 0_{\mathcal{2}}$, we know for sure that d is the undefined element in domain D. If $d^{\#} = 1_{\mathcal{2}}$, then d can be any element in D. Note that $0_{\mathcal{2}}$ and $1_{\mathcal{2}}$ can be expressed as 0 and 1 in language Λ^0.

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

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

with the ordering $0_{\mathcal{4}} \sqsubseteq 1_{\mathcal{4}} \sqsubseteq 2_{\mathcal{4}} \sqsubseteq 3_{\mathcal{4}}$. The abstract versions of the three primitive functions `hd`, `tl`, and `cons` on domain list(D) are described in Table 3.2. An abstract domain $\mathcal{6}$ for domain list(list(D)) are also defined, with its abstract elements as the following:

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

and with the ordering $0_{\mathcal{6}} \sqsubseteq 1_{\mathcal{6}} \sqsubseteq 2_{\mathcal{6}} \sqsubseteq 3_{\mathcal{6}} \sqsubseteq 4_{\mathcal{6}} \sqsubseteq 5_{\mathcal{6}}$. Table 3.3 contains the abstraction of functions `hd`, `tl`, and `cons` over domain $\mathcal{6}$.

variables as arguments. This makes the translation from a functional program into a Λ^1 term easy, and the length of the resulted Λ^1 term comparable to the length of the original program. The time spent in the $\preceq$ relationship testing between two successive approximation is then comparable to the cases in the frontier method. Furthermore, an approximation sequence in the syntactic method usually reaches the least fixed point more quickly than an approximation in the frontier method, because the former utilizes the textual information from the program, while the latter searches along the semantic domains, no matter what the given program looks like. An example involving an abstract domain of at least 10^6 elements is elaborated in the next section, using the proposed syntactic method. This example was previously used to show that a naive frontier method is not effective when dealing with abstract domain of considerable size [49,50]. Using the proposed syntactic method, we are able to compute least fixed points successfully.

In the worst case, the syntactic method will, just as the frontier method, require exponential running time (with respect to the program length). On the other hand, safe approximation schemes can also be developed based on the proposed syntactic method. The approximation scheme will calculate less accurate fixed points, but do it faster. For example, a reduction rule like

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

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

3.5 Strictness Analysis over Non–Flat Domains: An Example

In this section, we use the proposed syntactic method to solve a substantial problem. The problem is to derive the strictness property of a functional program which concatenates a list of lists into a single list. This function, named `cat`, is composed from a right–associative `foldr` function and the usual `append` function. The whole program text is in Figure 3.2.

The abstraction techniques for strictness analysis from Burn, Hankin, and Abramsky [15] and Wadler [75] are used to solve this problem. Recall that Burn, Hankin, and

domains based on domain $D_{\mathcal{3}} = \{\bot, \|, \top\}$, with the ordering $\bot \sqsubseteq_{\mathcal{3}} \| \sqsubseteq_{\mathcal{3}} \top$, and define abstract semantics of programs accordingly. Are domains $D_\sigma, \sigma \in \Gamma$, along with language Λ, helpful in such situations?

Figure 3.1 provides a good hint. We can embed domain $D_{\mathcal{3}}$ into domain $D_{\mathcal{2}\to\mathcal{2}}$ by the encoding $\bot = [0 \mapsto 0, 1 \mapsto 0]$, $\| = [0 \mapsto 0, 1 \mapsto 1]$, and $\top = [0 \mapsto 1, 1 \mapsto 1]$, and define the three elements of $D_{\mathcal{3}}$ in language $\Lambda^0_{\mathcal{2}\to\mathcal{2}}$ accordingly as $\bot = [\![\lambda\, x\, .\, 0]\!]$, $\| = [\![\lambda\, x\, .\, x]\!]$, and $\top = [\![\lambda\, x\, .\, 1]\!]$. Least fixed point computations on the domains of function spaces generated by $D_{\mathcal{3}}$ can then be carried out in Λ^0 as usual. As another example, we can also embed the boolean domain $\mathit{bool} = \{\bot, t, f\}$, with the ordering $\bot \sqsubseteq t$ and $\bot \sqsubseteq f$ but neither $f \sqsubseteq t$ nor $t \sqsubseteq f$, in domain $D_{\mathcal{2}\to\mathcal{2}\to\mathcal{2}}$ by $\bot = [\![\lambda\, x\, .\, \lambda\, y\, .\, x \sqcap y]\!]$, $t = [\![\lambda\, x\, .\, \lambda\, y\, .\, x]\!]$, and $f = [\![\lambda\, x\, .\, \lambda\, y\, .\, y]\!]$. However, since domains $D_\sigma, \sigma \in \Gamma$, are always distributive, only distributive finite domains can be properly embedded in D_σ.

We might wish to perform least fixed point computations directly on domain $D_{\mathcal{3}}$, and the function spaces generated by it, using a new language without the explicit encoding of the three elements in language $\Lambda^0_{\mathcal{2}\to\mathcal{2}}$. For example, we might want to define a new language based only on the three constants $\bot, \|, \top$ and the two operations $\sqcup$ and $\sqcap$ between them (without incorporating $0, 1$ and their $\sqcup$ and $\sqcap$ operations). Moreover, the inference rules for syntactically weaker and the reduction rules for normalization are similarly adapted to perform computations on this new language. However, it turns out that this is not a vital approach, at least if the syntax and reduction rules on Λ are not greatly changed. As an example, Figure 3.1 shows that there are 10 elements in domain $D_{\mathcal{3}\to\mathcal{3}}$; but the language $\Lambda^0_{\mathcal{3}\to\mathcal{3}}$, based on the above idea, can only represent 6 of them: $\lambda\, x\, .\, \bot$, $\lambda\, x\, .\, \|$, $\lambda\, x\, .\, \top$, $\lambda\, x\, .\, x$, $\lambda\, x\, .\, x \sqcup \|$, and $\lambda\, x\, .\, x \sqcap \|$. The new language is not able to express the element $[\bot \mapsto \bot,\ \| \mapsto \bot,\ \top \mapsto \|]$ in domain $D_{\mathcal{3}\to\mathcal{3}}$, for example. The difficulty seems to arise from the fact that the new language cannot even define the *step* functions (See Proposition 3.3.40).

3.4.4 Complexity and Approximation

With respect to the efficiency of the proposed syntactic method, is it better than simply using the frontier method? We believe it is better, especially if we use the approximation method on language Λ^1. We observe that a functional program often has simple textual structure. That is, function applications in typical functional programs often have bound

Example 3.4.49 The term $F \equiv \lambda f . \lambda x . f\ x$ is in $\Lambda^1_{(2\to 2)\to 2\to 2}$. By using the fixed point term Y defined in Example 3.3.46, we arrive at $\lambda x . 0$ as the normal form of YF, and thus as the least fixed point of F. The approximation method also finds $\lambda x . 0$ as the least fixed point.

Clack and Peyton Jones give a rather interesting example in [19]. It is a term H in language $\Lambda_{((2\to 2)\to 2)\to(2\to 2)\to 2}$, defined as

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

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

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

and the sequence does not reach a limit under relation $\preceq$. However, if H is translated (as described in Example 3.4.47) into a semantically equivalent term H^1 in Λ^1 as

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

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

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

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

then calculate the least fixed point of H^0. But this is not really necessary. □

3.4.3 Embedding of Other Finite Domains

The finite domains $D_\sigma, \sigma \in \Gamma$, and its language Λ are very useful when, for example, computing the strictness property of functional programs over flat basic domains, because the basic abstract domain for strictness happens to be D_2. But are they flexible enough for other abstract domains based not on D_2? For example, we may like to have abstract

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

□

The only difference between Λ^1 and Λ^0 is that the bound variables of a Λ^1 term are allowed to be function arguments. In doing so, we lose the completeness property. For example, the two terms $\lambda\,f\,.\,\lambda\,x\,.\,f\ x$ and $\lambda\,f\,.\,\lambda\,x\,.\,(f\ 0) \sqcup ((f\ 1) \sqcap x)$ are in language $\Lambda^1_{(\mathcal{2}\to\mathcal{2})\to\mathcal{2}\to\mathcal{2}}$, are semantically equivalent, but cannot be shown to be syntactically equivalent under the $\simeq$ relation. However, the two methods described in Section 3.3 for computing the least fixed points still work. That is, the least fixed point term $Y \in \Lambda^0_{(\sigma\to\sigma)\to\sigma}$ works both for terms in language $\Lambda^0_{\sigma\to\sigma}$ and $\Lambda^1_{\sigma\to\sigma}$. Also, the iterative method is guaranteed to converge because, for a given type $\sigma \in \Gamma$, there are only a finite number of terms in Λ^1_σ, and for terms $M, N \in \Lambda^1$, the normal form for MN is also in Λ^1.

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

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

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

of Λ^0. We also perform $\boldsymbol{\beta} \sqcup \sqcap \boldsymbol{d}$–reduction during the translation process to make sure that the final result is a Λ^0 term.

Example 3.4.47 Suppose we have a term M in $\Lambda_{(2\to 2\to 2)\to 2\to 2\to 2}$, defined as

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

We want to translate M into a term in $\Lambda^0_{(2\to 2\to 2)\to 2\to 2\to 2}$. First, we observe that the function $apply \in D_{(2\to 2)\to 2\to 2}$, where $apply\ f\ x = f\ x$ for all $f \in D_{2\to 2}$ and all $x \in D_2$, can be defined in $\Lambda^0_{(2\to 2)\to 2\to 2}$ by

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

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

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

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

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

but has normal form

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

which is in $\Lambda^0_{(2\to 2\to 2)\to 2\to 2\to 2}$. □

3.4.2 Relaxation of Language Λ^0 to Language Λ^1

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

Definition 3.4.48 The sub–language Λ^1_σ of Λ_σ is inductively defined for each type $\sigma \in \Gamma$ as follows.

as defined in Example 3.3.41 is such a term. Furthermore, YM normalizes to $\lambda x\,.\,0$, which is the least fixed point of M.

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

3.4 Implementation Issues

In this section, we briefly consider several implementation issues when using a syntactic approach based on language Λ^0 in computing least fixed points of the abstract functions derived from functional programs. One immediate concern is that, although the abstract semantics of a functional program can be easily described in language Λ, it is not necessary so in language Λ^0. In short, we must have a way to translate a closed term in Λ into an equivalent term in Λ^0 before we can compute its least fixed point. We also describe how we can relax language Λ^0 to a new language Λ^1 to greatly reduce the translation process. Language Λ^1 is no longer complete. However, we show that the syntactic approach can be adapted to compute least fixed points on Λ^1 too. We then consider the problem of embedding other distributive finite domains into domains $D_\sigma, \sigma \in \Gamma$. We also mention an approximation technique which speeds up the least fixed point iteration but computes less accurate results.

3.4.1 Translation from Language Λ to Language Λ^0

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

However, function application itself is definable in language Λ^0, and we can use this definition to turn each atomic term $M_{i,j}$ into an equivalent term where e_k is in $\Lambda^0_{\tau_{i_k}}$. The translation proceeds until all the function applications in a term satisfy the requirements

1. The existence of Y such that $[\![Y]\!] = \mathit{fix}$ is proved by Proposition 3.3.40. By the definition of $[\![\cdot]\!]$, it follows that $[\![YF]\!] = [\![Y]\!][\![F]\!] = \mathit{fix}\ [\![F]\!]$.

2. First of all, $[\![YF]\!] = \mathit{fix}[\![F]\!] = [\![F]\!](\mathit{fix}[\![F]\!]) = [\![F]\!]([\![YF]\!]) = [\![F(YF)]\!]$. On the other hand, YF normalizes to a Λ^0 term M with $[\![YF]\!] = [\![M]\!]$ (Proposition 3.2.35 and 3.3.39). Similarly, $F(YF)$ normalizes to a Λ^0 term N with $[\![F(YF)]\!] = [\![N]\!]$. That is, $[\![M]\!] = [\![N]\!]$. It follows that $M \simeq N$ (Corollary 3.3.44), which implies $M \equiv N$ (Lemma A.0.57).

$\diamond$

The above Theorem states that not only does there exist a term Y whose denotation is the least fixed point function *fix* (clause 1 in Theorem 3.3.45), but whose *computational* characteristic also serves as a least fixed point term (clause 2 in Theorem 3.3.45).

The above properties of language Λ^0 enable us to syntactically calculate the least fixed point of a term $M \in \Lambda^0$. In fact, there are two ways to do it. The first method uses Propositions 3.3.40 and Theorem 3.3.45 to find the least fixed point term $Y \in \Lambda^0_{(\sigma\to\sigma)\to\sigma}$ for the given type $\sigma \in \Gamma$, then calculate the normal form of YM. This normal form denotes the least fixed point of $[\![M]\!]$.

The second method uses an approximation sequence starting with $B_\sigma \in \Lambda^0_\sigma$, where $[\![B_\sigma]\!] = \bot_\sigma$. The iteration successively calculates a term $N^{(k)} \in \Lambda^0_\sigma$, where $N^{(0)} = B_\sigma$ and $N^{(k+1)}$ the normal form of $MN^{(k)}$, until it finds $N^{(i+1)} \preceq N^{(i)}$. Then term $N^{(i)}$ is the least fixed point of M. The iteration is guaranteed to be terminated because, for a given type $\sigma \in \Gamma$, there are only finite number of Λ^0_σ terms. Furthermore, the iteration will terminate as soon as possible because, by completeness (Proposition 3.3.42), $[\![N^{(i+1)}]\!] \sqsubseteq [\![N^{(i)}]\!]$ implies $N^{(i+1)} \preceq N^{(i)}$.

Example 3.3.46 Suppose we want to calculate the least fixed point of the following $\Lambda^0_{(2\to 2)\to(2\to 2)}$ term M, defined as

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

By Proposition 3.3.40, we can find a least fixed point term in $\Lambda^0_{((2\to 2)\to(2\to 2))\to(2\to 2)}$. For example,

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

and

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

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

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

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

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

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

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

Corollary 3.3.44 Let $M, N \in \Lambda^0$. $M \simeq N$ implies $[\![M]\!] = [\![N]\!]$, and $[\![M]\!] = [\![N]\!]$ implies $M \simeq N$. □

We show in Appendix A that $M \simeq N$ implies that M and N reduce to the same normal form (Lemma A.0.57). Because terms M and N in Corollary 3.3.44 are already in normal forms (they are in Λ^0), it follows that not only does $[\![M]\!] = [\![N]\!]$ imply $M \simeq N$, it also implies $M \equiv N$ (modulo associativity and commutativity of $\sqcup$ and $\sqcap$).

Theorem 3.3.45 Let $\sigma \in \Gamma$. Then there is a least fixed point term $Y \in \Lambda^0_{(\sigma\to\sigma)\to\sigma}$ such that for all terms $F \in \Lambda^0_{\sigma\to\sigma}$,

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

□

PROOF.

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

- $N_{k,l} \equiv x_h$, where x_h is in $\vec{x}$ and of type $\mathit{2}$.
  Thus x_h must not occur in the conjunctive term $\bigsqcap_{j\in J_i} M_{i,j}$. Otherwise we could have proved $M_{i,j} \preceq N_{k,l}$ for some $j \in J_i$, a contradiction. By the construction of environment ρ above, x_h is mapped to 0. Hence, $[\![\bigsqcap_{l\in L_k} N_{k,l}]\!]\rho = 0$.

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

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

Example 3.3.43 Suppose we have the following two $\Lambda^0_{((\mathit{2}\to\mathit{2})\to(\mathit{2}\to\mathit{2}))\to(\mathit{2}\to\mathit{2})}$ terms, Y and Z, defined by

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

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

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

(As a side note, the 10 elements in domain $D_{(2\to 2)\to(2\to 2)}$ happen to be defined in Example 3.3.38 by the 10 $\Lambda^0_{(2\to 2)\to(2\to 2)}$ terms.) □

Proposition 3.3.42 (Completeness) Let $M, N \in \Lambda^0$. Then, $[\![M]\!] \sqsubseteq [\![N]\!]$ implies $M \preceq N$. □

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

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

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

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

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

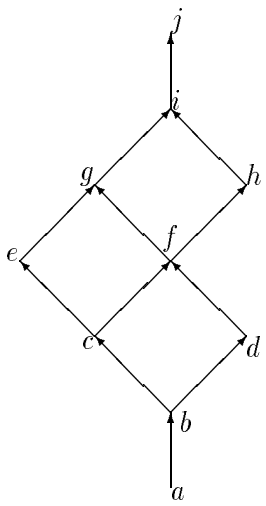
Given such an index $i \in I$, we now describe how to construct the environment ρ. Let x_h be a bound variable in $\vec{x}$. Suppose that x_h is of type 2 and $M_{i,j} \equiv x_h$ for some $j \in J_i$, then we map x_h to 1 in environment ρ. If x_h never occurs in the conjunctive term $\bigsqcap_{j\in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

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

and map x_h to the following function in ρ,

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

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



$$
\begin{array}{lcl}
j & = & [\bot \mapsto \top, \quad \| \mapsto \top, \quad \top \mapsto \top], \\
i & = & [\bot \mapsto \|, \quad \| \mapsto \top, \quad \top \mapsto \top], \\
h & = & [\bot \mapsto \bot, \quad \| \mapsto \top, \quad \top \mapsto \top], \\
g & = & [\bot \mapsto \|, \quad \| \mapsto \|, \quad \top \mapsto \top], \\
f & = & [\bot \mapsto \bot, \quad \| \mapsto \|, \quad \top \mapsto \top], \\
e & = & [\bot \mapsto \|, \quad \| \mapsto \|, \quad \top \mapsto \|], \\
d & = & [\bot \mapsto \bot, \quad \| \mapsto \bot, \quad \top \mapsto \top], \\
c & = & [\bot \mapsto \bot, \quad \| \mapsto \|, \quad \top \mapsto \|], \\
b & = & [\bot \mapsto \bot, \quad \| \mapsto \bot, \quad \top \mapsto \|], \\
a & = & [\bot \mapsto \bot, \quad \| \mapsto \bot, \quad \top \mapsto \bot].
\end{array}
$$

$$
\begin{array}{lcl}
\top & = & [0 \mapsto 1, \quad 1 \mapsto 1], \\
\| & = & [0 \mapsto 0, \quad 1 \mapsto 1], \\
\bot & = & [0 \mapsto 0, \quad 1 \mapsto 0].
\end{array}
$$

Figure 3.1: The 10 elements in domain $D_{(2\to 2)\to(2\to 2)}$.
Note. The three elements in domain $D_{2\to 2}$ are denoted by $\bot, \|$, and $\top$, with $\bot \sqsubseteq \| \sqsubseteq \top$. The 10 elements in domain $D_{(2\to 2)\to(2\to 2)}$ are denoted by $a, b, c, d, e, f, g, h, i,$ and j. The lower an element in the lattice, the less defined the element is.

Clearly the least fixed point function y can be described by

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

Which describes that for element $x \in D_{2\to 2}$, if $e \sqsubseteq x$ then the least fixed point of x is greater than $\|$, and if $i \sqsubseteq x$ then the least fixed point of x is greater than $\top$, otherwise the least fixed point of x is $\bot$. By using Proposition 3.3.40, $step_{e,\|}$ and $step_{i,\top}$ can be defined by

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

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

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

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

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

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

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

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

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

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

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

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

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

The normal form of F is in language Λ^0_σ. $\diamond$

Example 3.3.41 There is a function y in domain $D_{((2\to 2)\to(2\to 2))\to(2\to 2)}$ such that for all elements x in domain $D_{(2\to 2)\to(2\to 2)}$, $(y\ x)$ is the least fixed point of x. Can we find a term Y in language $\Lambda^0_{((2\to 2)\to(2\to 2))\to(2\to 2)}$ such that $[\![Y]\!] = y$?

Before calculating Y, let us first draw a diagram of domain $D_{(2\to 2)\to(2\to 2)}$. The diagram in Figure 3.1 illustrates the ordering of the 10 elements in domain $D_{(2\to 2)\to(2\to 2)}$. The functionalities of the 10 elements are also described as maps from domain $D_{2\to 2}$ to $D_{2\to 2}$ in the illustration. We write the 3 elements in $D_{2\to 2}$ as $\bot, \|$, and $\top$, with the ordering $\bot \sqsubseteq_{2\to 2} \| \sqsubseteq_{2\to 2} \top$. Note that they are definable in $\Lambda^0_{2\to 2}$ by $\bot = [\![\lambda\, x\, .\, 0]\!]$, $\| = [\![\lambda\, x\, .\, x]\!]$, and $\top = [\![\lambda\, x\, .\, 1]\!]$.

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

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

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

In addition to the soundness property inherited from language Λ, Λ^0 has the following nice properties.

Proposition 3.3.39 Let $M, N \in \Lambda^0$. The normal forms for MN, $\lambda x\,.\,M$, $M\sqcup N$, and $M\sqcap N$ are all in Λ^0. □

Proof. By Definition 3.3.37, term $\lambda x\,.\,M$ is in normal form if M is in normal form.

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

Proposition 3.3.40 (Definability) For every element $f\in D_\sigma, \sigma\in\Gamma$, there is a term $F\in\Lambda^0_\sigma$ such that $[\![F]\!]=f$. □

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

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

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

Furthermore, an element $f\in D_\sigma$ can be expressed as the least upper bound of a set of step functions in D_σ. That is, $f=\bigsqcup_{i\in I} step_{a_i,b_i}$ for some index set I. This standard construction can be found, for example, in Plotkin [61]. Since D_τ and D_γ are finite, the index set I is finite.

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

□

The motivation behind the above definition is to make language Λ^0 as simple as possible. Therefore, we let Λ^0 include only those closed terms which are in normal form and whose matrices are the disjunction of conjunctions of function applications, with the function arguments being in Λ^0. In addition, the bound variables of a Λ^0 term are never used as function arguments in the matrix; they are only used as function names, applied to other Λ^0 terms. We will later show that Λ^0 is actually not a very restricted language. In fact, it is expressive enough to describe all the elements in the semantic domains $D_\sigma, \sigma \in \Gamma$. On the other hand, the simplicity of Λ^0 helps in showing that semantically weaker implies syntactically weaker in Λ^0. Note that, by induction, it can be shown that for any given type $\sigma \in \Gamma$, there are only finite number of Λ^0_σ terms.

Example 3.3.38 Assume that variable f has type $\mathit{2} \to \mathit{2}$ and variable x has type $\mathit{2}$. Then the following are the only 10 terms in $\Lambda^0_{(\mathit{2}\to\mathit{2})\to(\mathit{2}\to\mathit{2})}$ (ignoring the variations introduced by α–congruence and by associativity and commutativity of $\sqcup$ and $\sqcap$):

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

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

Table 3.1: The value table of application $[\![M]\!]\,d$ and $[\![N]\!]\,d$, for all $d \in D_{\mathbb{2}\to\mathbb{2}}$.
Note The two sets of results agree on all three elements in domain $D_{\mathbb{2}\to\mathbb{2}}$, showing that $[\![M]\!] = [\![N]\!]$.

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

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

By Table 3.1, it is clear that $[\![M]\!] = [\![N]\!]$. However, neither $M \preceq N$ nor $N \preceq M$ is provable according to rules in Definition 3.2.28 and 3.2.29.

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

3.3 A Complete Sub–Language Λ^0 of Λ

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

Definition 3.3.37 The sub–language Λ^0_σ of Λ_σ is inductively defined for each type $\sigma \in \Gamma$ as follows.

To summarize, we have for all tuple $p \in \prod_{i\in I} J_i$ and all $i \in I$,

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

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

We conclude that conjunctive terms are strongly normalizable.

This completes the proof that all Λ terms are strongly normalizable. ◇

Proposition 3.2.34 *$\boldsymbol{\beta} \sqcup \sqcap\boldsymbol{d}$ is Church–Rosser.* □

Proof. See Appendix A. ◇

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

Proposition 3.2.35 (Soundness) Let $L, M, N \in \Lambda$. Then,

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

for all environments ρ. □

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

The above means the that the syntactically weaker relation $\preceq$ and the reduction relation $\boldsymbol{\beta} \sqcup \sqcap\boldsymbol{d}$ are faithful to the semantics of the language Λ. But is the semantically weaker relationship between two Λ terms completely captured by the syntactic notions of $\preceq$ and $\boldsymbol{\beta} \sqcup \sqcap\boldsymbol{d}$? If it is, then in order to determine the semantic relationship for any two terms $M, N \in \Lambda$, we can simply calculate the normal forms of M and N and then compare the two normal forms using the $\preceq$ rules. Unfortunately, the answer is no.

Proposition 3.2.36 (Incompleteness) *There exist normal forms $M, N \in \Lambda$ such that $[\![M]\!]\rho \sqsubseteq [\![N]\!]\rho$ for all environments ρ, but $M \preceq N$ is not provable.* □

For a conjunctive term $\bigsqcap_{i\in I} M_i$, its strong normalization proof is similar to the disjunction case above, but is slightly more complicated due to ***d***–reduction.[1] Let us define $\xi(M)$ to be the maximal number of $\sqcup$ symbol in the terms reducible from M. That is,

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

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

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

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

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

[1] We can no longer use a multiset induction based on the function μ defined by

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

The induction will fail.

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

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

and the induction fails.

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

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

and

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

The induction fails.

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

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

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

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

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

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

- $\bigsqcup_{i\in I} M_i \to_{\sqcup} \bigsqcup_{i\in I-J} M_j$,
  where $\emptyset \neq J \subset I$. This is a one–step $\sqcup$–reduction at the top level of the disjunction. Then, it is clear that $\mu(\bigsqcup_{i\in I-J} M_i) \sqsubset_{\mathcal{M}(N)} \mu(\bigsqcup_{i\in I} M_i)$.

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

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

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

is $\boldsymbol{\beta} \sqcup \sqcap \boldsymbol{d}$–strongly normalizable; That is, there is no infinite $\boldsymbol{\beta} \sqcup \sqcap \boldsymbol{d}$–reduction sequence starting with M.

We will use a technique from Dershowitz and Manna [20], based on multiset orderings, to show that every term in language Λ is strongly normalizable. A multiset is a bag, where an element may occur more than once. The equality $A = B$ for two multisets A and B means that any element occurring n times in A also occurs exactly n times in B, and *vice versa*. Also, $A \uplus B$ is the multiset containing $m + n$ occurrences of any element occurring m times in A and n times in B.

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

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

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

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

Proposition 3.2.33 Every term $M \in \Lambda$ is strongly normalizable. □

Proof. It is well known that the simply typed λ–calculus is strongly normalizable (see, for example, Girard, Taylor & Lafont [28]). Language Λ differs from simply typed λ–calculus in that it introduces new terms and new reduction rules for ground type $\mathit{2}$. It suffices to show that all the newly introduced terms are $\boldsymbol{\beta} \sqcup \sqcap \boldsymbol{d}$ strongly normalizable to complete the proof. There are four classes of new terms: $0, 1, M \sqcup N$, and $M \sqcap N$, where $M, N \in \Lambda_{\mathit{2}}$. It is clear that both 0 and 1 are strongly normalizable. It remains to show that both $\bigsqcup_{i \in I} M_i$ and $\bigsqcap_{i \in I} M_i$ are strongly normalizable if each M_i is.

This completes the $\Longrightarrow$ part of the second assertion in this lemma. The proofs for the $\Longrightarrow$ parts of the other four assertions in this lemma are similar to the above. The proofs for the $\Longleftarrow$ parts are straightforward, except for the first assertion, which has no $\Longleftarrow$ part. $\Diamond$

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

Definition 3.2.32 The reduction relations $\boldsymbol{\beta}$, $\sqcup$, $\sqcap$, and $\boldsymbol{d}$ on language Λ are defined as follows.

$$\boldsymbol{\beta} = \{((\lambda x\,.M)N,\ M[x := N]) \mid M, N \in \Lambda\}.$$

$$\sqcup = \{(\bigsqcup_{i\in I} M_i,\ \bigsqcup_{i\in I-J} M_i) \mid M_i \in \Lambda_2, I \supset J \neq \emptyset,$$
$$\text{and for all } j \in J, \text{there exists a } i \in I - J, \text{such that } M_j \preceq M_i\}.$$

$$\sqcap = \{(\bigsqcap_{i\in I} M_i,\ \bigsqcap_{i\in I-J} M_i) \mid M_i \in \Lambda_2, I \supset J \neq \emptyset,$$
$$\text{and for all } j \in J, \text{there exists a } i \in I - J, \text{such that } M_i \preceq M_j\}.$$

$$\boldsymbol{d} = \{(\bigsqcap_{i\in I} \bigsqcup_{j\in J_i} M_{i,j},\ \bigsqcup_{p\in\prod_{i\in I} J_i} \bigsqcap_{i\in I} M_{i,p|i}) \mid M_{i,j} \in \Lambda_2, |I| > 1,$$
$$\text{and for some } i \in I, |J_i| > 1\}.$$

□

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

Let $\boldsymbol{r}$ be a reduction relation on Λ. We use $\rightarrow_r$ to denote the compatible closure of $\boldsymbol{r}$, use $\rightarrow_r^*$ to denote the reflexive and transitive closure of $\rightarrow_r$, and use $\leftrightarrow_r^*$ to denote the equivalent relation generated by $\rightarrow_r^*$. We also use $\boldsymbol{rs}$ to denote the reduction relation $\boldsymbol{r} \cup \boldsymbol{s}$. The notations $\rightarrow$ and $\rightarrow^*$ are, respectively, shorthands for $\rightarrow_{\beta\sqcup\sqcap d}$ and $\rightarrow^*_{\beta\sqcup\sqcap d}$. The standard definitions of $\boldsymbol{r}$*–redex* and $\boldsymbol{r}$*–normal form* are used. We will use *normal form* as an abbreviation for $\boldsymbol{\beta} \sqcup \sqcap \boldsymbol{d}$–normal form. A term $M \in \Lambda$ is called *strongly normalizable* iff M

- $F \preceq G$ is an axiom described in Definition 3.2.29, or it is the reflexivity axiom (*i.e.*, $F \equiv G$); or

- some intermediate $\preceq$ relationships between Λ terms are proved first, and then the proof for $F \preceq G$ are obtained from those intermediate results either by using the compatibility inference rules (as described in Definition 3.2.28) or by using the transitivity inference rule (*i.e.*, $F \preceq G$ because $F \preceq H$ and $H \preceq G$ for some H).

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

For example, suppose we want to prove the $\Longrightarrow$ part of the second assertion in this lemma: $EF \preceq GH$ iff $E \preceq G$ and $F \preceq H$. We prove it by showing that the following assertion is true: if there is a proof for $EF \preceq GH$, then there are also proofs for both $E \preceq G$ and $F \preceq H$. What would a proof for $EF \preceq GH$ look like? There are two cases:

- $EF \preceq GH$ is simply proved by an axiom. The only applicable axiom in this setting is the reflexivity axiom. That is, $EF \equiv GH$. It follows that $E \equiv G$ and $F \equiv H$. Then, by reflexivity, we have proofs for both $E \preceq G$ and $F \preceq H$.

- $EF \preceq GH$ is the result obtained by using one of those inference rules on some intermediate results. The only applicable rules in this setting is the application rule or the transitivity rule. Suppose that $EF \preceq GH$ follows from the application rule. Then either $E \preceq G$ and $F \equiv H$, or $E \equiv G$ and $F \preceq H$. In either case, by reflexivity, we know that there are proofs for both $E \preceq G$ and $F \preceq H$. On the other hand, suppose that $EF \preceq GH$ follows from the transitivity rule. Then there is a term $L \equiv MN$ such that both $EF \preceq L$ and $L \preceq GH$ can be proved. Furthermore, both of the proofs for $EF \preceq L$ and $L \preceq GH$ are smaller than the proof for $EF \preceq GH$ because they are parts of it. Therefore, there is no circularity (*i.e.*, $L \not\equiv EF$). Since we can prove $EF \preceq MN$, then, by induction hypothesis, there must be proofs for both $E \preceq M$ and $F \preceq N$. Similarly, there must be proofs for both $M \preceq G$ and $N \preceq H$. By transitivity, it follows that we can prove both $E \preceq G$ and $F \preceq H$.

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

Relation $\preceq$ imposes a rather weak theory on Λ. For example, we cannot even prove that $1 \preceq (\lambda\, x_2\, .\, 1)\, 1$, although it is certainly true by semantic reasoning. It is clear that some reduction rules, β–reduction in the above case, will be needed if a more complete theory is desirable. On the other hand, the simplicity of $\preceq$ has its virtue. For example, it is easy to check whether two Λ terms satisfy the $\preceq$ relationship or not, as shown by Lemma 3.2.31.

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

Lemma 3.2.31 Let $E, F, G, H \in \Lambda$, and let $M_i, N_j \in \Lambda_2$ for $i \in I$ and $j \in J$, where I and J are some finite index sets. Then,

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

□

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

- (application)

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

- (abstraction)

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

- ($\sqcup$)

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

- ($\sqcap$)

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

□

Definition 3.2.29 The relation $\preceq$ on language Λ is the compatible, reflexive, and transitive relation induced by the following axioms for all terms $L, M, N \in \Lambda_{\mathbb{2}}$.

- (0 and 1)

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

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

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

- (idempotence)

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

- (commutativity)

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

- (associativity)

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

□

Definition 3.2.30 Let $M, N \in \Lambda$. Then $M \simeq N$ if $M \preceq N$ and $N \preceq M$. □

- Type 2 denotes the domain $D_2 = \{0, 1\}$, with the ordering $0 \sqsubseteq_2 1$, and
- Type $\sigma \to \tau$ denotes the domain $D_{\sigma\to\tau} = \{f \mid f$ is a total, monotonic function from domain D_σ to domain $D_\tau\}$, with the ordering $f \sqsubseteq_{\sigma\to\tau} g$ iff $f\,x \sqsubseteq_\tau g\,x$ for all $x \in D_\sigma$.

□

It can be shown that for each type $\sigma \in \Gamma$, domain D_σ is a finite and distributive lattice.

Definition 3.2.27 Let environment ρ be a total function from typed variables to the union of all finite domains, $\bigcup_{\sigma\in\Gamma} D_\sigma$. Let $[\![M]\!]\rho$ be the interpretation of a term $M \in \Lambda$ under the environment ρ, and be defined as follows:

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

□

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

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

Definition 3.2.28 A binary relation $\mathcal{R}$ on language Λ is *compatible* if the following inference rules apply for all $F, G, H \in \Lambda$ and all $L, M, N \in \Lambda_2$.

soon work on the semantic domains $\mathcal{3}, \mathcal{4}, \mathcal{5}, \ldots$, which are domains each containing a single chain of $3, 4, 5, \ldots$ elements, it is easier, both for writing and reading, to use the literals $0, 1, 2, 3, 4, \ldots$ to describe the ordering among the semantic elements. (See Section 3.5.) This notation also gives us the same metalanguage for syntactic terms and their semantic denotations.

Definition 3.2.24 The set Γ of type expressions is inductively defined as follows.

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

□

Definition 3.2.25 The language Λ, along with the sub–language Λ_σ for each type $\sigma \in \Gamma$, is inductively defined as follows.

- $x_\sigma \in \Lambda_\sigma$, where x_σ is a variable of type σ,
- $(M\,N) \in \Lambda_\tau$ if $M \in \Lambda_{\sigma\to\tau}$ and $N \in \Lambda_\sigma$,
- $(\lambda\, x_\sigma \,.\, M) \in \Lambda_{\sigma\to\tau}$ if $x_\sigma \in \Lambda_\sigma$ and $M \in \Lambda_\tau$,
- $0, 1 \in \Lambda_{\mathcal{2}}$, and
- $(M \sqcup N), (M \sqcap N) \in \Lambda_{\mathcal{2}}$ if $M, N \in \Lambda_{\mathcal{2}}$.

□

Type $\mathcal{2}$ is the *ground* type of language Λ. Language Λ can be viewed as the set of the simply typed λ–terms constructed from the ground type $\mathcal{2}$ and the four predefined constants: $0, 1 \in \Lambda_{\mathcal{2}}$, and $\sqcup, \sqcap \in \Lambda_{\mathcal{2}\to\mathcal{2}\to\mathcal{2}}$. The type constructor $\to$ is right associative. That is, $\sigma_1 \to \sigma_2 \to \ldots \to \sigma_n$ is a shorthand for $(\sigma_1 \to (\sigma_2 \to (\ldots \to \sigma_n)))$. Γ can also be defined inductively by the following: $\mathcal{2} \in \Gamma$, and $\sigma_1 \to \sigma_2 \to \ldots \to \sigma_n \to \mathcal{2} \in \Gamma$ if $\sigma_1, \sigma_2, \ldots \sigma_n \in \Gamma$. We take the liberty to omit some parentheses and type subscripts in a Λ term if it is clear to do so. All type expressions in Γ and all terms in Λ are understood to be of finite length.

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

Definition 3.2.26

iteration approach, used above to compute the least fixed point of F in (3.1) by successively calculating more accurate estimates, is also shown to be sound and complete.

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

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

3.2 A Language Λ for Finite Domains

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

Language Λ can be viewed as a variation of the simply typed lambda calculus. They differ in that Λ is augmented with four predefined constants — $0, 1, \sqcup$, and $\sqcap$ — and their associated reduction rules. They also differ in their interpretations of type expressions. Most of the definitions in this section are standard or intuitive, many of them borrowed from Barendregt [8].

Throughout this chapter, the two elements in the semantic domain $\mathit{2}$ will be denoted by 0 and 1, instead of the usual $\bot_{\mathit{2}}$ and $\top_{\mathit{2}}$ used in the previous chapter. Because we will

But how would the frontier method reach this result? The frontier method will approximate the maximal $\perp$–frontier of the least fixed point from the least element in domain $\mathit{2}^{11} \to \mathit{2}$. It is not difficult to see that the frontier method will have to make, step by step, 2^{10} approximations to reach the above maximal $\perp$–frontier, which is right in the middle of every ascending chain from the least element $\lambda x_0 . \lambda x_1 . \ldots . \lambda x_{10} . \perp_{\mathit{2}}$ (whose maximal $\perp$–frontier is $\{\langle \top_{\mathit{2}}, \top_{\mathit{2}}, \top_{\mathit{2}}, \top_{\mathit{2}}, \top_{\mathit{2}}, \top_{\mathit{2}}, \top_{\mathit{2}}, \top_{\mathit{2}}, \top_{\mathit{2}}, \top_{\mathit{2}}, \top_{\mathit{2}} \rangle\}$) to the greatest element $\lambda x_0 . \lambda x_1 . \ldots . \lambda x_{10} . \top_{\mathit{2}}$ (whose maximal $\perp$–frontier is $\emptyset$) in the domain $\mathit{2}^{11} \to \mathit{2}$. In such "badly behaved" cases, the frontier method is very inefficient compared to a symbolic evaluation method. This phenomenon has been observed by Clack and Peyton Jones [19,60]. Hunt and Hankin [43,44] further suggest that higher–order functional programs are often badly behaved in this way.

A symbolic evaluation method for computing least fixed points is proposed by Martin [50]. However, the method is limited to first–order functions over (the Cartesian products of) domain $\mathit{2}$. In this chapter, we will develop a syntactic method suitable for symbolic calculation of least fixed points over (the Cartesian product of) the monotonic function spaces generated by domain $\mathit{2}$. Because higher–order functions over domain $\mathit{2}$ are themselves elements in the monotonic function spaces generated by domain $\mathit{2}$, our method naturally yields a syntactic approach to computing least fixed points of higher–order functions.

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

We will show that, in a sufficiently expressive sub–language of Λ, the proposed calculus is sound and complete with respect to the semantics. Because functions for computing least fixed points are themselves elements in some monotonic function spaces, we can directly perform the least fixed point computation on Λ by means of symbolic calculation. In fact, for a given finite type σ, we will show that there is a fixed point term $Y \in \Lambda_{(\sigma \to \sigma) \to \sigma}$ such that for any term F in the sub–language of $\Lambda_{\sigma \to \sigma}$, not only do YF and $F(YF)$ denote the same element in domain D_σ, they also reduce to the same normal form in Λ_σ. The

least fixed point finding is used to calculate the abstract semantics of the program. Much work had been performed to devise elegant and effective methods to calculate least fixed points on finite domains. Most notable is the frontier method developed by Clack and Peyton Jones [19,60], Martin and Hankin [49,50], and Hunt and Hankin [43,44], which we have described in Section 2.3.

The frontier method is attractive in several ways. The representation is economical in space. It also allows fast function application. (We simply check whether the argument is weaker than any of the elements in the maximal $\perp$–frontier. If it is, then the result is $\perp_{\mathcal{2}}$; otherwise the result is $\top_{\mathcal{2}}$.)

Though elegant, there are several drawbacks in the frontier method. First, the frontier representations do not compose easily. Suppose that we have the frontier representations of functions f and g, what is the frontier representation of the functional composition $f \circ g$? It seems that we do not have much choice but to calculate it from scratch. A functional program is very likely to be built up from smaller functional components by using the mechanism of abstraction, application, and composition. But the frontier method does not provide such a building mechanism, unless, of course, when functions are fully applied to their arguments.

Secondly, the frontier method is carried out mainly on the semantic domains of a program; the method pays little attention to the program text itself. This may cause great inefficiency. Consider a function $f \in \mathcal{2}^{(12)}$, which is defined as the least fixed point of the following functional F,

$$F \equiv \lambda\, f\,.\,\lambda\, x_0\,.\,\lambda\, x_1 \ldots .\lambda\, x_{10}\,.\, x_0 \sqcup (f\ x_0\ x_1\ \ \ldots\ x_{10}), \tag{3.1}$$

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

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

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

$$\lambda\, x_0\,.\,\lambda\, x_1 \ldots .\lambda\, x_{10}\,.\, \perp_{\mathcal{2}},$$

and takes only two iterations. By the above result, we also know that (the uncurried version of) f has maximal $\perp$–frontier $\{\langle \perp_{\mathcal{2}}, \top_{\mathcal{2}}, \top_{\mathcal{2}}, \top_{\mathcal{2}}, \top_{\mathcal{2}}, \top_{\mathcal{2}}, \top_{\mathcal{2}}, \top_{\mathcal{2}}, \top_{\mathcal{2}}, \top_{\mathcal{2}}, \top_{\mathcal{2}}, \rangle\}$.

Chapter 3

A Syntactic Method for Least Fixed Point Finding

We describe a method for finding the least fixed points of higher–order functions over finite domains using symbolic manipulation. Least fixed point finding is an essential component in the calculation of the abstract semantics of functional programs, providing the foundation for program analyses based on abstract interpretation. Previous methods for least fixed point finding have primarily used semantic approaches, which often must traverse large portions of the semantic domain and which may be very inefficient even for simple programs.

The proposed syntactic method is based on an augmented simply typed lambda calculus where the symbolic representation of each function produced in the fixed point iteration is transformed to a syntactic normal form. We show the method to be sound and complete. Examples are presented to show how this method can be used to perform strictness analysis over non-flat domains.

3.1 Motivation and Introduction

We have shown in Chapter 2 that finding the least fixed points of monotonic functions over finite domains is an important task in abstract interpretation. In abstract interpretation, a standard (or non–standard) semantics of a functional program is abstracted to a monotonic function over finite domains, and, if the program contains recursive definitions,

we check whether this is the least fixed point. We then evaluate element $\langle \top, \top, \top \rangle$, the only element in the $\bot$–frontier of $F^{(0)}$, under function $F(F^{(0)})$, where

$$F(F^{(0)}) = \lambda \langle x_2, y_2, z_2 \rangle . (x \sqcap y \sqcap z) \sqcup (F^{(0)} \langle \top, y, z \rangle) \sqcup (F^{(0)} \langle \bot, x, z \rangle) \sqcup (F^{(0)} \langle \bot, \bot, x \sqcap y \rangle).$$

We find out that $F(F^{(0)}) \langle \top, \top, \top \rangle = \top \neq \bot$. Hence, $F^{(0)}$ is not the least fixed point. Lowering $F^{(0)}$ according to element $\langle \top, \top, \top \rangle$, we have

$$F^{(1)} = \{\langle \bot, \top, \top \rangle, \langle \top, \bot, \top \rangle, \langle \top, \top, \bot \rangle\}$$

as the next estimate for the least fixed point. Evaluate elements $\langle \bot, \top, \top \rangle, \langle \top, \bot, \top \rangle$, and $\langle \top, \top, \bot \rangle$ under function $F(F^{(1)})$, where

$$F(F^{(1)}) = \lambda \langle x_2, y_2, z_2 \rangle . (x \sqcap y \sqcap z) \sqcup (F^{(1)} \langle \top, y, z \rangle) \sqcup (F^{(1)} \langle \bot, x, z \rangle) \sqcup (F^{(1)} \langle \bot, \bot, x \sqcap y \rangle).$$

We find out that

$$F(F^{(1)}) \langle \top, \bot, \top \rangle = \bot \text{ and } F(F^{(1)}) \langle \top, \top, \bot \rangle = \bot, \text{ but } F(F^{(1)}) \langle \bot, \top, \top \rangle = \top \neq \bot.$$

Hence, $F^{(1)}$ is not the least fixed point either. Replacing element $\langle \bot, \top, \top \rangle$ in $F^{(1)}$ by $\langle \bot, \top, \bot \rangle$ and $\langle \bot, \bot, \top \rangle$, we have

$$F^{(2)} = \{\langle \top, \bot, \top \rangle, \langle \top, \top, \bot \rangle\}.$$

The iteration continues until we finally obtain $F^{(8)} = \emptyset$, which is the maximal $\bot$–frontier of the function $\lambda \langle x_2, y_2, z_2 \rangle . \top_2$, which must be the least fixed point of F. □

The details of the frontier approach to least fixed point calculation can be found in the work of Clack and Peyton Jones [19,60]. Martin and Hankin [49,50] extend the frontier approach to higher–order functions and to cases where the base domain is not *2*. Recently, Hunt and Hankin [43,44] give a clear account of the frontier method based on open sets. They also mention how to further abstract the abstract domains when they are still too large, and how to obtain an upper bound or a lower bound of the least fixed points in cases when the abstract domains are huge.

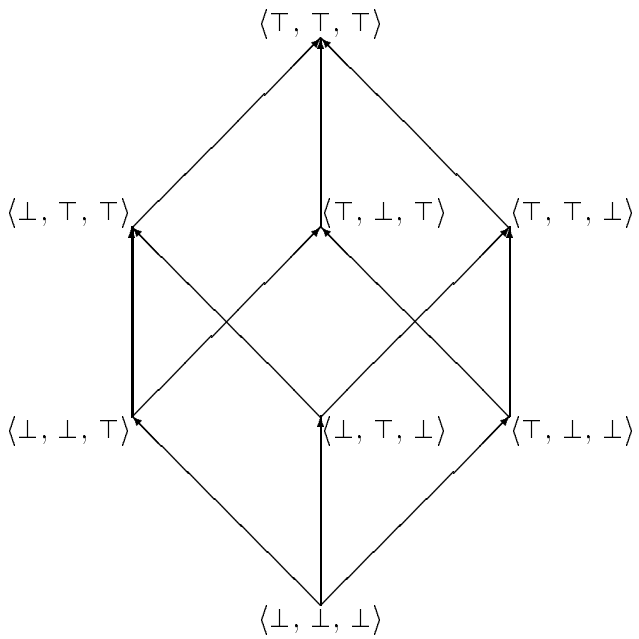


Figure 2.5: The lattice structure of domain $2 \times 2 \times 2$.

function space. By factoring the abstract domain using the above rules, we can always use 2 as the image domain in strictness analysis. For this reason, we often work on the uncurried version of a function instead of the curried one when using the frontier method. Let us revisit Example 2.3.22 to demonstrate how the frontier method works.

Example 2.3.23 We want to find the least fixed point of the following functional F, where

$$F = \lambda f_{2\times 2\times 2\to 2} \,.\, \lambda \langle x_2, y_2, z_2 \rangle \,.\, (x \sqcap y \sqcap z) \sqcup (f \langle \top, y, z \rangle) \sqcup (f \langle \bot, x, z \rangle) \sqcup (f \langle \bot, \bot, x \sqcap y \rangle).$$

Note that the above F is the uncurried version of the functional F in Example 2.3.22. The domain $2 \times 2 \times 2$ is illustrated in Figure 2.5.

The least fixed point iteration starts with $\{\langle \top, \top, \top \rangle\}$, which is the maximal $\bot$–frontier of the function $\lambda \langle x_2, y_2, z_2 \rangle \,.\, \bot_2$, the least defined element in domain $2 \times 2 \times 2 \to 2$. The first few iterations go as follows.

After assuming that

$$F^{(0)} = \{\langle \top, \top, \top \rangle\},$$

in domain $\bar{\alpha}$ are included (as compared to, say, the value table representation of the same function, which contains some redundancy). This makes easy the comparison of two frontiers to see if one is weaker than the other or not. For any two functions $f, g \in \bar{\alpha} \to 2$, $f \sqsubseteq_{\bar{\alpha}\to 2} g$ if and only if for all $x \in \mathbf{0}_g$, there exists a $y \in \mathbf{0}_f$ such that $x \sqsubseteq_{\bar{\alpha}} y$, where $\mathbf{0}_f$ and $\mathbf{0}_g$ are the maximal $\perp$–frontiers of f and g, respectively. This is easy to check, and is exactly one of the tasks needed in the least fixed point finding. The other task is to evaluate the next approximation $F^{(i+1)}$ from the previous approximation $F^{(i)}$ with the functional $F \in (\bar{\alpha} \to 2) \to (\bar{\alpha} \to 2)$. Let $\mathbf{0}^{(i)}$ be the maximal $\perp$–frontier of $F^{(i)}$. Then the functionality of $F(F^{(i)})$ can be obtained by interpreting the function f in the body of F by its maximal $\perp$–frontier $\mathbf{0}^{(i)}$, according to (2.1). Thus, $F(F^{(i)}) \sqsubseteq_{\bar{\alpha}\to 2} F^{(i)}$ if all elements in $\mathbf{0}^{(i)}$, are mapped by function $F(F^{(i)})$ to $\perp_2$. If that is the case, then $F(F^{(i)}) = F^{(i)}$ because both $F(F^{(i)}) \sqsubseteq_{\bar{\alpha}\to 2} F^{(i)}$ and $F^{(i)} \sqsubseteq_{\bar{\alpha}\to 2} F(F^{(i)})$. We then would have reached the least fixed point, whose maximal $\perp$–frontier is $\mathbf{0}^{(i)}$. If not all of the elements in $\mathbf{0}^{(i)}$ are mapped by function $F(F^{(i)})$ to $\perp_2$, then the function denoted by $\mathbf{0}^{(i)}$ is too weak to be the least fixed point. Consequently, we lower the maximal frontier $\mathbf{0}^{(i)}$ — and make stronger the function it will denote — by replacing each $x \in \mathbf{0}^{(i)}$ such that $F(F^{(i)})\ x \not\sqsubseteq_2 \perp_2$ by the strongest $x' \in \bar{\alpha}$ such that $x \not\sqsubseteq_{\bar{\alpha}} x'$. This will lead to a new maximal $\perp$–frontier $\mathbf{0}^{(i+1)}$, and the iteration for the least fixed point is continued.

Before moving to an example that explicitly describes the least fixed point finding iteration using the maximal $\perp$–frontier, we make two observations. First, we notice that the progression to the next maximal $\perp$–frontier, $\mathbf{0}^{(i+1)}$, from the current one, $\mathbf{0}^{(i)}$, occurs by replacing some element $x \in \mathbf{0}^{(i)}$ by the strongest $x' \in \bar{\alpha}$ such that $x \not\sqsubseteq_{\bar{\alpha}} x'$. This may cause great inefficiency because the maximal $\perp$–frontier is lowered step by step *no matter what* the functional F looks like. Functional F is only used to check if the least fixed point have been reached, but it is not utilized otherwise in getting the next approximation. We will show in Chapter 3 that the frontier method can be very inefficient even for very simple examples, and an alternative approach is proposed in Chapter 3 to better utilize the program denoting F to find the least fixed point faster. Secondly, we observe that the frontier method works best if the image domain $\bar{\beta}$ of the abstract function space $\bar{\alpha} \to \bar{\beta}$ is domain 2. This will not cause too much trouble, however, because

$$\bar{\alpha} \to (\bar{\beta} \to \bar{\gamma}) \cong \bar{\alpha} \times \bar{\beta} \to \bar{\gamma} \quad \text{and} \quad \bar{\alpha} \to \bar{\beta} \times \bar{\gamma} \cong (\bar{\alpha} \to \bar{\beta}) \times (\bar{\alpha} \to \bar{\gamma})$$

if $\bar{\alpha}, \bar{\beta}$, and $\bar{\gamma}$ are complete partially ordered sets, $\times$ is the product, and $\to$ is the continuous

efficiency of the above two tasks. Two representations for a function over a finite domain come to mind: we can use a value table, or we can use a symbolic expression.

Value tables are out of consideration because it consumes too much time to create the tables. Symbolic expressions are used in Example 2.3.22 to represent first–order functions over abstract domains. However, a naive application of symbolic expressions to represent higher–order functions over abstract domains will fail miserably, as we shall see in Chapter 3. The problem is that we can have two higher–order functions f and g over finite domains such that $f \sqsubseteq g$ but we cannot tell this relationship by a simple comparison of the symbolic representations of f and g. (Such a case is shown in Example 3.4.49.) We will show in next chapter how to deal with this problem. But right now let us look at an elegant method for finding least fixed points over finite domains, both for first–order and higher–order functions, by Clack and Peyton Jones [19,60]. It is called the *frontier* method.

The idea of a frontier representation of a function comes from the observation that every function f in domain $\alpha \to \mathit{2}$ can be described by a Scott-closed subset $\mathbf{0} \subseteq \alpha$ such that for every element $d \in \alpha$,

$$f(d) \sqsubseteq_{\mathit{2}} \bot_{\mathit{2}} \quad \text{iff} \quad d \in \mathbf{0}.$$

That is, $\mathbf{0}$ contains exactly those elements in domain α that will be mapped to $\bot_{\mathit{2}}$ by function f. For elements not in $\mathbf{0}$, they will be mapped to $\top_{\mathit{2}}$, the only other element in domain $\mathit{2}$, by function f. Since the Scott–closed subset $\mathbf{0}$ is downward–closed, it suffices to represent $\mathbf{0}$ by the set of maximal elements in $\mathbf{0}$. This set is called the *maximal* $\bot$*–frontier* representation of the function f. That is, $\mathbf{0}$ is the smallest set such that for function f,

$$f(d) \sqsubseteq_{\mathit{2}} \bot_{\mathit{2}} \quad \text{iff} \quad d \sqsubseteq_{\alpha} z \text{ for some } z \in \mathbf{0}, \tag{2.1}$$

where $\mathbf{0}$ now denotes the maximal $\bot$–frontier of f. Note that a *minimal* $\top$*–frontier* $\mathbf{1}$ can be similarly defined for every function f in $\alpha \to \mathit{2}$, such that

$$\top_{\mathit{2}} \sqsubseteq_{\mathit{2}} f(d) \quad \text{iff} \quad z \sqsubseteq_{\alpha} d \text{ for some } z \in \mathbf{1},$$

and works equally well.

How would the frontier representation help in finding the least fixed points? For a finite abstract domain $\bar{\alpha}$, the maximal $\bot$–frontier representation of any function $f \in \bar{\alpha} \to \mathit{2}$ is also finite. The representation is also economical in the sense that only relevant elements

Hence, f is the least fixed point of F, which is defined as

$$F = \lambda f_{2\to 2\to 2\to 2} . \lambda x_2 . \lambda y_2 . \lambda z_2 . (x \sqcap y \sqcap z) \sqcup (f \top y z) \sqcup (f \bot x z) \sqcup (f \bot \bot (x \sqcap y)).$$

We will show that F has the least fixed point $\lambda x_2 . \lambda y_2 . \lambda z_2 . \top_2$.

Starting with $\lambda x_2 . \lambda y_2 . \lambda z_2 . \bot_2$, the least defined element in domain $2 \to 2 \to 2 \to 2$, we will need exactly $2^3 = 8$ iterations to reach $\lambda x_2 . \lambda y_2 . \lambda z_2 . \top_2$, the least fixed point of F. The approximation sequence is the following:

$$\begin{aligned}
F^{(0)} &= \lambda x . \lambda y . \lambda z . \bot, \\
F^{(1)} &= \lambda x . \lambda y . \lambda z . x \sqcap y \sqcap z, \\
F^{(2)} &= \lambda x . \lambda y . \lambda z . y \sqcap z, \\
F^{(3)} &= \lambda x . \lambda y . \lambda z . (x \sqcup y) \sqcap z, \\
F^{(4)} &= \lambda x . \lambda y . \lambda z . z, \\
F^{(5)} &= \lambda x . \lambda y . \lambda z . (x \sqcap y) \sqcup z, \\
F^{(6)} &= \lambda x . \lambda y . \lambda z . y \sqcup z, \\
F^{(7)} &= \lambda x . \lambda y . \lambda z . x \sqcup y \sqcup z, \\
F^{(8)} &= \lambda x . \lambda y . \lambda z . \top.
\end{aligned}$$

□

It has been shown that strictness analysis for first order functional programs is deterministic exponential time complete with respect to program size [39,79]. For higher–order functional programs, the computational complexity is worse. However, it can be argued that the worst cases, such as Example 2.3.22, do not occur often in everyday programs. What we would like to see are methods for finding the least fixed points that work reasonably well in practice.

We observe that during each step of the least fixed point approximation, two tasks are performed: 1) the next approximation $F^{(i+1)}$ is defined using the previous approximation $F^{(i)}$ and the functional F, and 2) the result $F^{(i+1)}$ is compared to $F^{(i)}$ to see if they are equal.[5] The explicit representation of the intermediate results $F^{(i)}, i \in N$, which may be *functions* over abstract domains, is very important because it greatly effects the execution

[5]It suffices to see if $F^{(i+1)} \sqsubseteq F^{(i)}$. But this implies they are equal since $F^{(i)} \sqsubseteq F^{(i+1)}$.

and

$$\begin{aligned}
&\lambda\, x_1, x_2, \ldots, x_k \,.\, (e_0\, x_1\, x_2 \ldots x_{k-1}) \sqcup x_k,\\
&\lambda\, x_1, x_2, \ldots, x_k \,.\, (e_1\, x_1\, x_2 \ldots x_{k-1}) \sqcup x_k,\\
&\lambda\, x_1, x_2, \ldots, x_k \,.\, (e_2\, x_1\, x_2 \ldots x_{k-1}) \sqcup x_k,\\
&\ldots\\
&\lambda\, x_1, x_2, \ldots, x_k \,.\, (e_{2^k-1}\, x_1\, x_2 \ldots x_{k-1}) \sqcup x_k.
\end{aligned}$$

Each of them has length $2^{k-1}+1$. It is clear that each sequence is an ascending chain in domain $2^{(k+1)}$. Moreover,

$$\lambda\, x_1, x_2, \ldots, x_k \,.\, (e_{2^k-1}\, x_1\, x_2 \ldots x_{k-1}) \sqcap x_k \quad \sqsubseteq_{2^{(k+1)}} \quad \lambda\, x_1, x_2, \ldots, x_k \,.\, (e_0\, x_1\, x_2 \ldots x_{k-1}) \sqcup x_k$$

because

$$\lambda\, x_1, x_2, \ldots, x_k \,.\, (e_{2^k-1}\, x_1\, x_2 \ldots x_{k-1}) \sqcap x_k \quad \sqsubseteq_{2^{(k+1)}} \quad \lambda\, x_1, x_2, \ldots, x_k \,.\, x_k$$

and

$$\lambda\, x_1, x_2, \ldots, x_k \,.\, x_k \quad \sqsubseteq_{2^{(k+1)}} \quad \lambda\, x_1, x_2, \ldots, x_k \,.\, (e_0\, x_1\, x_2 \ldots x_{k-1}) \sqcup x_k.$$

If $e_{2^k-1} = \top_{2^{(k)}}$ and $e_0 = \bot_{2^{(k)}}$, then

$$\begin{aligned}
& \lambda\, x_1, x_2, \ldots, x_k \,.\, (e_{2^k-1}\, x_1\, x_2 \ldots x_{k-1}) \sqcap x_k\\
= \;& \lambda\, x_1, x_2, \ldots, x_k \,.\, x_k\\
= \;& \lambda\, x_1, x_2, \ldots, x_k \,.\, (e_0\, x_1\, x_2 \ldots x_{k-1}) \sqcup x_k
\end{aligned}$$

and we conclude that the combined ascending chain is of length $2(2^{k-1}+1)-1 = 2^k+1$ in domain $2^{(k+1)}$. If $e_{2^k-1} \neq \top_{2^{(k)}}$ or $e_0 \neq \bot_{2^{(k)}}$, then we conclude that the combined ascending chain is of length 2^k+2. This completes the proof. ◇

The above theorem implies that for a recursively defined, n–argument, first–order functional program `f`, whose abstract semantics will be the least fixed point for some functional in domain $2^{(n)} \to 2^{(2)}$, the worst–case cost for calculating `f`'s abstract semantics will be $O(2^n)$ by a bottom up iteration. The following example is a worst case instance for $n = 3$.

Example 2.3.22 Assume the function f gives the strictness property of some 3–argument first–order function, and is recursively defined as

$$f = \lambda\, x_2 \,.\, \lambda\, y_2 \,.\, \lambda\, z_2 \,.\, (x \sqcap y \sqcap z) \sqcup (f \top y\, z) \sqcup (f \bot x\, z) \sqcup (f \bot \bot (x \sqcap y)).$$

is to make sure that all the abstract domains we are working with only have finite height. Abstract domains with infinite ascending chains are not allowed. As a consequence, we know the approximation will always reach a fixed point after a finite number of iterations.

Take strictness analysis as an example. The abstract domain for a basic standard domain like *Nat* or *Bool* is the two element domain $\mathcal{2}$, which has height 2. For a well-typed program whose semantics is in the domain $\alpha \to \beta$, its strictness property will be in the abstract domain $\bar{\alpha} \to \bar{\beta}$, where $\bar{\alpha}$ is the abstract domain of α, and $\bar{\beta}$ abstract domain of β. By an inductive assumption that there are only finitely many elements in $\bar{\alpha}$ and $\bar{\beta}$, we infer that there are only finite number of monotonic functions from $\bar{\alpha}$ to $\bar{\beta}$. Since that there are only finite number of elements in $\bar{\alpha} \to \bar{\beta}$, there cannot be an infinite ascending chain in $\bar{\alpha} \to \bar{\beta}$. We conclude that such abstract domains have finite height and the corresponding strictness analysis will always terminate.

However, the height of a strictness abstract domain grows exponentially with respect to the number of input arguments of a functional program. This is the case even for first–order functional programs.

Theorem 2.3.21 Denote the two–element abstract domain $\mathcal{2}$ by $\mathcal{2}^{(1)}$ and the abstract domain $\mathcal{2} \to \mathcal{2}^{(n)}$ by $\mathcal{2}^{(n+1)}$. Then there is an ascending chain of length $2^n + 1$ in domain $\mathcal{2}^{(n+1)}$. □

PROOF. We use an induction on n. For the base case of $n = 0$, domain $\mathcal{2}$ does have an ascending chain of length $2 = 2^0 + 1$. For $k > 0$, by induction hypothesis, we assume that domain $2^{(k)}$ has an ascending chain $e_0, e_1, e_2, \ldots, e_{2^k-1}$.

The following are two sequences of elements in domain $2^{(k+1)}$:

$$\begin{aligned}
&\lambda\, x_1, x_2, \ldots, x_k \,.\, (e_0\, x_1\, x_2 \ldots x_{k-1}) \sqcap x_k, \\
&\lambda\, x_1, x_2, \ldots, x_k \,.\, (e_1\, x_1\, x_2 \ldots x_{k-1}) \sqcap x_k, \\
&\lambda\, x_1, x_2, \ldots, x_k \,.\, (e_2\, x_1\, x_2 \ldots x_{k-1}) \sqcap x_k, \\
&\ldots \\
&\lambda\, x_1, x_2, \ldots, x_k \,.\, (e_{2^k-1}\, x_1\, x_2 \ldots x_{k-1}) \sqcap x_k
\end{aligned}$$

[14,15]. Strictness analysis for functional programs was first exploited in Mycroft [51,52], which was used to transform call–by–name evaluation (normal–order reduction) into call–by–value evaluation (applicative–order reduction) in non–strict functional programs. Mycroft's work only deals with first–order functions. Burn, Hankin, and Abramsky extend Mycroft's work to higher–order functions.

2.3 Finding Least Fixed Points over Finite Domains

In previous sections we have seen that least fixed point finding is an important task in interpreting recursively defined functional programs, both in the standard semantics and in an abstract semantics. The least fixed point of a recursive function F in domain $\alpha \to \alpha$ can be calculated by a sequence of approximations starting with the least defined element $\bot_\alpha$ in domain α. Specifically, we calculate a sequence

$$F^{(0)}, F^{(1)}, F^{(2)}, \ldots, F^{(n)}, \ldots$$

where $F^{(0)} = \bot_\alpha$ and $F^{(n+1)} = F(F^{(n)}), n \geq 0$, until we find an integer k such that

$$F^{(k+1)} = F^{(k)}.$$

Then $F^{(k)}$ is a fixed point of F. That $F^{(k)}$ is the *least* fixed point of F is shown in Theorem 2.1.14.

In the standard semantics, the domain α above may contain an infinite ascending chain. An ascending chain is defined as a sequence of elements $e_0, e_1, e_2, e_3, \ldots$ in the domain such that

$$e_0 \sqsubseteq_\alpha e_1 \sqsubseteq_\alpha e_2 \sqsubseteq_\alpha e_3 \sqsubseteq_\alpha \ldots$$

but

$$e_0 \neq e_1, \quad e_1 \neq e_2, \quad e_2 \neq e_3, \quad \ldots.$$

It is possible that the least fixed point of function F can only be approximated by an infinite ascending chain.

Because abstract interpretation is often incorporated into the optimization phase of a compiler, we are mostly interested in abstract interpretations that always terminate. Otherwise we may have programs that fail to compile because they cause the compiler to loop forever. One way to insure the termination of the least fixed point finding approximation

since multiplication is strict in both of its arguments.

Starting with the abstraction semantics for predefined syntactic terms, we then can construct the abstract semantics of any program. As an example, let us revisit the the factorial program in Figure 2.2.

Example 2.2.20 What is the strictness property of the function `factorial` in Figure 2.2? First, we know that the literals `0` and `1` are mapped to element $\top_{\mathcal{2}}$ in the abstract domain $\mathcal{2}$. The strictness property of the predefined terms `=`, `*`, `-`, and the expression `if ... then ... else` have been given by Fact 2.2.19. Because function `factorial` is recursively defined, also by Fact 2.2.19, its strictness property can be approximated by the least fixed point of the following functional F over the abstraction domain $\mathcal{2} \to \mathcal{2}$,

$$F = \lambda\, f_{\mathcal{2}\to\mathcal{2}}\, .\, \lambda\, x_{\mathcal{2}}\, .\, (x \sqcap \top_{\mathcal{2}})\ \textbf{and}\ (\top_{\mathcal{2}} \sqcup (x \sqcap (f\ (x \sqcap \top_{\mathcal{2}})))).$$

The least fixed point of F can be approximated by an iteration starting with $\lambda\, x_{\mathcal{2}}\, .\, \bot_{\mathcal{2}}$, the least element in the abstract domain $\mathcal{2} \to \mathcal{2}$. The approximation reaches $\lambda\, x_{\mathcal{2}}\, .\, x_{\mathcal{2}}$ as the least fixed point of F. That is to say that program `factorial` has strictness property $\lambda\, x_{\mathcal{2}}\, .\, x_{\mathcal{2}}$.

We now have

- $(\lambda\, x_{\mathcal{2}}\, .\, x_{\mathcal{2}})(\bot_{\mathcal{2}}) = \bot_{\mathcal{2}}$, and
- $(\lambda\, x_{\mathcal{2}}\, .\, x_{\mathcal{2}})(\top_{\mathcal{2}}) = \top_{\mathcal{2}}$.

which means that

- program `factorial` is strict (that is, when it is applied to a non–terminating expression, the computation will not terminate), and
- when it is applied to a terminating expression, the computation may or may not terminate.

Notice that the above interpretation uses concretization functions based on Hoare power domains. The concretization function $Conc_{\mathrm{P}(\mathcal{2})\to\mathrm{P}(Nat)}$ maps $\{\bot_{\mathcal{2}}\}^*$ to $\{\bot_{Nat}\}$ and map $\{\top_{\mathcal{2}}\}^*$ to Nat. Also see Fact 2.2.18. □

The detailed development of strictness analysis for typed higher–order functions as well as the proofs for Fact 2.2.18 and Fact 2.2.19 can be found in Burn, Hankin, and Abramsky

Fact 2.2.19

1. Denote base domain D by $D^{(1)}$ and domain $D \to D^{(n)}$ by $D^{(n+1)}$. If $f \in D^{(n+1)}, f \neq \bot_{D^{(n+1)}}$, and f is strict in all its arguments, then

$$abs_{D^{(n+1)} \to 2^{(n+1)}}(f) = \lambda\, x_{1\,2}\, . \cdots \lambda\, x_{n\,2}\, . x_{1\,2} \sqcap \cdots \sqcap x_{n\,2},$$

   where $\sqcap$ is the greatest lower bound function over domain 2.

2. Let $if \in Bool \to \alpha \to \alpha \to \alpha$ be the standard semantics of the (possible higher–order) expression `if ...then ...else ...`, then

$$abs_{(Bool \to \alpha \to \alpha \to \alpha) \to (2 \to \bar{\alpha} \to \bar{\alpha} \to \bar{\alpha})}(if) = \lambda\, x_{2}\, . \lambda\, y_{\bar{\alpha}}\, . \lambda\, z_{\bar{\alpha}}\, . x_{2}\ \mathbf{and}\ (y_{\bar{\alpha}} \sqcup z_{\bar{\alpha}}),$$

   where $\sqcup$ is the least upper bound function, and $\mathbf{and} : 2 \to \bar{\alpha} \to \bar{\alpha}$ is defined by

$$\begin{aligned} \bot_{2}\ \mathbf{and}\ e_{\bar{\alpha}} &= \bot_{\bar{\alpha}}, \\ \top_{2}\ \mathbf{and}\ e_{\bar{\alpha}} &= e_{\bar{\alpha}}. \end{aligned}$$

3. Let f be a function from domain α to domain α, then

$$abs_{\alpha \to \bar{\alpha}}(fix(f)) \sqsubseteq_{\bar{\alpha}} fix(abs_{(\alpha \to \alpha) \to (\bar{\alpha} \to \bar{\alpha})}(f)).$$

□

Note that the above results are not definitions, they are *theorems* in the (Hoare power domain based) framework of abstract interpretation. Of particular interest is the third clause in the above fact, which states that the abstraction of a recursively defined function, $abs_{\alpha \to \bar{\alpha}}(fix(f))$, can be approximated, $\sqsubseteq_{\bar{\alpha}}$, by a recursively defined abstraction of the function, $fix(abs_{(\alpha \to \alpha) \to (\bar{\alpha} \to \bar{\alpha})}(f))$. This yields a method for abstract interpretation of recursively defined functions.

Now the abstraction functions operate upon the semantic domain. The abstraction function $abs_{\alpha \to \bar{\alpha}}$ is a function from standard domain α to its abstract domain $\bar{\alpha}$. However, we can define a *textual* abstraction which maps directly from the category of syntactic terms to the category of the abstract domains. This gives an *abstract* denotational semantics for functional programs, instead of their standard denotational semantics. For example, for the predefined multiplication operator for the natural numbers, `*`, we now have its strictness interpretation as

$$mult = \lambda\, x_{2}\, . \lambda\, y_{2}\, . x_{2} \sqcap y_{2}$$

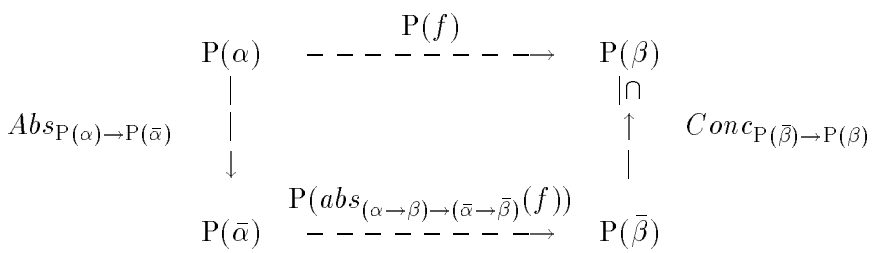


Figure 2.4: The relationship, in terms of the Hoare power domains, between a function $f \in \alpha \to \beta$ and its abstraction $abs_{(\alpha\to\beta)\to(\bar{\alpha}\to\bar{\beta})}(f) \in \bar{\alpha} \to \bar{\beta}$.

If we have the abstraction functions for the basic domains, such as *Nat*, *Bool*, and so on, then we can calculate the abstraction functions for higher–order functions over these basic domains. For strictness analysis, the abstraction functions for basic domains are straightforward. If D is a basic domain, such as *Nat*, then its abstraction onto 2, the two element domain for strictness property, is defined by

$$abs_{D\to 2}(d) = \begin{cases} \bot_2 & \text{if } d = \bot_D, \\ \top_2 & \text{if } d \neq \bot_D. \end{cases}$$

Then

$$Abs_{\mathrm{P}(D)\to\mathrm{P}(2)}(X) = \begin{cases} \{\bot_2\} & \text{if } X = \{\bot_D\}, \\ 2 & \text{otherwise;} \end{cases}$$

and

$$Conc_{\mathrm{P}(2)\to\mathrm{P}(D)}(X) = \begin{cases} \{\bot_D\} & \text{if } X = \{\bot_2\}, \\ D & \text{if } X = 2; \end{cases}$$

which are the abstraction and concretization functions based on the Hoare power domains. By Fact 2.2.18 and with the above results on *Abs* and *Conc*, it is clear that if the answer of a strictness analysis is $\top_2$, then it results from taking the least upper bound of $\{\bot_2, \top_2\}$, and its correct interpretation on the concrete domain should be D. Which means that it can be any element in the concrete domain D. It must not be interpreted as representing only the non–bottom elements in domain D.

Now it is time to derive the abstraction functions for the predefined terms of a functional language. Burn, Hankin, and Abramsky prove the following results.

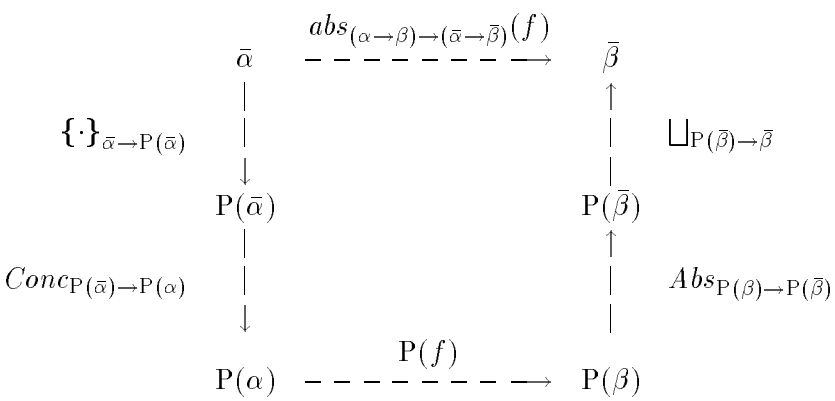


Figure 2.3: A diagram showing how to inductively define the abstraction function over domain $\alpha \to \beta$ based on the abstraction functions over domains α and β.

diagram in Figure 2.3 illustrates the construction here. The diagram shows that we can define $abs_{(\alpha\to\beta)\to(\bar{\alpha}\to\bar{\beta})}(f)$, the abstraction of a function $f \in \alpha \to \beta$, by

$$abs_{(\alpha\to\beta)\to(\bar{\alpha}\to\bar{\beta})}(f) \;=\; \bigsqcup_{\mathrm{P}(\bar{\beta})\to\bar{\beta}} \circ\; Abs_{\mathrm{P}(\beta)\to\mathrm{P}(\bar{\beta})} \circ \mathrm{P}(f) \circ Conc_{\mathrm{P}(\bar{\alpha})\to\mathrm{P}(\alpha)} \circ \{\cdot\}_{\bar{\alpha}\to\mathrm{P}(\bar{\alpha})}.$$

To show that the above definition really makes sense, Burn, Hankin, and Abramsky prove the following relationship between $\mathrm{P}(abs_{(\alpha\to\beta)\to(\bar{\alpha}\to\bar{\beta})}(f))$ and $\mathrm{P}(f)$, and show that the definition is safe.

Fact 2.2.18

$$\mathrm{P}(f) \sqsubseteq_{\mathrm{P}(\alpha)\to\mathrm{P}(\beta)} Conc_{\mathrm{P}(\bar{\beta})\to\mathrm{P}(\beta)} \circ \mathrm{P}(abs_{(\alpha\to\beta)\to(\bar{\alpha}\to\bar{\beta})}(f)) \circ Abs_{\mathrm{P}(\alpha)\to\mathrm{P}(\bar{\alpha})}$$

for every function f in domain $\alpha \to \beta$. □

We use the diagram in Figure 2.4 to illustrate the relationship between them. Fact 2.2.18 simply states that, in terms of the Hoare power domains, the abstract interpretation of f by $abs_{(\alpha\to\beta)\to(\bar{\alpha}\to\bar{\beta})}(f)$ will "cover" the standard semantics of f. That is, for all Scott–closed subset $X \subseteq \alpha$, we will have

$$(\mathrm{P}(f))X \subseteq (Conc_{\mathrm{P}(\bar{\beta})\to\mathrm{P}(\beta)} \circ \mathrm{P}(abs_{(\alpha\to\beta)\to(\bar{\alpha}\to\bar{\beta})}(f)) \circ Abs_{\mathrm{P}(\alpha)\to\mathrm{P}(\bar{\alpha})})X.$$

also apply it to any continuous function over a domain.[2] For a function $f \in D \to E$, the function $\mathrm{P}(f) \subseteq \mathrm{P}(D) \to \mathrm{P}(E)$ is defined as

$$(\mathrm{P}(f))(X) = \{f(x) \mid x \in X\}^*.$$

If f is the standard semantics of a functional program `f`, then in the sense of Hudak and Young [40], $\mathrm{P}(f)$ can be viewed as the *collecting interpretation* of program `f`, based on the Hoare power domain. The Hoare power domain construction has the pleasant property that it preserves continuity: $\mathrm{P}(f)$ is continuous if f is.[3]

We now define two natural mappings between D and $\mathrm{P}(D)$. The function $\{\cdot\}$ from domain D to its Hoare power domain $\mathrm{P}(D)$ is defined by

$$\{d\} = \{d\}^*$$

for all $d \in D$. It can be shown that $\{\cdot\}_{D \to \mathrm{P}(D)}$ is continuous. For the function from $\mathrm{P}(D)$ to D, we choose the least upper bound function, $\bigsqcup_{\mathrm{P}(D) \to D}$, which is also continuous.[4]

We also define the abstraction function and the corresponding concretization function based on the Hoare power domains. Define function $Abs \in \mathrm{P}(\alpha) \to \mathrm{P}(\bar{\alpha})$ and function $Conc \in \mathrm{P}(\bar{\alpha}) \to \mathrm{P}(\alpha)$ by

$$\begin{aligned} Abs_{\mathrm{P}(\alpha) \to \mathrm{P}(\bar{\alpha})} &= \mathrm{P}(abs_{\alpha \to \bar{\alpha}}), \quad \text{and} \\ Conc_{\mathrm{P}(\bar{\alpha}) \to \mathrm{P}(\alpha)}(S) &= \bigcup \{T \mid T \in \mathrm{P}(\alpha), Abs_{\mathrm{P}(\alpha) \to \mathrm{P}(\bar{\alpha})}(T) \sqsubseteq_{\mathrm{P}(\bar{\alpha})} S\}. \end{aligned}$$

It can be shown that both $Abs_{\mathrm{P}(\alpha) \to \mathrm{P}(\bar{\alpha})}$ and $Conc_{\mathrm{P}(\bar{\alpha}) \to \mathrm{P}(\alpha)}$ are continuous if $abs_{\alpha \to \bar{\alpha}}$ is continuous.

Now, given the abstraction functions $abs_{\alpha \to \bar{\alpha}}$ and $abs_{\beta \to \bar{\beta}}$, we can define the abstraction function $abs_{(\alpha \to \beta) \to (\bar{\alpha} \to \bar{\beta})}$ inductively, using the Hoare power domain mechanism. The

[2] In category theory, P would be termed a functor.

[3] If we were to use the power set, instead of the Hoare power domain, as the power domain for the purpose of describing the semantic property we are interested in, the function $\mathrm{P}(f)$ would not be continuous even if f is continuous. Worse yet, the power set $P(D)$ will not preserve the ordering among the elements in a domain D. We would have, for example,

$$\mathrm{P}(\{\bot_{Nat}\}) = \{\bot_{Nat}\} \not\sqsubseteq \{7\} = \mathrm{P}(\{7\})$$

although $\bot_{Nat} \sqsubseteq_{Nat} 7$.

[4] The use of the least upper bound function implies that every element of domain $\mathrm{P}(D)$ is in fact a complete lattice. The proof is straightforward by the definition of the Hoare power domain.

set $\{\perp_{Nat}\}$ to the set $\{\perp_{Nat}\}$ in the standard interpretation. The use of power domains is essential in establishing the relationship between the standard semantics and abstract semantics.

We now provide an overview of Burn, Hankin, and Abramsky's work. We will use $\alpha, \beta, \ldots$ to refer to the the standard domains, and use $\bar{\alpha}, \bar{\beta}, \ldots$ to refer to their corresponding abstract domains. The objective in the development of Burn, Hankin, and Abramsky can be summarized as the following:

> Given an abstraction function $abs_{\alpha\to\bar{\alpha}}$ from a standard domain α to an abstract domain $\bar{\alpha}$ and an abstraction function $abs_{\beta\to\bar{\beta}}$ from a standard domain β to an abstract domain $\bar{\beta}$, how does one construct an abstraction mechanism $abs_{(\alpha\to\beta)\to(\bar{\alpha}\to\bar{\beta})}$ such that when applied to *any* function $f \in \alpha \to \beta$, its result, $abs_{(\alpha\to\beta)\to(\bar{\alpha}\to\bar{\beta})}(f)$, is an abstraction of f? Also, what is the relationship between f and $abs_{(\alpha\to\beta)\to(\bar{\alpha}\to\bar{\beta})}(f)$?

Note that, once the above question is answered, we then have a general mechanism for abstract interpretation of any typed higher–order functions.

Definition 2.2.16 Let D be a domain. A set $X \subseteq D$ is *Scott–closed* if

1. If $Y \subseteq X$ and Y is directed, then $\bigsqcup Y \in X$.
2. If $x \in X, y \sqsubseteq_D x$, then $y \in X$.

The least Scott–closed set containing a set Y is written as Y^*. □

We call a set $X \subseteq D$ is *left–closed* if it satisfies clause 2 of the above definition.

Definition 2.2.17 Let D be a domain. Define poset $\mathrm{P}(D)$ by

1. $\mathrm{P}(D) = \{S \mid \emptyset \neq S \subseteq D, S \text{ is Scott–closed }\}$, and $\perp_{\mathrm{P}(D)} = \{\perp_D\}$.
2. $s \sqsubseteq_{\mathrm{P}(D)} t$ if and only if $s \subseteq t$.

□

It can be shown that for any domain D, poset $\mathrm{P}(D)$ forms a domain. It is called the *Hoare power domain*. Not only can we apply the operator P to a domain, we can

perform an *abstract interpretation* of the program in some *abstract domain* that describes only the semantic property we are interested in. There are two important issues when performing an abstract interpretation upon a program to extract its semantic property: that the abstract interpretation must be correct with respect to the standard semantics, and that the execution of the abstract interpretation must terminate.

The execution of the abstract interpretation must terminate for all programs because it is part of the compilation process, and we expect all correct programs to be compiled successfully. We will describe in Section 2.3 the least fixed point finding procedure which is used in abstract interpretation to extract the desired semantic property, and prove the termination of this procedure. The correctness relationship between the standard interpretation and its abstract interpretation will be addressed in this section.

In this section, we will outline the framework developed by Burn, Hankin, and Abramsky [15] for abstract interpretation of typed higher–order functional programs based on Hoare power domains. Burn, Hankin, and Abramsky's initial concern is the strictness property of functional programs, but the framework can be applied easily to other abstract semantics as well. Before we start, we point out that the framework of Burn, Hankin, and Abramsky is not the only approach to abstract interpretation of functional programs. For example, Hudak and Young [39,40,81] have developed a method for abstract interpretation of untyped higher–order functional programs, Mycroft and Jones [53] have proposed a relational framework, while Nielson and Nielson [55,56,57] have used a two–level metalanguage to describe the standard semantics, the abstract semantics, and their relationship in the same setting.

However, the framework of Burn, Hankin, and Abramsky is appealing for several reasons. One is that the use of type information in their development gives a clear path to the abstraction of higher–order functions. The use of the Hoare power domain as an abstraction mechanism also results in a clean relationship between the standard semantics and the abstract semantics. Note that a semantic property, such as termination and strictness, often denotes membership in a subset of the standard domain. For example, in strictness analysis, a defined element in domain *Nat* is mapped to the element $\top_{\mathcal{2}}$ in domain $\mathcal{2}$. The element $\top_{\mathcal{2}}$ can be thought of denoting the subset $Nat - \{\bot_{Nat}\}$ of domain *Nat*, while the element $\bot_{Nat}$ denoting the subset $\{\bot_{Nat}\}$. A strict function over natural numbers, whose abstraction will map $\bot_{\mathcal{2}}$ to to $\bot_{\mathcal{2}}$ in the abstract domains, simply maps all elements in the

Further studies of denotational semantics can be found in the work of Scott [69,70], and in the books of Stoy [72], Schmidt [66], and Gunter [31]. A method to find the least fixed point of a recursive domain equation can be found in the work of Smyth and Plotkin [71].

2.2 Abstract Interpretations of Functional Programs

A *standard* interpretation gives a program its intended denotational semantics. For example, syntactic literals for natural numbers are interpreted as elements in the semantic domain *Nat*, and the equality operator `=` is interpreted as the semantic function *eq* as described in Figure 2.1. An implementation of a program is correct if it yields a result described by the standard semantics. However, we can also interpret the same program in other ways. Suppose that we want to extract a certain semantic property of a program to exploit a possible optimization opportunity in its implementation. We then can interpret the programs according only to the semantic property we are interested in, not according to the intended standard semantics. Such an interpretation is called a *non–standard* interpretation.

For example, when implementing a program `f` in a non–strict functional language, we are interested in whether or not the standard semantics of `f`, called *f*, is strict or not. Recall that a function *f* is strict if it does not terminate whenever it is applied to non–terminating arguments (Definition 2.1.9). If *f* is strict in all of its arguments, then we can implement `f` by evaluating all its arguments before *f* is called. This is more efficient than the usual implementation, where the evaluation of arguments have to be delayed until they are actually needed, but yields identical results. Because we are only interested in whether a computation is terminating or not, we can interpret syntactic terms for natural numbers as elements of the semantic domain $\mathcal{2}$, with element $\top_{\mathcal{2}} \in \mathcal{2}$ representing terminating computations and $\bot_{\mathcal{2}}$ for non–terminating computations. Also, the operator `=` will be interpreted as a monotonic function in the semantic domain $\mathcal{2} \to \mathcal{2} \to \mathcal{2}$, describing only the termination property of the equality test.

When optimizing an implementation of a functional program, the non–standard semantics we are interested in usually is an abstraction of the standard semantics.[1] That is, we

[1] The non–standard semantics we are interested in can be abstraction of other non–standard semantics as well. The emphasis here is that it is an abstraction of some existing semantics.

described in Figure 2.1. Because function `factorial` is recursively defined, its semantics is interpreted as the least fixed point of the functional F, where

$$F \equiv \lambda f \,.\, \lambda x \,.\, \mathit{if}\ (\mathit{eq}\ x\ 0)\ 1\ (\mathit{mult}\ x\ (f\ (\mathit{minus}\ x\ 1)))).$$

Functional F is an element in the domain $(Nat \to Nat) \to (Nat \to Nat)$.

To have a better idea of what the least fixed point of F looks like, we can construct a directed set $X_F \subseteq Nat \to Nat$ such that

$$X_F = \{\bot_{Nat\to Nat},\ \ F(\bot_{Nat\to Nat}),\ \ F(F(\bot_{Nat\to Nat})),\ \ \ldots,\ \ F^n(\bot_{Nat\to Nat}),\ \ \ldots\},$$

and compute the least upper bound of X_F, which is the semantics of program `factorial`.

Because $\bot_{Nat\to Nat} = \lambda\, x_{Nat} \,.\, \bot_{Nat}$, after some calculations, we get

$$\begin{aligned} X_F = \{\ & \lambda x \,.\, \bot, \\ & \lambda x \,.\, \mathit{if}\ (\mathit{eq}\ x\ 0)\ 1\ \bot, \\ & \lambda x \,.\, \mathit{if}\ (\mathit{eq}\ x\ 0)\ 1\ (\mathit{if}\ (\mathit{eq}\ x\ 1)\ 1\ \bot), \\ & \lambda x \,.\, \mathit{if}\ (\mathit{eq}\ x\ 0)\ 1\ (\mathit{if}\ (\mathit{eq}\ x\ 1)\ 1\ (\mathit{if}\ (\mathit{eq}\ x\ 2)\ 2\ \bot)), \\ & \ldots \\ & \lambda x \,.\, \mathit{if}\ (\mathit{eq}\ x\ 0)\ 1\ (\mathit{if}\ (\mathit{eq}\ x\ 1)\ 1\ (\mathit{if}\ (\mathit{eq}\ x\ 2)\ 2\ (\ldots \\ & \quad (\mathit{if}\ (\mathit{eq}\ x\ (n-1))\ ((n-1)*(n-2)*\ldots*1)\ \bot)\ldots))), \\ & \ldots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \} \end{aligned}$$

Program `factorial` then has semantics $\bigsqcup X_F$. Notice that X_F is an infinite chain, in which the least defined element in the chain gives no information about the semantics of `factorial`. However, $F^n(\bot_{Nat\to Nat})$, the $(n+1)$–th element in the chain, correctly describes the semantics of `factorial` for arguments $0, 1, \ldots, n-1$ but gives no information for arguments n and above. It is clear that $\bigsqcup X_F$, the limit of the chain X_F, does correspond to our intuitive understanding of the semantics of the program `factorial`. □

Note that in the above example the semantics of a recursively defined function is explained by only using mathematical constructions. There is no abstract machine execution, no self–reference pointers, no stacks, and no heaps. One of the merit of denotational semantics is that we can reason about recursive programs without resorted to use a particular machinery for implementing recursion.

$$
\begin{array}{llllll}
\mathit{if} & \bot_{Bool} & x_{Nat} & y_{Nat} & = & \bot_{Nat}, \\
\mathit{if} & \mathrm{true}_{Bool} & x_{Nat} & y_{Nat} & = & x_{Nat}, \\
\mathit{if} & \mathrm{false}_{Bool} & x_{Nat} & y_{Nat} & = & y_{Nat}; \\
 & & & & & \\
\mathit{eq} & \bot_{Nat} & y_{Nat} & & = & \bot_{Bool}, \\
\mathit{eq} & x_{Nat} & \bot_{Nat} & & = & \bot_{Bool}, \\
\mathit{eq} & x_{Nat} & x_{Nat} & & = & \mathrm{true}_{Bool}, \quad \text{where } x_{Nat} \neq \bot_{Nat} \\
\mathit{eq} & x_{Nat} & y_{Nat} & & = & \mathrm{false}_{Bool}, \quad \text{where } x_{Nat} \neq \bot_{Nat}, y_{Nat} \neq \bot_{Nat}, \\
 & & & & & \quad \text{and } x_{Nat} \neq y_{Nat}; \\
 & & & & & \\
\mathit{mult} & \bot_{Nat} & y_{Nat} & & = & \bot_{Nat}, \\
\mathit{mult} & x_{Nat} & \bot_{Nat} & & = & \bot_{Nat}, \\
\mathit{mult} & x_{Nat} & y_{Nat} & & = & x_{Nat} * y_{Nat}, \quad \text{otherwise}; \\
 & & & & & \\
\mathit{minus} & \bot_{Nat} & y_{Nat} & & = & \bot_{Nat}, \\
\mathit{minus} & x_{Nat} & \bot_{Nat} & & = & \bot_{Nat}, \\
\mathit{minus} & x_{Nat} & y_{Nat} & & = & 0_{Nat}, \quad \text{where } x_{Nat} \neq \bot_{Nat}, y_{Nat} \neq \bot_{Nat}, \\
 & & & & & \quad \text{and } x_{Nat} \leq y_{Nat}; \\
\mathit{minus} & x_{Nat} & y_{Nat} & & = & x_{Nat} - y_{Nat}, \quad \text{otherwise}.
\end{array}
$$

Figure 2.1: Denotational semantics of some predefined terms of a functional language. Note. The *if* function is the denotational semantics of the (natural number valued) `if ... then ...else ...`expression. Function *eq* is for the natural number equality operator `=`, *mult* is for natural number multiplication `*`, and *minus* is for natural number subtraction `-`. All of the four functions are continuous. While function *eq*, *mult*, and *minus* are strict in both of their arguments, function *if* is strict only in its first argument and is not necessary strict in the next two arguments. An `if` expression will not necessarily evaluate its branch expressions (and if it indeed evaluate them, it only evaluates one of its two branches). Note that this interpretation of the `if` expression is used both in strict and non–strict functional languages.

```
factorial n  =  if n = 0 then 1 else n * (factorial (n - 1))
```

Figure 2.2: A functional program `factorial` that computes the factorial number of its input argument.
Note. The denotational semantics for `factorial` is described in Example 2.1.15.

We then can define a functional F, which is not recursive, that takes an $f \in D \to D'$ and an $x \in D$ as arguments, but whose body is the same as the body of f,

$$F = \lambda\, f_{D \to D'}\, .\, \lambda\, x_D \ldots . \ldots f \ldots$$

In the above definition of F, the appearance of f is not recursive any more, and F is in the domain $(D \to D') \to (D \to D')$ if f is in the domain $D \to D'$. We then treat the least fixed point of F, which is unique, as the semantics of f.

Domains are semantic images of syntactic terms. For example, terms that represent values in the "disjoint union" of data types D and D' can be described by elements in their coalesced sum domain $D \oplus D'$ or in their sum domain $D + D'$, depending whether or not the union is strict with respect to its components D and D'. Variant records in Pascal, unions in C, or tag–discriminated datatypes in ML can all be described by coalesced sum domains, while tag–discriminated data in the non–strict functional language Haskell are better described as elements in sum domains. Product domains and smashed product domains can apply to similar situations. Continuous function spaces model functions in a non–strict language, while strict continuous function spaces (where functions are both continuous and strict) model functions in a strict language. A recursively defined data type, such as the list data type in functional languages, is described by a recursively defined domain equation, whose least fixed point solution is treated as the semantic domain of the data type.

How will a functional program be given its denotational semantics? First, we interpret predefined terms in the language as elements in their associated semantic domains. For example, the syntactic literal `0` is interpreted as the semantic element 0 in the domain *Nat* for natural number, and the predefined operator `=` for equality between two natural numbers is interpreted as the continuous function *eq* in the semantic domain $Nat \to (Nat \to Bool)$. The functionality of *eq* and some other predefined terms of a functional language are described in Figure 2.1. For a term defining a function, the semantics of the term is then interpreted as a continuous function from its arguments, which belong to their associated semantic domains, to the semantics of the function body, which is interpreted by composing the semantics of the subterms in the body.

Example 2.1.15 What is the semantics of the function `factorial` in Figure 2.2? The semantics of the terms `0`, `1`, `=`, `*`, and `-`, and of the expression `if ... then ... else` have been

1. Every function $f \in D \to D$ has a least fixed point.
2. Moreover, there exists a function $\mathit{fix} \in (D \to D) \to D$ such that for every function $f \in D \to D$, $\mathit{fix}(f)$ is the least fixed point of f.

□

PROOF OUTLINE.

1. For a given function f, define a directed set $X_f \subseteq D$ by

$$X_f = \{\bot_D,\ f(\bot_D),\ f(f(\bot_D)),\ \ldots,\ f^{(n)}(\bot_D),\ \ldots\}.$$

   Then by the continuity of f,

$$f(\bigsqcup X_f) = \bigsqcup f\{X_f\} = \bigsqcup X_f.$$

   That is, $\bigsqcup X_f$ is a fixed point of f. Suppose that d is a fixed of f as well. Then,

$$\bot_D \sqsubseteq_D d,\ \ f(\bot_D) \sqsubseteq_D f(d) = d,\ \ \ldots\ \ f^{(n)}(\bot_D) \sqsubseteq_D f^{(n)}(d) = d,\ \ \ldots$$

   Taking the least upper bound of both sides, it follows that $\bigsqcup X_f$ is the least fixed point.

2. Define function fix by

$$\mathit{fix} = \lambda f_{D\to D}.\bigsqcup X_f.$$

   Then for any function $f \in D \to D$, $\mathit{fix}(f)$ is its least fixed point. By rearranging indices, we can show that

$$\mathit{fix}(\bigsqcup F) = \bigsqcup \mathit{fix}\{F\}$$

   for all directed set $F \subseteq D \to D$. That is, function fix is continuous, and $\mathit{fix} \in (D \to D) \to D$.

◇

The unique existence of the least fixed point of a continuous function is used in denotational semantics as the mechanism to deal with recursively defined functional programs. A function f in a program is recursively defined if f appears in the expression defining f, as in the following:

$$f = \lambda x_D \ldots . \ldots f \ldots$$

What are the motivations behind using continuous functions spaces as the semantic domains of functions? First of all, we shall only accept functions which are monotonic with respect to information content. Intuitively, if an element x contains less information than element y, then the result of the function application $f(x)$ shall yield less information than $f(y)$, regardless of what function f is. Secondly, if a directed set X is viewed as an approximation (Definition 2.1.2, clause 2 (b)), then applying a function f to the least upper bound of X shall be the same as applying f to all elements of X then calculating their least upper bound. That is, a function f shall be understood as an approximation too. Put it in another way, we do not want to admit functions that jump arbitrarily at the limit of an approximation. In addition, continuous function spaces fit in the cpo framework very well because there always exists a lub for a directed set of continuous functions.

Example 2.1.12 The domain $\mathit{2} \to \mathit{2}$ has three elements. These are $\bot_{\mathit{2}\to\mathit{2}}$, $\mathrm{id}_{\mathit{2}\to\mathit{2}}$, and $\top_{\mathit{2}\to\mathit{2}}$, where

$$\begin{aligned} \bot_{\mathit{2}\to\mathit{2}} &= \lambda x_{\mathit{2}} . \bot_{\mathit{2}}, \\ \mathrm{id}_{\mathit{2}\to\mathit{2}} &= \lambda x_{\mathit{2}} . x_{\mathit{2}}, \quad \text{and} \\ \top_{\mathit{2}\to\mathit{2}} &= \lambda x_{\mathit{2}} . \top_{\mathit{2}} \end{aligned}$$

and

$$\bot_{\mathit{2}\to\mathit{2}} \quad \sqsubseteq_{\mathit{2}\to\mathit{2}} \quad \mathrm{id}_{\mathit{2}\to\mathit{2}} \quad \sqsubseteq_{\mathit{2}\to\mathit{2}} \quad \top_{\mathit{2}\to\mathit{2}}.$$

□

Note that the function that maps $\bot_{\mathit{2}}$ to $\top_{\mathit{2}}$ and maps $\top_{\mathit{2}}$ to $\bot_{\mathit{2}}$ is not an element of domain $\mathit{2} \to \mathit{2}$ because it is not monotonic.

Definition 2.1.13 Let D be a poset and f a total function from D to D.

1. $x \in D$ is a *fixed point* of f if and only if $f(x) = x$.
2. x is the *least fixed point* of f if and only if x is a fixed point of f and for every fixed point $d \in D$ of $f, x \sqsubseteq_D d$.

□

Theorem 2.1.14 Let D be a domain.

Definition 2.1.10 Let D and D' be domains. Define $D \to D'$ by

1. $D \to D' = \{f \mid f \text{ is a continuous function from } D \text{ to } D'\}$, and $\bot_{D\to D'} = \lambda\, d_D\,.\,\bot_{D'}$.
2. $f \sqsubseteq_{D\to D'} g$ if and only if for all $d \in D, f(d) \sqsubseteq_{D'} g(d)$.

□

The above definition leads to the following well–known theorem.

Theorem 2.1.11 The continuous function space $D \to D'$ is a domain if both D and D' are. □

PROOF. It is clear that $\bot_{D\to D'} \sqsubseteq_{D\to D'} f$ for every function f in the poset $D \to D'$. To prove that the poset $D \to D'$ is a domain, it remain to show that

- every directed set $F \subseteq D \to D'$ has a least upper bound, and
- the least upper bound is in $D \to D'$; that is, itself is a continuous function.

Define a function $F^{\sqcup}$ for a directed set F by

$$F^{\sqcup} = \lambda\, d_D\,.\, \bigsqcup_{f\in F} \{f(d)\}.$$

Since F is a directed set, we know that $\{f(d) \mid f \in F\} \subseteq D'$ is a directed set for any $d \in D$. Because D' is a domain, it follows that $\bigsqcup_{f\in F}\{f(d)\}$ exists, $F^{\sqcup}$ is a well defined function, and $F^{\sqcup}$ is the least upper bound of F.

But is $F^{\sqcup}$ a continuous function? The answer is yes. For all directed set $X \subseteq D$,

$$\begin{array}{lll}
 & F^{\sqcup}(\bigsqcup X) & \\
= & \bigsqcup_{f\in F}\{f(\bigsqcup X)\} & \text{(Definition of } F^{\sqcup}) \\
= & \bigsqcup_{f\in F}\{\bigsqcup_{x\in X}\{f(x)\}\} & (X \subseteq D \text{ is directed, and each } f \in F \text{ is continuous}) \\
= & \bigsqcup_{x\in X}\{\bigsqcup_{f\in F}\{f(x)\}\} & \text{(Rearrange indices)} \\
= & \bigsqcup_{x\in X}\{F^{\sqcup}(x)\} & \text{(Definition of } F^{\sqcup}) \\
= & \bigsqcup F^{\sqcup}\{X\}. & \text{(Definition of } \bigsqcup X)
\end{array}$$

This shows that $F^{\sqcup}$ is continuous. From now now, we will simply write $\bigsqcup F$ to denote $F^{\sqcup}$, the least upper bound of a directed set $F \subseteq D \to D'$. ◇

Example 2.1.8 Let domain *Bool* be defined as $\mathit{2} \oplus \mathit{2}$. We write the three elements in *Bool* as $\bot_{Bool}, \text{true}_{Bool}$, and false_{Bool}, with $\bot_{Bool} \sqsubseteq_{Bool} \text{true}_{Bool}$ and $\bot_{Bool} \sqsubseteq_{Bool} \text{false}_{Bool}$, but neither $\text{true}_{Bool} \sqsubseteq_{Bool} \text{false}_{Bool}$ nor $\text{false}_{Bool} \sqsubseteq_{Bool} \text{true}_{Bool}$. We usually call *Bool* the Boolean domain. □

As we can see, the subscripts for domains can be a little tedious. We will omit a subscript whenever its meaning is clear from the context. At first sight, it might seem odd that there is no ordering between true_{Bool} and false_{Bool}. However, knowing that an element x in *Bool* is true_{Bool} gives no more information than knowing that x is false_{Bool}, and vice versa. What really matters is knowing that x is, or is not, more informative than $\bot_{Bool}$.

It is worth knowing that in denotational semantics, we often lift a discrete set S, where the elements are not related to each other by a $\sqsubseteq$ relationship, to form a semantic domain lift(S). Element $\bot_S$ in domain lift(S) represents a divergent computation of an element of S, and it is semantically weaker than any element in lift(S). The domain *Bool* above is an example of this. In fact, *Bool* can equivalently be defined by lift($\mathbf{B}$), where $\mathbf{B} = \{\text{true}, \text{false}\}$. Another example is the domain of natural numbers, *Nat*, which is defined by Nat = lift($\mathbf{N}$), where $\mathbf{N} = \{0, 1, 2, \ldots\}$. Elements $0, 1, 2, \ldots$ in *Nat* are not related to one another by the $\sqsubseteq_{Nat}$ relationship. Rather, $\bot_{Nat} \sqsubseteq_{Nat} n$ for every element $n \in \mathit{Nat}$.

Before we define $D \to D'$, the continuous function space from domain D to domain D', let us define a few properties on functions.

Definition 2.1.9 Let D and D' be domains, and f be a total function from D to D'.

1. f is *monotonic* if and only if $f(d_1) \sqsubseteq_{D'} f(d_2)$ whenever $d_1 \sqsubseteq_D d_2$.
2. f is *continuous* if and only if $f(\bigsqcup X) = \bigsqcup f\{X\}$ for every directed set $X \subseteq D$, where $f\{X\}$ is defined as $f\{X\} = \{f(x) \mid x \in X\}$.
3. f is *strict* if and only if $f(\bot_D) = \bot_{D'}$.

□

If a function f is not strict, it is called *non–strict*. If a function is continuous then it is monotonic, but the reverse is not true.

It is easy to verify that lift(D) is a cpo if D is. We call lift(D) the *lifted* domain of D.

Example 2.1.6 Let domain *2* be defined as lift(*1*), and let the two elements in domain *2* be noted as $\bot_{\mathit{2}}$ and $\top_{\mathit{2}}$. We define $\bot_{\mathit{2}} \sqsubseteq_{\mathit{2}} \top_{\mathit{2}}$. □

Domain *1* and domain *2* are very useful in the semantic analysis of functional programs. We will see them later in this chapter and the following chapters. Given two domains D and D', we can make the following constructions.

Definition 2.1.7 Let D and D' be domains.

Define $D + D'$, their sum, by

1. $D + D' = D \cup D' \cup \{\bot_{D+D'}\}$, where the elements in D are made distinct from the elements in D', and $\bot_{D+D'} \notin D \cup D'$.
2. $x \sqsubseteq_{D+D'} y$ if and only if $x = \bot_{D+D'}$ or $x \sqsubseteq_D y$ or $x \sqsubseteq_{D'} y$.

Define $D \oplus D'$, their *coalesced sum*, by

1. $D \oplus D' = D \cup D' \cup \{\bot_{D\oplus D'}\}$, where the elements in D are made distinct from the elements in D', except $\bot_D$ and $\bot_{D'}$. Let $\bot_{D\oplus D'} = \bot_D = \bot_{D'}$.
2. $x \sqsubseteq_{D+D'} y$ if and only if $x = \bot_{D\oplus D'}$ or $x \sqsubseteq_D y$ or $x \sqsubseteq_{D'} y$.

Defined $D \times D'$, their *product*, by

1. $D \times D' = \{ \langle d, d' \rangle \mid d \in D,\ d' \in D' \}$, and $\bot_{D\times D'} = \langle \bot_D, \bot_{D'} \rangle$.
2. $\langle d_1, d_1' \rangle \sqsubseteq_{D\times D'} \langle d_2, d_2' \rangle$ if and only if $d_1 \sqsubseteq_D d_2$ and $d_1' \sqsubseteq_{D'} d_2'$.

Define $D \otimes D'$, their *smashed product*, by

1. $D \otimes D' = \{ \langle d, d' \rangle \mid d \in D,\ d' \in D' \}$, and $\bot_{D\otimes D'} = \langle \bot_D, d' \rangle = \langle d, \bot_{D'} \rangle$ for any $d \in D$ and $d' \in D'$.
2. $\langle d_1, d_1' \rangle \sqsubseteq_{D\otimes D'} \langle d_2, d_2' \rangle$ if and only if $\langle d_1, d_1' \rangle = \bot_{D\otimes D'}$, or $d_1 \sqsubseteq_D d_2$ and $d_1' \sqsubseteq_{D'} d_2'$.

□

It can easily be shown that $D + D'$, $D \oplus D'$, $D \times D'$, and $D \otimes D'$ are domains, if both D and D' are.

(b) For all $x, y \in X$ there is a $z \in X$ such that $x \sqsubseteq_D z$ and $y \sqsubseteq_D z$.

2. D is a *complete partially ordered set* (cpo) if

   (a) There is a least element $\bot_D \in D$ such that for all $x \in D$, $\bot_D \sqsubseteq_D x$.

   (b) Every directed set $X \subseteq D$ has a *least upper bound* (lub) $\bigsqcup X \in D$.

□

Remark 2.1.3 A *complete lattice* is a poset D where *every* subset $X \subseteq D$ has a least upper bound. Each complete lattice D is a cpo since $\bot_D = \bigsqcup \emptyset$. □

The relationship $\sqsubseteq_D$, which we will later call "semantically weaker", of a domain D is used to describe how two elements in D are related to each other in term of computational progress. We will interpret $x \sqsubseteq_D y$ as x is less defined than y computationally. That is, y has more information content than x does (Definition 2.1.1). Because a computation may not terminate, which gives no information at all, there is always a least defined element $\bot_D$ for every semantic domain D (Definition 2.1.2, clause 2 (a)). A directed set X of domain D may be viewed as an approximation for some computation in D. A directed set X is an approximation because for every two elements $x, y \in X$ there is always a more defined element $z \in X$ which both x and y can progress to (Definition 2.1.2, clause 1 (b)). For such an approximation $X \subseteq D$, we insist that at least all the computations in X progress to an unique element, their least upper bound, in domain D; though the least upper bound is not necessarily in X itself (Definition 2.1.2, clause 2 (b)). The terms cpo and domain will be used interchangeably from now on.

Example 2.1.4 Let *1* be the poset of only one element $\bot_1$, where $\bot_1 \sqsubseteq_1 \bot_1$. Then *1* is a domain. □

We can also construct domains from other existing domains. The most simple construction is the following.

Definition 2.1.5 Let D be a domain. Define poset $\mathrm{lift}(D)$ by

1. $\mathrm{lift}(D) = D \cup \{\bot_{\mathrm{lift}(D)}\}$, where $\bot_{\mathrm{lift}(D)} \notin D$.
2. $x \sqsubseteq_{\mathrm{lift}(D)} y$ if and only if $x = \bot_{\mathrm{lift}(D)}$ or $x \sqsubseteq_D y$.

□

domain has an algebraic structure. The composition functions over various semantic domains also have certain mathematical properties, such that the semantics of an expression can be easily understood by applying the composition function to the semantics of its sub–expressions. It is important to note that the denotational semantics of a program does not describe how the program should be implemented, rather, it only describes what the program means.

Perhaps the motivation for a denotational semantics of a programming language is better understood by contrasting it to an operational semantics of the same language. An operational semantics of a programming language uses an "abstract machine", which has a state and set of instructions that modify the state, to describe how would a program behave on the machine. Syntactic terms are translated into machine instructions and executed by the abstract machine. The meaning of a syntactic term is described by the effect of carrying out the translated instructions upon a machine.

Although operational semantics is valuable when implementing a program, it is less desirable when reasoning about it. To understand a program's operational semantics, we have to carry out its execution, which may be tedious and erroneous. Furthermore, it is difficult to compare a class of related programs because we first have to execute them separately and then compare the results. However, the denotational semantics of a program can be understood solely by the semantics denoted by its parts; and a class of related programs can be compared by concentrating on (the semantics of) the parts where they differ.

The semantic value denoted by a syntactic term belongs to a semantic domain. Most often these semantic domains are complete partially ordered sets. We will describe the motivation behind complete partially ordered sets shortly. Let us start with the definitions first.

Definition 2.1.1 A *partially ordered set* (poset) D is a set with a binary relation $\sqsubseteq_D \subseteq D \times D$ such that $\sqsubseteq_D$ is reflexive, asymmetric, and transitive. □

Definition 2.1.2 Let D be a poset.

1. A set $X \subseteq D$ is *directed* if
   (a) $X \neq \emptyset$.

Chapter 2

Semantics–Based Analyses and Least Fixed Points

In this chapter, we will describe some well–established methods and results in the field of programming language semantics, with emphases on functional programs and their analyses. In particular, we will outline the development and motivation of denotational semantics, the mechanism of performing semantic analyses by abstract interpretation, and the methods for finding least fixed points over abstract domains. The purpose here is to give a background for the results in Chapter 3, in which we develop a syntactic method for finding least fixed points over finite domains. The writing in this chapter is somewhat informal, with emphases on the motivation and intuition behind the development. It is not a detailed exposition of the subjects. Some technical results, however, will be introduced and proved if they are necessary for the understanding of the subjects. We will also cite literature where a detailed formulation of the outlined results can be found.

2.1 Denotational Semantics — An Overview

The denotational semantics of a programming language gives a "semantic valuation function" to map syntactic terms of the language to the semantic values which they denote. These valuation functions are usually recursively defined, and are compositional in the sense that the semantic value of a term is composed of the semantic values denoted by its syntactic subterms. Semantic values are grouped into semantics domains, where each

of multihead Turing machines. We also describe some "surprising" consequences of the implementation. For example, a list of arbitrary number of elements can now be reversed in constant real–time in a purely functional language.

We consider in Chapter 7 some possible directions of future work. This chapter concludes the dissertation.

and Chapter 4 are of introductory nature, in which general backgrounds for subsequent technical developments are reviewed. Original contributions are in Chapters 3, 5, and 6. The outlines of the seven chapters are as the following.

Chapter 1, the one you are currently reading, includes the objectives of this dissertation and their justifications. Functional programming style and some associated terminologies are briefly introduced.

Chapter 2 provides a short overview of denotational semantics, a framework of abstract interpretation of typed functional programs, and the role of least fixed point finding. The frontier method, which is a semantics–based approach, for least fixed point finding is described. Its advantages and disadvantages are discussed.

Chapter 3 describes a syntactic approach to least fixed point finding over finite domains. Proofs and examples are provided in the development of the syntactic method, so are the intuitions behind the methods and related implementation issues. A large example involving strictness analysis over non–flat domains is included.

In Chapter 4, we review three popular approaches to the aggregate update problem in functional languages. The first approach uses a compile–time analysis or a run–time check to detect aggregates whose previous versions will never be needed in subsequent computation, hence allows update operations to perform efficient (and safe) in–place modifications. The second approach uses language restrictions or programming styles to make sure that, for programs written under the restrictions or styles, only the newest version of an aggregate is accessible; hence all update operations can be performed in–place. Another general approach is to design efficient run–time data structures to maintain multiple versions of an aggregate. The advantages and disadvantages of the three approaches are discussed.

An efficient implementation of fully persistent arrays is described in Chapter 5. The implementation use a fragmented shallow binding scheme, an extension of Baker's shallow binding technique. It is shown that, using the proposed implementation and some locality assumptions, each update operation costs constant worst–case time, and each read operation in a sufficiently long sequence of reads costs only constant amortized time. Empirical results are collected to measure the effectiveness of the implementation scheme.

Chapter 6 describes a purely functional implementation of double–ended queues such that every pop and push operation to either end of any version of a deque costs only constant real–time. We show that this results in a purely functional, real–time simulation

restrictions on the language to coordinate multiple accesses to a single aggregate. We concentrate in this dissertation on techniques that make aggregate data structures fully persistent, that is, any version of an aggregate can be read and updated efficiently.

Standard techniques for making data structures fully persistent are well established [23]. However, they mostly deal with linked data structures (with bounded in–degree for each node) and use the side–effect features of an imperative language to achieve much of the efficiency. We develop efficient implementations for functional aggregates where previous techniques for making aggregates fully persistent do not apply. More specifically, we will extend the shallow binding technique of Baker [6,7] to implement functional arrays such that each update operation costs only constant time in the worst case and, assuming some common reference locality, each read operation in a sufficiently long sequence costs only constant amortized time. The previous standard technique for full persistence does not apply because arrays are not linked data structures. We will use side–effects in the above–mentioned implementation of functional arrays. We then investigate the possibility of implementing efficient aggregate data structures in a purely functional way. We will show that deques can be implemented efficiently in a purely functional way such that every pop and push operation can be performed on either end of any version of a deque in constant time. This extends results of Hood and Melville [34] and Hoogerwoord [35], refines a design of Sarnak [64], and leads to several interesting applications. Notice that the standard technique for full persistence will require an imperative language and does not achieve constant time in the worst case. Although these are just two case studies, the techniques we use shared some common characteristics. We hope that they can shed some light to a general approach to the aggregate update problem.

1.3 Outline of the Dissertation

This dissertation consists of seven chapters, of which the first chapter is an introduction and last chapter is some thought on future work. The remaining chapters can be partitioned into two groups. The first group, Chapter 2 and Chapter 3, contains materials on semantic analysis of functional programs and a syntactic approach to least fixed point finding over finite domains. Chapters 4, 5, and 6 constitute the second group which deals with efficient implementations of aggregate data structures for functional programs. Chapter 2

1.2 Objectives of the Dissertation

This dissertation seeks to resolve two optimization issues in the analysis and implementation of functional programs. The first goal is to investigate a *syntactic* approach, contrary to the usual semantic approach, of finding the least fixed points of higher–order functions over finite domains. The second objective is to develop implementation techniques for aggregate data structures for functional programs such that accesses to the aggregates are both efficient and side–effect free, without the help of any compile–time analysis.

Finding the least fixed point of a monotonic function over a finite domain is an essential task when extracting some specific property, such as strictness, of a functional program. In the framework of abstract interpretation, a functional program is interpreted over an abstract domain, with the finite number of elements in the abstract domain corresponding to some specific program properties. Evaluating the least fixed point of a monotonic function over a finite domain corresponds to inferring the abstract property of a recursive defined functional program. The least fixed point will describe the property of the program being analyzed, and will be used to optimize the implementation of the program accordingly.

The usual semantic methods for least fixed point finding often traverse large portions of an abstract domain to search for the least fixed point, and the functional program being analyzed is used to check whether the search has converged or not [19,60]. A syntactic method, on the other hand, will better utilize the program text at hand and will literally use it in the approximation iteration to help seeking the least fixed point. Because a semantic element can be denoted by two different syntactic terms such that the two terms cannot be compared syntactically for their equality, a naive syntactic method will fail to converge in the approximation iteration. We will show that, for finite domains, there is a syntactic approach for least fixed point finding that is both sound and complete with respect to the semantics. Moreover, we will show that the proposed syntactic method can be quite effective in cases where the usual semantic method is very inefficient.

Efficient implementations of aggregate data structures, particularly arrays, for functional programs has been an active research topic. The problem arises because once an aggregate is updated in a functional program, both the old version and newly updated copy must be preserved to maintain the side–effect free semantics of the functional language. There have been many approaches to this problem, some based on compile–time analysis or run–time checking to eliminate unnecessary updates and duplications, and some using

body. In a lazy implementation, once an actual parameter expression is evaluated, the corresponding formal parameter name in the body will refer to the evaluated value, not to the original expression.

As shown above, a lazy implementation of a non–strict functional language has the best of both worlds. It avoids evaluating an argument too early at the function invocation time, but also avoids repeated evaluations of the same argument in the function body. Hence, a non–strict functional language and its lazy implementation can often relieve a programmer of many unnecessary concerns over the evaluation order of some components in a large task. Simply put, we can say that, in a lazy implementation of a non–strict functional language, if a component is needed in the final output, it will be evaluated as needed and the result of the evaluation will be remembered. Otherwise, the component will not be evaluated.

However, a lazy implementation does incur some overhead in delaying the evaluation of function arguments from the function invocation time to the time when the values of the arguments are actually needed. This situation is also true in non–lazy implementations of a non–strict language. A *thunk* — a closure with sufficient variable/value binding information — is usually needed to represent the delayed arguments. An optimization opportunity is to detect places in a non–strict functional program where lazy evaluation can be replaced by strict evaluation, without affecting the semantics of the program. For example, in the above function `f`, it can be determined that the function arguments `p` and `x` are always needed when computing the result of the function application. (The argument `y`, on the other hand, is not always needed.) It does not matter if we compute it earlier, at the function invocation time, or later in the function body. If they result in non–termination or program abortion when they are evaluated "too early" by the called–by–value method, they will still result in non–termination or abortion later when they are evaluated by the called–by–name method. An analysis for non–strict functional programs which detects places where called–by–name or call–by–need parameter passing can be safely converted into called–by–value parameter passing is called *strictness analysis*. Such an analysis can help a compiler produce an optimized implementation of a non–strict functional program.

be divided into two groups. A functional language is *strict* if arguments to a function are always evaluated before the function body is executed. A language is *non–strict* if the function arguments are evaluated only if their values are actually required by the computation. The term *call–by–value* is often used to describe the way a strict functional program evaluates its function arguments, while the term *call–by–name* is used to refer the cases in a non–strict functional program. An implementation of a non–strict functional language is called *lazy* if a function argument is evaluated at most once — if it is ever evaluated — in the function body, although the argument may appear more than once in the body. This is also known as *called–by–need* evaluation.

As an example, let us use the following toy program to illustrate the above terms.

```
f p x y  =  if p then x*x else x*y
```

Function `f` accepts three arguments, `p`, `x`, and `y`, and returns the value of `x*x` or `x*y` depending on whether `p` is true or false. Notice that the phrase `if ...then ...else ...` is a language primitive that only evaluates one of its two branch expressions, depending on the outcome of the predicate condition. We can say that the primitive `if ...then ...else ...` is strict in its predicate condition, but non–strict in both of its two branches. This interpretation of `if ...then ...else ...` is used both in strict and non–strict functional languages.

Let us assume that evaluating the expression `(1 div 0)` results in an aborted computation. What would the expression

```
f true (2 + 3) (1 div 0)
```

compute in a strict language? What would be the result in a non–strict language, and in a lazy implementation? In a strict language, the computation is aborted because `(1 div 0)` is evaluated upon the function invocation, although it is not needed when evaluating the `if ...then ...else ...` primitive in the function body. In a non–strict language, the expression will be "correctly" evaluated as

```
if true then (2 + 3)*(2 + 3) else (2 + 3)*(1 div 0)
```

which evaluates `(2 + 3)*(2 + 3)`. It evaluates to `25` by executing the primitive `*`, which computes `(2 + 3)` *twice*. However, in a lazy implementation of a non–strict language, the argument `(2 + 3)` will only be computed once, although it is used twice in the function

Chapter 1

Introduction

Functional programming languages provide programmers with clean semantics and side–effect free computation, which make easier the tasks of designing programs and reasoning about them. However, they also pose certain challenges in the efficient implementation of functional programs because they are often written in a more high–level style and may bear little resemblance to the machines that execute them. Our purpose in this dissertation is to develop new techniques for the efficient analysis and implementation of functional programs.

1.1 Functional Programming: What and Why

A purely functional program consists entirely of functions and constants, with the main program being a function application. High–level functions in the program are defined using lower–level functions, until at the lowest level the functions are language primitives. Functions can also accept functions as arguments and return them as results. Such functions are usually called *higher–order* functions.

Purely functional programming languages differ from imperative programming languages in two major ways: that function application is the only control structure, and that side–effects are not allowed. There are no side–effects because, within a binding scope, the value bound to a variable will never be changed. (The same variable name, however, may be bound to other values in other binding scopes.) Depending on how a function application is invoked during the execution of a program, purely functional languages can

List of Tables

3.1 The value table of application $[\![M]\!]d$ and $[\![N]\!]d$, for all $d \in D_{\mathit{2}\to\mathit{2}}$. 49
3.2 The abstraction of the primitive functions `hd`, `tl`, and `cons` over domain *4*. 65
3.3 The abstraction of the primitive functions `hd`, `tl`, and `cons` over domain *6*. 65
3.4 The definitions in language Λ^0 of some semantic elements over the finite domains *2*, *3*, *4*, *5*, and *6*. 67
3.5 The results of applying $cat^{\#}$ to all the elements in the abstract domain *6*. . 71

5.1 Operation counts of the four programs. 105
5.2 Execution time of the four programs. 105

5.3 A walk in a region tree. 97
5.4 Four ways to cross a region. 99
5.5 Trace graphs of the read sequences of four programs. 106
5.6 Magnified versions of the trace graphs. 107

List of Figures

2.1 Denotational semantics of some predefined terms of a functional language. . 17

2.2 A functional program `factorial` that computes the factorial number of its input argument. 17

2.3 A diagram showing how to inductively define the abstraction function over domain $\alpha \to \beta$ based on the abstraction functions over domains α and β. . 23

2.4 The relationship, in terms of the Hoare power domains, between a function $f \in \alpha \to \beta$ and its abstraction $abs_{(\alpha\to\beta)\to(\bar{\alpha}\to\bar{\beta})}(f) \in \bar{\alpha} \to \bar{\beta}$. 24

2.5 The lattice structure of domain $2 \times 2 \times 2$. 33

3.1 The 10 elements in domain $D_{(2\to2)\to(2\to2)}$. 53

3.2 The definitions of functions `append`, `foldr`, and `cat`. 65

3.3 Strictness analysis for function `append`. 68

3.4 The strictness property of function `foldr`, described in language Λ^0. 70

3.5 Strictness analysis for function `cat`. 71

4.1 A purely functional program with two independent threads updating the same array. 74

4.2 A purely functional program that can be implemented single–threadedly. . . 76

4.3 Two functional programs which illustrate that single–threadedness is a global property of a program, not a local one. 77

5.1 The shallow binding scheme, as illustrated by configurations of a context tree after two rerootings. 88

5.2 The fragmented shallow binding scheme, as illustrated by configurations of a version tree after two rerootings. 93

A Proofs **130**

B SML Code for Fully Persistent Arrays **135**

B.1 SML Signature of Fully Persistent Arrays 135

B.2 SML Code for Fully Persistent Arrays 136

C SML Code for Real–Time Deques **139**

C.1 SML Signatures of the Stack and Deque Structures 139

C.2 SML Code for Stacks 140

C.3 SML Code for Real–Time Deques 141

Bibliography **146**

3.4.3 Embedding of Other Finite Domains 61
3.4.4 Complexity and Approximation 62
3.5 Strictness Analysis over Non–Flat Domains: An Example 63
3.6 Conclusion and Comparison to Other Work 69

4 Aggregate Data Structures in Functional Programs 73
4.1 Safe In–Place Modifications 75
4.2 Language Restrictions or Programming Styles 78
4.3 Efficient Implementations of Persistent Aggregates 81

5 Fully Persistent Arrays for Functional Programs 84
5.1 Survey and Motivation 85
5.2 Deep Binding and Shallow Binding 87
5.3 Functional Arrays by Shallow Binding and Fragmented Shallow Binding . . 89
5.4 Time and Space Complexity of the Fragmented Shallow Binding Scheme . . 94
5.5 Examples and Empirical Results 103
5.6 Future Work 108

6 Purely Functional, Real–Time Deques 110
6.1 Introduction and Survey 110
6.2 Functional Queues with Good Real–Time Performance 113
6.3 Functional Deques with Good Amortized Performance 115
6.4 Functional Deques with Good Real–Time Performance 117
6.5 Applications 121
6.5.1 Purely Functional List Processing 121
6.5.2 Real–Time Simulation of a Multihead Turing Machine in a Purely Functional Language 121
6.6 Remarks 123
6.7 Related Work and Future Work 124

7 Conclusion and Future Work 126
7.1 A Syntactic Method for Least Fixed Point Finding 126
7.2 Efficient Implementations of Functional Aggregates 127

Contents

Acknowledgments iv

Abstract vi

List of Figures xi

List of Tables xiii

1 Introduction 1

1.1 Functional Programming: What and Why 1

1.2 Objectives of the Dissertation 4

1.3 Outline of the Dissertation 5

2 Semantics–Based Analyses and Least Fixed Points 8

2.1 Denotational Semantics — An Overview 8

2.2 Abstract Interpretations of Functional Programs 19

2.3 Finding Least Fixed Points over Finite Domains 27

3 A Syntactic Method for Least Fixed Point Finding 35

3.1 Motivation and Introduction 35

3.2 A Language Λ for Finite Domains 38

3.3 A Complete Sub–Language Λ^0 of Λ 49

3.4 Implementation Issues 58

3.4.1 Translation from Language Λ to Language Λ^0 58

3.4.2 Relaxation of Language Λ^0 to Language Λ^1 59

scheme, however, uses side–effects and cannot be implemented in purely functional languages themselves. We then investigate the possibility of implementing efficient aggregates without using side–effects, and show that real–time deques can be implemented in a purely functional way. We describe several interesting applications of this technique.

Abstract

Functional programming languages provide programmers with clean semantics and side–effect free computation, which make easier the tasks of designing programs and reasoning about them. Efficient implementations of purely functional programs, however, can pose certain challenges. Our purpose in this dissertation is to develop new techniques for the efficient analysis and implementation of functional programs.

Our first goal is to investigate a syntactic approach, contrary to the usual semantic approaches, of finding the least fixed points of higher–order functions over finite domains. The second objective is to develop implementation techniques for aggregate data structures for functional programs such that accesses to aggregates are both efficient and side–effect free.

Finding the least fixed point of a monotonic function over a finite domain is an essential task when analyzing a functional program in the framework of abstract interpretation. Previous methods for least fixed point finding have primarily used semantic approaches, which often traverse large portions of the semantic domain and may be very inefficient even for simple programs. We propose a syntactic method based on an augmented simply typed lambda calculus. It is shown that, for finite domains, the syntactic method is both sound and complete with respect to the semantics. Moreover, we demonstrate that the proposed syntactic method can be quite effective in cases where the usual semantic method is very inefficient.

Efficient implementations of aggregate data structures for functional programs has been an active research topic. The problem arises because once an aggregate is updated, both the old version and newly updated copy must be preserved to maintain the side–effect free semantics of functional languages. We modify the shallow binding scheme of Baker to implement functional arrays for efficient incremental updates and voluminous reads. The

course taught by Ferng-Ching Lin at National Taiwan University in 1982. I thank him for providing an inspiring role model.

My father and mother provided the financial resource for my first year's study at New York University, and have been very supportive ever since. Thanks, Mom and Dad. Finally, I would like to thank my wife for her love and understanding over the years. Her confidence in me is always a source of encouragement. Thank you, Kuei–Feng.

This research was supported in part by National Science Foundation (#CCR–8909634), by DARPA (DARPA/ONR #N00014–91–J1472, #N00014–92–J1719), and by a Dean's Dissertation Fellowship from the Graduate School of Arts and Science, New York University.

Acknowledgments

It is a great pleasure to thank the many people who have been helping me to produce the work described in this dissertation.

First and foremost, I would like to thank my advisor, Benjamin Goldberg, for his support. Without his continuous encouragement and advice, both on the work reported here and on other aspects of research, it is doubtful whether this dissertation will eventually materialize. I would like to thank Malcolm Harrison and Edmond Schonberg, who have been very supportive. They, together with Ben and Robert Paige, also helped me, several times, to secure financial assistantship for study and research. I also thank Ernest Davis, Robert Dewar, and Dennis Shasha, not only for being in the committee, but also for their flexibility in scheduling the examination date.

I like to thank Fritz Henglein and Robert Paige for their help and advice. Special thanks go to Paul Hudak, who encouraged me greatly, and to Henry G. Baker and Martin Odersky for their interest in my work. I thank Adam Buchsbaum for pointing out related work and for lending an ear to half–baked ideas.

I had many stimulating discussions with the fellow students in the Computer Science Department, especially with Chia–Hsiang Chang, Konstantin Läufer, Atul Sibal, and Young Gil Park. Together with them and with Chien–Chan Hsu, Wen–Liang Hwang, Ping–Wen Ong, Frank Chee–Da Tsai, and other students, I have had quite a memorable time.

If not for Priscilla Karant, who introduced the writings of Henry David Thoreau and Russell Baker to her class at the American Language Institute, New York University, I probably would not enjoy reading English as much as I do now. I thank her very much. I also thank the administration and system staff at the Courant Institute for their assistance. I was attracted to programming language semantics and related topics when taking a

To Kuei-Feng and Our Parents

© Tyng–Ruey Chuang
All Rights Reserved, 1993.

New Techniques for the Analysis and Implementation of Functional Programs

by
Tyng–Ruey Chuang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Department of Computer Science
New York University

July 1993

Approved

Benjamin Goldberg
Advisor