Evaluador de Lógica Modal embebido en Haskell

Bautista José Peirone[0009-0004-7282-660X]

Universidad Nacional de Rosario
bpeirone@dcc.fceia.unr.edu.ar

Resumen Las lógicas modales son una familia de lógicas derivadas de la lógica proposicional, las cuales resultan útiles para expresar y probar razonamientos que requieren un mayor grado de expresividad que el provisto por la lógica de la cual parten. Si bien pueden aplicarse en diversas áreas, no existen muchas herramientas que permitan utilizar las mismas de manera fácil. De aquí surge la iniciativa de implementar una herramienta al estilo de un lenguaje de dominio específico embebido (EDSL, embedded domain-specific language) que permita al usuario del lenguaje poder trabajar de forma algebraica con los símbolos de la lógica de manera muy sencilla.

Keywords: Lógica Modal, Haskell, Programación Funcional, Lenguaje Embebido de Dominio Específico

Modal Logic interpreter embedded in Haskell

Abstract. Modal logics are a family of logics derived from propositional logic that allow for more expressive reasoning than the one provided by the one they build from. Despite its numerous theoretical and practical applications there is a relatively scarce number of tools available for working with them in a user-friendly and intuitive manner. This work presents the implementation of an embedded domain-specific language (EDSL) designed with the goal to make algebraic manipulation of modal logic formulas simple.

Keywords: Modal Logic, Haskell, Functional Programming, Embedded Domain Specific Language

1 Introducción

1.1 Lógica Modal

La lógica modal es una teoría formal para realizar deducciones que parte de agregar dos operadores unarios a los presentes en la lógica proposicional, $\Box, \Diamond$, llamados operadores modales. Estas preceden fórmulas y les brindan interpretaciones nuevas, donde $\Box\phi$

significa “es necesario que ϕ”, mientras que $\Diamond\phi$ significa “es posible que ϕ”. Sin embargo, estas no son las únicas formas de entender lo que representan (Buehler, 2014), y esto contribuye al enriquecimiento de esta familia de lógicas con diversidad de usos y campos de aplicación tanto teóricos como prácticos, que van desde los silogismos filosóficos hasta la verificación de sistemas de software. Además, ambos operadores están relacionados por la equivalencia $\Box\phi \iff \neg\Diamond\neg\phi$, por lo que usualmente se le asigna interpretación solo a uno de ellos y la otra es obtenida a partir de esta relación.

La semántica de esta lógica se basa en un modelo de grafos dirigidos (Vardi, 1997), y añade una forma de relacionar fórmulas con propiedades en grafos (Huth and Ryan, 2004), lo cual amplía aún más las formas de interpretar estas lógicas. Esta idea se desarrolla con más detalle en el cuerpo del trabajo, junto con su implementación en el lenguaje.

Además, esta familia de lógicas es utilizada como punto de partida para otras más complejas, como la lógica temporal o lógica de acciones, que surgen de extender a modelos y predicados más complejos.

Consideremos un conjunto de átomos $\mathbb{A}$, el conjunto $\mathcal{F}$ de fórmulas modales se define inductivamente como el menor conjunto tal que las siguientes propiedades son válidas:

- $\mathbb{A} \subset \mathcal{F}$
- $\forall\phi \in \mathcal{F}, \{\neg\phi, \Box\phi, \Diamond\phi\} \subset \mathcal{F}$
- $\forall\phi,\psi \in \mathcal{F}, \{\phi \wedge \psi, \phi \vee \psi, \phi \to \psi, \phi \longleftrightarrow \psi\} \subset \mathcal{F}$

1.2 Kripke Frames como modelo semántico

La semántica de esta lógica depende de una estructura conocida como Kripke Frame (Estivill-Castro and Rosenblueth, 2013). Esta consiste en un grafo cuyos vértices representan estados y sus aristas las transiciones entre ellos, y una función de etiquetado que indica qué proposiciones atómicas son verdaderas en cada estado, ambas siendo fácilmente representables en un lenguaje de programación.

Es decir, formalmente definimos un modelo como $M = (V, E, T)$, donde V es el conjunto de vértices, $E \subset V \times V$ representa las transiciones entre estados y $T : V \to \mathcal{P}(\mathbb{A})$ es la función de etiquetado, que a cada vértice le asigna el conjunto de átomos que son verdaderos en él. El operador secuente indica qué fórmulas valen en cada estado y se define como la relación $\Vdash \subset V \times \mathcal{F}$ que verifica:

$$
\begin{aligned}
w \Vdash p &\iff p \in T(w) \\
w \Vdash \phi \wedge \psi &\iff w \Vdash \phi \text{ y } w \Vdash \psi, \\
w \Vdash \phi \vee \psi &\iff w \Vdash \phi \text{ o } w \Vdash \psi, \\
w \Vdash \neg\phi &\iff w \nVdash \phi, \\
w \Vdash \phi \to \psi &\iff w \Vdash \phi \implies w \Vdash \psi. \\
w \Vdash \phi \longleftrightarrow \psi &\iff w \Vdash \phi \iff w \Vdash \psi. \\
w \Vdash \Box\phi &\iff \forall v \in W, \text{ si } (w,v) \in E, \text{ entonces } v \Vdash \phi, \\
w \Vdash \Diamond\phi &\iff \exists v \in W \text{ tal que } (w,v) \in E \text{ y } v \Vdash \phi.
\end{aligned}
$$

Las primeras seis reglas son análogas a las que encontramos en la lógica proposicional, ya que los operadores asociados a estas se heredan de ella. Las últimas dos reglas son las propias de los operadores modales y es donde entran en juego las transiciones entre estados. Ambos operadores, $\Box$ y $\Diamond$, cumplen los roles de cuantificador universal y existencial respectivamente, aunque de forma más limitada que en la lógica de predicados o de ordenes mayores. Aquí la cuantificación se da sobre un conjunto finito, característica fundamental que hace a esta lógica decidible (Vardi, 1997).

Estas reglas son adaptadas al lenguaje, y cuando la relación es válida, la expresión correspondiente evalúa a `True`, y cuando no lo es, evalúa a `False`. Sobre la relación del secuente se construyen las operaciones de satisfacibilidad y validez, denominadas respectivamente `isSatis` y `isValid`, y definidas como

$$\frac{\forall w \in W,\ w \Vdash \phi}{\texttt{isValid}\ \phi} \qquad \frac{\exists w \in W,\ w \Vdash \phi}{\texttt{isSatis}\ \phi}$$

A modo de ejemplo, se presenta un posible modelo, donde el conjunto de fórmulas atómicas al lado de cada vértice indica las que son verdaderas en dicho estado.

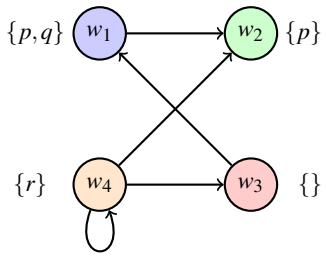

Figura 1. Modelo de ejemplo

Basándonos en este ejemplo, podemos clasificar las siguientes afirmaciones como

✓ $w_1 \Vdash p \wedge \neg t$
✓ $w_4 \Vdash \Diamond r$
✓ $w_2 \Vdash \Box(p \wedge \neg p)$
✗ $w_3 \Vdash p \vee q \vee r$
✗ $w_4 \Vdash \Diamond(p \wedge r)$

1.3 Lenguaje Embebido de Dominio Específico

El objetivo del trabajo es implementar una herramienta práctica para el manejo algebraico y automático de fórmulas y modelos, por lo tanto, el poder de expresividad de la sintaxis junto con su semántica asociada son reducidas al campo de operadores lógicos, descripción de funciones como pares ordenados y relaciones, por ello un lenguaje de dominio específico es una herramienta que se adapta muy bien a las necesidades y flexibilidad requerida del problema.

La implementación de nuestro programa como un lenguaje embebido en Haskell se debe a diversos factores. Haskell es un lenguaje puramente funcional, estático y fuertemente tipado, (Hutton, 2016), (Gibbons, 2013). Por un lado, su infraestructura de alto nivel nos resuelve muchos problemas al proveernos de un sistema de tipos muy expresivo que facilita la representación de las estructuras y modelos que encontramos en nuestro programa, y la verificación de tipos resulta muy útil para la detección temprana de errores durante las etapas de desarrollo. Además, podemos utilizar las herramientas de programación y compilado con las que ya cuenta Haskell. Por otro lado, el desarrollo embebido nos permite trabajar en un diseño e implementación de alto nivel, sin tener que preocuparnos por los detalles de los que se encarga el lenguaje anfitrión, como por ejemplo la entrada y salida de archivos, compilación, entre otros.

Finalmente, las cualidades funcionales de Haskell nos permiten abstraer nociones de estado (Hughes, 1989), aislando mejor cada componente en la medida de lo posible, facilitando su desarrollo y escalabilidad. Teniendo en cuenta que la herramienta fue diseñada para que la extensión a variantes de la familia modal resulte sencilla, este factor es de gran importancia.

Todo esto nos lleva a la elección de Haskell como lenguaje anfitrión, porque provee una forma simple de implementar una herramienta con las características buscadas en la solución a nuestro problema.

2 Desarrollo

2.1 Capacidades del lenguaje

La herramienta presentada en este trabajo consiste en una interfaz interactiva que permite definir fórmulas, cargar modelos y evaluar expresiones, para lo cual se utiliza la semántica previamente detallada, y aplicar una operación para determinar si un conjunto de axiomas se verifica sobre un modelo. Además, esta admite la carga de archivos de extensión `mll` para poder almacenar definiciones y modelos.

Al definir una fórmula se le asigna un nombre a la misma, para que pueda ser utilizada en posteriores expresiones. De esta manera se pueden definir estructuras comunes entre fórmulas y después aplicar sustituciones sintácticas sobre ellas para obtener otras. Por ejemplo si definimos la fórmula *P* como:

```
def P = p and q -> r
```

y aplicamos en *P* las siguientes sustituciones:

```
P [a and t/p][s -> t /r]
```

obtenemos una fórmula equivalente a:

```
(a and t) and q -> (s -> t)
```

Los nombres de las definiciones deben empezar con mayúscula, y los de los átomos con minúscula, para que no haya ambigüedades a la hora de evaluar las expresiones.

A continuación se expresa la carga del modelo de ejemplo de la figura 1:

```
set frame = {
  w1 -> {w2},
  w3 -> {w1},
  w4 -> {w2,w3,w4}
}

set tag = {
  w1 -> {p,q},
  w2 -> {p},
  w4 -> {r}
}
```

En la primer sentencia se definen los vértices V y las transiciones entre estos, E, mientras que en la segunda sentencia se define el etiquetado de vértices T. Estas partes se definen por separado para poder modificar el etiquetado sin tener que cambiar las transiciones y viceversa, lo cual hace más flexible y dinámica la carga de diversos modelos.

Evaluar la fórmula f en el estado w del modelo se corresponde con la expresión `w ||- f`. El modelo que se utiliza para la evaluación es siempre el último que se cargó.

También se pueden evaluar las dos operaciones adicionales mencionadas anteriormente, verificación de validez y satisfacibilidad, definidas sobre la operación primitiva anterior, escribiendo las expresiones `isValid f` y `isSatis f` respectivamente.

Uno de los aspectos principales de la lógica modal es la estrecha relación que existe entre algunas de sus fórmulas y propiedades sobre los grafos de los modelos (Huth and Ryan, 2004). Por ejemplo, la propiedad $\Box\phi \rightarrow \phi$ vale para cualquier fórmula modal ϕ en un modelo M si y solo si el grafo de M es reflexivo.

Esta relación entre fórmulas y propiedades de grafos permitió implementar de manera sencilla en el EDSL una operación para asegurar que se cumplen determinados axiomas, analizando las propiedades sobre el grafo subyacente que se corresponden con cada axioma. La sintaxis de esta operación es la siguiente:

```
assume {Axiom-A1, Axiom-A2, ...}
```

donde `A1, .. , An` son nombres de axiomas pertenecientes a una lista definida en la implementación en la cual se encuentran. Por ejemplo, el axioma de la propiedad antes mencionada, $\Box\phi \rightarrow \phi$, se llama T, y por tanto se podría verificar la propiedad escribiendo:

```
assume {Axiom-T}
```

Es común ver varios de estos axiomas usados en conjunto ya que acarrean alguna interpretación propia para los operadores. A estos conjuntos de axiomas se los denominan lógicas, y el lenguaje provee las más utilizadas y conocidas. Por lo tanto, si el conjunto de axiomas a verificar se corresponde con alguna lógica implementada en el lenguaje, se puede utilizar la expresión:

```
assume Logic-L
```

donde `L` es el nombre de dicha lógica.

Finalmente, el lenguaje admite una configuración para poder observar más en detalle los pasos de evaluación de cada una de las expresiones con el fin de proveer al usuario información que puede ser útil para la interpretación de los resultados.

2.2 Implementación de la herramienta

La construcción del intérprete se dividió en los pasos de representación de datos, análisis sintáctico y evaluación.

Los grafos los interpretamos como funciones de sus vértices a su conjunto de vecinos. Por otro lado, el etiquetado de vértices puede también plantearse como una función con dominio en los vértices del grafo y cuya imagen corresponde al conjunto de átomos válidos en cada uno. Los modelos fueron implementados en Haskell con los siguientes tipos de datos:

```
data Graph v = Graph
  { vertices :: [v]
  , edges    :: M.Map v [v]
  }
```

```
data Model w a = Model
  { graph :: Graph w
  , tag   :: M.Map w (S.Set a)
  }
```

Las fórmulas escritas en el lenguaje soportan dos características de alto nivel que facilitan el proceso de escritura: sustituciones sintácticas y definiciones por nombre. Para implementar esto fue necesario separar en dos etapas el procesamiento de las fórmulas. Los tipos de datos de los resultados de cada etapa son los presentados en la figura 2. La única diferencia entre estos tipos es la posibilidad que provee el primero de ellos para expresar las dos características anteriormente mencionadas. El analizador sintáctico genera una salida de tipo `F1 String`, y el módulo `Elab.hs` luego convierte la representación de `F1` a `F2`. Parte del algoritmo utilizada para esta conversión se presenta en la figura 3.

```
elab' env (LAtomic x)     = return $ sub env (Atomic x)
elab' env (LIdent v)      = do def <- snd <$> getEnv
                               maybe (undefVarError v) (return
                                    . sub env) (lookup v def)
elab' env (LAnd s1 s2)    = do f1 <- elab' env s1
                               f2 <- elab' env s2
                               return $ And f1 f2
elab' env (LSub x s1 s2) = do f2 <- elab' env s2
                               elab' ((x,f2):env) s1
```

Figura 3. Fragmento de la elaboración de términos

```
data F1 a = LBottom
  | LTop
  | LAtomic a
  | LIdent a
  | LSub    a (F1 a) (F1 a)
  | LAnd      (F1 a) (F1 a)
  | LOr       (F1 a) (F1 a)
  | LImply    (F1 a) (F1 a)
  | LIff      (F1 a) (F1 a)
  | LNot      (F1 a)
  | LSquare   (F1 a)
  | LDiamond  (F1 a)
```

```
data F2 a  = Bottom
  | Top
  | Atomic a
  | And      (F2 a) (F2 a)
  | Or       (F2 a) (F2 a)
  | Imply    (F2 a) (F2 a)
  | Iff      (F2 a) (F2 a)
  | Not      (F2 a)
  | Square   (F2 a)
  | Diamond  (F2 a)
```

Figura 2. Representación de fórmulas

Para la implementación del analizador sintáctico se utilizó una libreria de Haskell, Happy. A partir de una gramática, Happy construye un parser para esta, lo cual acelera y hace muy flexible el desarrollo del parser.

Debido a que tanto los grafos como los etiquetados los pensamos como funciones, solo es necesario diseñar una gramática que permita especificar funciones por extensión genéricas. Se implementó la siguiente gramática en el analizador sintáctico de funciones

$$
\begin{aligned}
\mathit{funcion} &::= \text{`\{'}\ \mathit{secuenciaAsignaciones}\ \text{`\}'} \\
\mathit{secuenciaAsignaciones} &::= \mathit{secuenciaAsignaciones}\ \text{`,'}\ \mathit{asignacion} \\
&\quad \mid \mathit{asignacion} \\
&\quad \mid \varepsilon \\
\mathit{asignacion} &::= \mathit{identificador}\ \text{`->'}\ \text{`\{'}\ \mathit{secuenciaIdentificadores}\ \text{`\}'}
\end{aligned}
$$

Para las sentencias del lenguaje, que comprenden desde asignaciones, especificación de modelos y expresiones a evaluar, se utiliza la sintaxis definida por

$$
\begin{aligned}
\mathit{sentencia} ::= {} & \text{`def'}\ \mathit{identificador}\ \text{`='}\ \mathit{formula} \\
& \mid \text{`set'}\ \text{`frame'}\ \text{`='}\ \mathit{funcion} \\
& \mid \text{`set'}\ \text{`tag'}\ \text{`='}\ \mathit{funcion} \\
& \mid \mathit{expresion}
\end{aligned}
$$

Y las expresiones, divididas en 4 tipos, se determinan por

$$
\begin{aligned}
\mathit{expresion} ::= {} & \text{`assume'}\ \mathit{logica} \\
& \mid \text{`isSatis'}\ \mathit{formula} \\
& \mid \text{`isValid'}\ \mathit{formula} \\
& \mid \mathit{estado}\ \text{`||-'}\ \mathit{formula}
\end{aligned}
$$

Los lenguajes funcionales suelen carecer, a primera vista, de características consideradas fundamentales para casi cualquier persona acostumbrada a las facilidades provistas por los lenguajes imperativos. Entre estas se encuentran el manejo de excepciones,

variables mutables accesibles de forma global o incluso entrada y salida de archivos o terminal. El paradigma funcional resuelve estas carencias con funciones de alto orden, tipos polimórficos y, a menudo, patrones especiales de escritura que permiten este manejo de forma más organizada, como lo hace nuestro lenguaje anfitrión con mónadas (Wadler, 1995). Este es solo un nombre elaborado para referirse a un estilo de programación que abstrae en la implementación las características mencionadas.

Esto es lo que se observa en el fragmento del programa de la figura 3, donde se utilizó una mónada implementada con transformadores provistos en la librería estándar, además de proveerles una interfaz propia que se adecuara al uso.

Teniendo ya una representación completa del estado diseñado para el problema, con un manejo cómodo de este, se procede a diseñar el proceso de evaluación. Este queda dividido en dos módulos totalmente separados e independientes, ya que cada uno implementa operaciones del lenguaje que dependen de algoritmos totalmente distintos: `Modal.hs` y `Frame.hs`. El primero se encarga de la evaluación semántica de fórmulas sobre modelos, mientras que el segundo implementa la verificación de propiedades sobre grafos, asociada a la operación `assume`.

Por ejemplo, en el segundo módulo mencionado se implementan las funciones isReflexive e isSymmetric, la primera verifica que para cada vértice v del grafo exista la arista (v,v), mientras que la segunda chequea que para cada arista (v,u) también exista (u,v).

```
type GraphProperty v = Graph v -> Bool

isReflexive :: Ord v => GraphProperty v
isReflexive g = all hasLoop (vertices g)
    where
        hasLoop v = existsEdge g (v,v)

isSymmetric :: Ord v => GraphProperty v
isSymmetric g = checkSymmetry (vertices g)
    where
        checkSymmetry [] = True
        checkSymmetry (v:vs) = all (isSymmetricPair v) vs &&
            checkSymmetry vs
        isSymmetricPair x y = existsEdge g (x,y) ==
            existsEdge g (y,x)
```

En cuanto a la evaluación de fórmulas, la lógica principal se halla en la implementación de la relación de secuente ($\Vdash$). Distinguimos principalmente tres casos: un átomo, un operador n-ario proposicional y un operador modal. Cabe destacar que la versión mostrada aquí es una simplificación que omite la construcción de trazas de ejecución, debido a que no es relevante.

En el caso del átomo, se accede al modelo del entorno y se verifica si en dicho estado vale de acuerdo con el etiquetado. Este es uno de los casos base de la recursión. La evaluación sobre las operaciones es muy sencilla, consiste en aplicar recursivamente la función sobre los argumentos de la operación y utilizar los resultados con el operador correspondiente a esta en Haskell. Nuevamente, las mónadas facilitan la ocultación del estado, y permiten propagar el modelo en cada llamada recursiva. Finalmente, un ope-

rador modal requiere también la evaluación recursiva sobre todos los vecinos de nuestro estado actual, lo cual queda determinado por el grafo, y la secuencia de valores obtenida es reducida mediante una operación que depende del operador modal en cuestión.

```
w ||- f@(Atomic p)  =
    do model <- ask
       let b = p `elem` validAtoms model w
       return b

w ||- f@(And f1 f2) =
    do t1 <- w ||- f1
       t2 <- w ||- f2
       return (t1 && t2)

w ||- f@(Square f1) =
    do model <- ask
       ts <- mapM (||- f1) (nextStates model w)
       return (and ts)

w ||- f@(Diamond f1) =
    do model <- ask
       ts <- mapM (||- f1) (nextStates model w)
       return (or ts)
```

2.3 Conclusión

Este trabajo consistió en el diseño e implementación de un lenguaje de dominio específico para poder representar modelos y fórmulas de la lógica modal. Este se implementó como un lenguaje embebido en Haskell para aprovechar la infraestructura y herramientas que este lenguaje ya nos provee, como su sistema de tipos, funciones de alto orden, compiladores y librerías. Además, la elección de un lenguaje funcional facilitó varias etapas del proceso por su simplicidad para representar el estado interno del programa y el manejo de errores.

Consideramos que este trabajo presenta potencial para su aplicación en contextos académicos, particularmente en la enseñanza de lógica modal y sus aplicaciones. La naturaleza interactiva de la herramienta, combinada con su capacidad para visualizar tanto modelos como procesos de evaluación, la convierte en un recurso valioso para la comprensión de conceptos abstractos que tradicionalmente resultan desafiantes para los estudiantes. Su uso podría facilitar la resolución de problemas prácticos mediante la experimentación directa con diferentes configuraciones de modelos y fórmulas, permitiendo a los estudiantes desarrollar intuición sobre las propiedades de los sistemas modales. Además, la posibilidad de verificar axiomas y explorar correspondencias entre propiedades de grafos y validez de fórmulas ofrece un enfoque pedagógico que conecta la teoría formal con representaciones visuales concretas, lo que puede resultar especialmente beneficioso en cursos introductorios de lógica computacional y verificación formal.

2.4 Trabajo futuro

La herramienta presenta a día de hoy numerosas ramas con desarrollo pendiente, tanto para expandir el poder expresivo como para aumentar las capacidades ya existentes. Es claro que el potencial está limitado a evaluaciones sobre modelos; sin embargo, las gramáticas y tipos de datos son suficientemente flexibles como para que cualquier modificación resulte sencilla de aplicar. Los ejemplos más claros y adecuados son las familias LTL y CTL (Huth and Ryan, 2004), resultantes de ampliar la familia de operadores disponibles a costa de dar semántica propia a estos, pero usando el mismo tipo de modelos utilizados en la familia modal aquí presentada.

Adicionalmente, se identifica la necesidad de mejorar significativamente la visualización de resultados, particularmente en lo que respecta a la presentación de trazas de ejecución y la representación gráfica de modelos complejos. Otras áreas de desarrollo incluyen la optimización del rendimiento para el manejo de modelos de gran escala, la implementación de pruebas automatizadas más exhaustivas, y la extensión de la documentación mediante casos de estudio que demuestren la aplicabilidad de la herramienta en escenarios reales de verificación formal.

Bibliografía

Buehler RJ (2014) Modal reasoning. Based on lectures by W. Holliday

Estivill-Castro V, Rosenblueth DA (2013) Efficient construction of kripke structures and model checking of logic-labeled sequential finite state machines. INFORMATION 16(2(B)):1555–1560

Gibbons J (2013) Functional programming for domain-specific languages. In: Central European Functional Programming - Summer School on Domain-Specific Languages, Springer, LNCS, vol 8606, pp 1–28, DOI 10.1007/978-3-319-15940-9_1, URL http://link.springer.com/chapter/10.1007/978-3-319-15940-9_1

Hughes J (1989) Why Functional Programming Matters. The Computer Journal 32(2):98–107, DOI 10.1093/comjnl/32.2.98, URL https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/32.2.98

Huth MRA, Ryan MD (2004) Logic in computer science: modelling and reasoning about systems. In: Logic in computer science: modelling and reasoning about systems, 2nd edn, Cambridge university press, Cambridge, pp 306–328

Hutton G (2016) Programming in Haskell, 2nd edn. Cambridge University Press, USA

Vardi MY (1997) Why is modal logic so robustly decidable?, URL http://www.cs.rice.edu/~vardi, department of Computer Science, Rice University

Wadler P (1995) Monads for functional programming. In: Jeuring J, Meijer E (eds) Advanced Functional Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 24–52