Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing

Tillmann Rendel Klaus Ostermann
University of Marburg, Germany

Abstract

Parsers and pretty-printers for a language are often quite similar, yet both are typically implemented separately, leading to redundancy and potential inconsistency. We propose a new interface of *syntactic descriptions*, with which both parser and pretty-printer can be described as a single program. Whether a syntactic description is used as a parser or as a pretty-printer is determined by the implementation of the interface. Syntactic descriptions enable programmers to describe the connection between concrete and abstract syntax once and for all, and use these descriptions for parsing or pretty-printing as needed. We also discuss the generalization of our programming technique towards an algebra of partial isomorphisms.

Categories and Subject Descriptors D.3.4 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Design, Languages

Keywords embedded domain specific languages, invertible computation, parser combinators, pretty printing

1. Introduction

Formal languages are defined with a concrete and an abstract syntax. The concrete syntax specifies how words from the language are to be written as sequences of characters, while the abstract syntax specifies a structural representation of the words well-suited for automatic processing by a computer program. The conversion of concrete syntax to abstract syntax is called parsing, and the conversion of abstract syntax into concrete syntax is called unparsing or pretty printing.

These operations are not inverses, however, because the relation between abstract and concrete syntax is complicated by the fact that a single abstract value usually corresponds to multiple concrete representations. An unparser or pretty printer has to choose among these alternative representations, and pretty printing has been characterized as choosing the "nicest" representation (Hughes 1995).

Several libraries and embedded domain-specific languages (EDSLs) for both parsing and pretty printing have been proposed and are in wide-spread use. For example, the standard libraries of the Glasgow Haskell Compiler suite include both Parsec, an embedded parser DSL (Leijen and Meijer 2001), and a pretty printer EDSL (Hughes 1995). However, these EDSLs are completely independent, which precludes the use of a single embedded program to specify both parsing and pretty printing. This means that due to the dual nature of parsing and pretty-printing a separate specification of both is at least partially redundant and hence a source of potential inconsistency.

This work addresses both invertible computation and the unification of parsing and pretty printing as separate, but related challenges. We introduce the notion of *partial isomorphisms* to capture invertible computations, and on top of that, we propose a language of *syntax descriptions* to unify parsing and pretty printing EDSLs. A syntax description specifies a relation between abstract and concrete syntax, which can be interpreted as parsing a concrete string into an abstract syntax tree in one direction, and pretty printing an abstract syntax tree into a concrete string in the other direction. This dual use of syntax descriptions allows a programmer to specify the relation between abstract and concrete syntax once and for all, and use these descriptions for parsing or printing as needed.

After reviewing the differences between parsing and pretty printing in Sec. 2, the following are the main contributions of this paper:

- We propose partial isomorphisms as a notion of invertible computation (Sec. 3.1).
- On top of partial isomorphisms, we present the polymorphically embedded DSL of syntax descriptions (Sec. 3) to eliminate the redundancy between parser and pretty-printer specifications while still leaving open the choice of parser/pretty-printer implementation.
- We provide proof-of-concept implementations of the language of syntax descriptions and discuss the adaption of existing parser or pretty printer combinators to our interface (Sec. 4).
- We illustrate the feasibility of syntactic descriptions in a case study, showing that real-world requirements for parsing and pretty-printing such as the handling of whitespace and infix operators with priorities can be supported (Sec. 4).
- We present a semantics of syntactic descriptions as a relation between abstract and concrete syntax as a possible correctness criterion for parsers and pretty-printers (Sec. 4.3).
- We explore the expressivity of partial isomorphisms by presenting fold and unfold as an operation on partial isomorphisms, implemented as a single function (Sec. 5).

Section 7 discusses related and future work, and the last section concludes. This paper has been written as literate Haskell and contains the full implementation. The source code is available for download at `http://www.informatik.uni-marburg.de/~rendel/unparse/`.

Haskell'10, September 30, 2010, Baltimore, Maryland, USA.

2. Parsing versus Pretty-Printing

EDSLs for parsing such as Parsec tend to be structured as parser combinator libraries, providing both elementary parsers and combinators to combine parsers into more complex ones. In a typed language, the type of a parser is usually a type constructor taking one argument, so that *Parser* α is the type of parsers which produce a value of type α when successfully run on appropriate input.

We will present parsers and pretty-printers in a style that makes it easy to see their commonalities and differences. Using the combinators for applicative functors (McBride and Paterson 2008), one can implement a parser for an algebraic datatype in such a way that the structure of the parser follows the structure of the datatype. Here is an example for a parser combinator producing a list:

```
data List α
  = Nil
  | Cons α (List α)

parseMany :: Parser α → Parser (List α)
parseMany p
  = const Nil <$> text ""
  <|> Cons    <$> p
              <*> parseMany p
```

The combinator $\langle|\rangle$ is used to choose between the possible constructors, $\langle\$\rangle$ is used to associate constructors with their arguments, and $\langle*\rangle$ is used to handle constructors with more than one field. Since *Nil* does not take any arguments, *const* is used to ignore the result of parsing the empty string.

The structure of *parseMany* follows the structure of *List*: *parseMany* is composed of a parser for empty lists, and a parser for non-empty lists, just like *List* is composed of a constructor for empty lists, and a constructor for non-empty lists.

On the other hand, EDSLs for pretty printing such as the library by Hughes (1995) are usually structured around a proper type *Doc* with elementary documents and combinators for the construction of more complex documents. These combinators can be used to write a pretty printer for a datatype such that the structure of the pretty printer follows the structure of the datatype.

```
printMany :: (α → Doc) → (List α → Doc)
printMany p list
  = case list of
      Nil        → text ""
      Cons x xs → p x
                  ◇ printMany p xs
```

The structure of *printMany* follows the structure of *List*, but this time, pattern matching is used to give alternative pretty printers for different constructors. The combinator $\diamond$ is used to combine two documents side by side.

We introduce a type synonym *Printer* to show the similarity between the types of *parseMany* and *printMany* even more clearly.

```
type Printer α = α → Doc

printMany :: Printer α → Printer (List α)
```

These code snippets show how the structure of both parsers and pretty printers are similar in following the structure of a datatype. Jansson and Jeuring (2002) have used this structural similarity between datatype declarations, and parsers and pretty printers for the same datatypes, to derive serialization and deserialization functions generically from the shape of the datatype. We offer the programmer more freedom in the choice of parser and pretty printer by using the structural similarity between parsers and pretty printers to unify these concepts without depending directly on the shape of some datatype.

But these snippets also show the remaining syntactic differences between parsers and pretty printers. Parsers use combinators $\langle\$\rangle$, $\langle*\rangle$ and $\langle|\rangle$ to apply functions and branch into the alternatives of the data type, while pretty printing uses the usual function application and pattern matching. This syntactic difference has to be resolved in order to unify parsing and pretty printing.

3. A language of syntax descriptions

We adapt polymorphic embedding of DSLs (Hofer et al. 2008) to Haskell by specifying an abstract language interface as a set of type classes. This interface can be implemented by various implementations, i.e., type class instances. A program in the DSL is then a polymorphic value, which can be used at different use sites with different type class instances, that is, which can be interpreted polysemantically.

In this section, we are concerned with the definition of the language interface for syntax descriptions as a set of type classes. Our goal is to capture the similarities and resolve the differences between parsing and pretty printing so that a single polymorphic program can be used as both a parser and a pretty printer.

The combinators $\langle\$\rangle$, $\langle*\rangle$ and $\langle|\rangle$ as shown in the previous section are at the core of parser combinator libraries structured with applicative functors. The combinator $\langle\$\rangle$ is used to associate semantic actions with parsers, the combinator $\langle*\rangle$ is used to combine two parsers in sequentially, and the combinator $\langle|\rangle$ is used to combine two parsers as alternatives. As we will see in the next subsection, these combinators cannot be implemented directly for *Printer*. Therefore, our goal in the following subsections is to find variants of $\langle\$\rangle$, $\langle*\rangle$ and $\langle|\rangle$ which can be implemented both for type constructors like *Parser* and for type constructors like *Printer*. These combinators will be assembled in type classes to form the language interface of the language of syntax descriptions.

3.1 The category of partial isomorphisms and the $\langle\$\rangle$ combinator

The *fmap* combinator for *Parser* (or its synonym $\langle\$\rangle$) is used to apply a pure function $\alpha \to \beta$ to the eventual results of a *Parser* α, producing a *Parser* β. The behavior of a $f \langle\$\rangle p$ parser is to first use p to parse a value of some type α, then use f to convert it into a value of some other type β, and finally return that value of type β.

$$(\langle\$\rangle) :: (\alpha \to \beta) \to \mathit{Parser}\ \alpha \to \mathit{Parser}\ \beta$$

Unfortunately, we cannot implement the same $\langle\$\rangle$ function for *Printer*, because there is no point in first printing a value, and then apply some transformation. Instead we would like to apply the transformation first, then print the transformed values. However, this would require a function of type $\beta \to \alpha$. The behavior of a $f \langle\$\rangle p$ pretty printer could be to first get hold of a value of type β, then use f to convert it into a value of some other type α, and finally use p to print that value of type α.

$$(\langle\$\rangle) :: (\beta \to \alpha) \to \mathit{Printer}\ \alpha \to \mathit{Printer}\ \beta$$

How can we hope to unify the types of $\langle\$\rangle$ for parsers and pretty printers? Our idea is to have functions that can be used both forwards and backwards. A $f \langle\$\rangle p$ parser could use f forwards to convert values after parsing, and a $f \langle\$\rangle p$ pretty printer could use f backwards before printing. Clearly, this would work for invertible functions, but not all functions expressible in Haskell, or any general-purpose programming language, are invertible. Since we cannot invert all functions, we have to restrict the $\langle\$\rangle$ operator to work with only such functions which can be used forwards and backwards.

An invertible function is also called an isomorphism. We define a data type constructor *Iso* so that *Iso* α β is the type of isomorphisms between α and β. More precisely, the type *Iso* α β captures

what we call *partial isomorphisms*. A partial isomorphism between α and β is represented as a pair of functions f of type $\alpha \to Maybe\ \beta$ and g of type $\beta \to Maybe\ \alpha$ so that if $f\ a$ returns *Just b*, $g\ b$ returns *Just a*, and the other way around.

```
data Iso α β
  = Iso (α → Maybe β) (β → Maybe α)
```

We are interested in *partial* isomorphisms because we want to modularly compose isomorphisms for the whole extension of a type from isomorphisms for subsets of the extension. For example, each constructor of an algebraic data type gives rise to a partial isomorphism, and these partial isomorphisms can be composed to the (total) isomorphism described by the **data** equation.

The partial isomorphisms corresponding to the constructors of an algebraic data type can be mechanically derived by a system like Template Haskell (Sheard and Jones 2002). For example, with the Template Haskell code in Appendix A, the macro call

```
$(defineIsomorphisms "List)
```

expands to the following definitions.

```
nil  :: Iso ()          (List α)
cons :: Iso (α, List α) (List α)

nil  = Iso
  (λ()      → Just Nil)
  (λxs      → case xs of
                Nil       → Just ()
                Cons x xs → Nothing)

cons = Iso
  (λ(x, xs) → Just (Cons x xs))
  (λxs      → case xs of
                Nil       → Nothing
                Cons x xs → Just (x, xs))
```

Partial isomorphisms can be inverted and applied in both directions.

```
inverse :: Iso α β → Iso β α
inverse (Iso f g) = Iso g f

apply :: Iso α β → α → Maybe β
apply (Iso f g) = f

unapply :: Iso α β → β → Maybe α
unapply = apply ∘ inverse
```

We will generally not be very strict with the invariant stated above (if $f\ a$ returns *Just b*, $g\ b$ returns *Just a*, and the other way around). In particular we will sometimes interpret this condition modulo equivalence classes. A typical example from our domain is that a partial isomorphism maps strings of blanks of arbitrary length to a unit value but maps the unit value back to a string of blanks of length one—that is, all strings of blanks of arbitrary length are in the same equivalence class.

The need for invertible functions can also be understood from a categorical point of view. In category theory, a type constructor such as *Parser* can be seen as a covariant functor from the category *Hask* of Haskell types and Haskell functions to the same category. This notion is captured in the standard Haskell *Functor* class, which provides the *fmap* function. Note that the usual $\langle\$\rangle$ for parsers is simply an alias for *fmap*.

```
class Functor f where
  fmap :: (α → β) → (f α → f β)
```

This kind of functor is called covariant because the direction of the arrow does not change between $\alpha \to \beta$ and $f\ \alpha \to f\ \beta$.

Unfortunately, *Printer* is not a covariant functor, because the type variable occurs in a contravariant position, to the left of a function arrow. Instead, it is a contravariant functor, which could be captured in Haskell by the following type class.

```
class ContravariantFunctor f where
  contrafmap :: (β → α) → (f α → f β)
```

This kind of functor is called contravariant because the direction of the arrow is flipped between $\beta \to \alpha$ and $f\ \alpha \to f\ \beta$. In general, value producers such as *Parser* are covariant functors, while value consumers such as *Printer* are contravariant functors.

Partial isomorphisms can be understood as the arrows in a new category different from *Hask*. Categories which differ from Hask in the type of arrows can be expressed as instances of the type class *Category*, which is defined in *Control.Category* as follows.

```
class Category cat where
  id  :: cat a a
  (∘) :: cat b c → cat a b → cat a c
```

The category of partial isomorphisms has the same objects as *Hask*, but contains only the invertible functions as arrows. It can be expressed in Haskell using the following instance declaration.

```
instance Category Iso where
  g ∘ f = Iso (apply f ≫ apply g)
              (unapply g ≫ unapply f)
  id    = Iso Just Just
```

The $\gg$ combinator is defined in *Control.Monad* as

```
(≫=) :: Monad m ⇒ (a → m b) → (b → m c) → (a → m c)
f ≫ g = λx → f x ≫= g
```

and implements Kleisli composition for a monad, here, the *Maybe* monad.

We want to abstract over functors from *Iso* to *Hask* to specify our $\langle\$\rangle$ operator which works for both *Parser* and *Printer*, but Haskell does only provide the *Functor* typeclass for functors from *Hask* to *Hask*. To capture our variant of functors, we introduce the *IsoFunctor* typeclass.

```
class IsoFunctor f where
  (⟨$⟩) :: Iso α β → (f α → f β)
```

The type class *IsoFunctor* and its $\langle\$\rangle$ method forms the first component of the language interface of our language of syntax descriptions.

3.2 Uncurried application and the $\langle\!\circledast\!\rangle$ combinator

The $\langle\!\circledast\!\rangle$ combinator for *Parser* is used to combine a *Parser* $(\alpha \to \beta)$ and a *Parser* α into a *Parser* β. The behavior of the $(p \langle\!\circledast\!\rangle q)$ parser is to first use p to parse a function of type $\alpha \to \beta$, then use q to parse a value of type α, then apply the function to the value, and finally return the result of type β.

```
(⟨⊛⟩) :: Parser (α → β) → (Parser α → Parser β)
```

The *Applicative* type class specifies such a $\langle\!\circledast\!\rangle$ operator for functors from *Hask* to *Hask*, i.e. instances of the *Functor* type class. But since our language of syntax descriptions is based on functors from *Iso* to *Hask*, we cannot use the standard *Applicative* type class as a component in our language interface. We would like to generalize the notion of applicative functors to functors from *Iso* to *Hask*.

```
class IsoApplicative f where
  (⟨⊛⟩) :: f (Iso α β) → (f α → f β)
```

Unfortunately, this version of $\langle\!\circledast\!\rangle$ cannot be implemented by *Printer*. Expanding the definition of *Printer*, we see that we would have to implement the following function.

$(\mathbin{\diamondsuit\!\!*})::(\mathit{Iso}\ \alpha\ \beta \to \mathit{Doc}) \to (\alpha \to \mathit{Doc}) \to (\beta \to \mathit{Doc})$
$(\mathbin{\diamondsuit\!\!*})\ p\ q\ b = ...$

We have b of type β and want to produce a document. Our only means of producing documents would be to call p or q, but neither of them accepts β. We furthermore have no isomorphism $\mathit{Iso}\ \alpha\ \beta$ available to convert b into a value of type α. Instead, we could print such an isomorphism, if only we had one.

Since *Printer* does not support the applicative $\mathbin{\diamondsuit\!\!*}$ combinator, we have to specify an alternative version of $\mathbin{\diamondsuit\!\!*}$ to combine two syntax descriptions side by side. Note that in our *parseMany* code, $\mathbin{\diamondsuit\!\!*}$ is always used together with $\mathbin{\diamondsuit\!\!\$}$ in an expression like the following.

$f \mathbin{\diamondsuit\!\!\$} p1 \mathbin{\diamondsuit\!\!*} ... \mathbin{\diamondsuit\!\!*} pn$

In this restricted usage, the role of $\mathbin{\diamondsuit\!\!*}$ is simply to support curried function application. We do support the $(f \mathbin{\diamondsuit\!\!\$} p1 \mathbin{\diamondsuit\!\!*} ... \mathbin{\diamondsuit\!\!*} pn)$ pattern through a different definition of $\mathbin{\diamondsuit\!\!\$}$. Our operator $\mathbin{\diamondsuit\!\!*}$ will not be used to express curried function application, but it will be used to express uncurried function application. Therefore, our $\mathbin{\diamondsuit\!\!*}$ has the following type.

$(\mathbin{\diamondsuit\!\!*})::\mathit{Printer}\ \alpha \to \mathit{Printer}\ \beta \to \mathit{Printer}\ (\alpha,\beta)$

This $\mathbin{\diamondsuit\!\!*}$ operator is supported by both printing and parsing. Printing with $(p \mathbin{\diamondsuit\!\!*} q)$ means printing the first component of the input with p, and the second component with q. And parsing with $(p \mathbin{\diamondsuit\!\!*} q)$ means parsing a first value with p, then a second value with q, and returning these values as components of a tuple.

The applicative version of $\mathbin{\diamondsuit\!\!*}$ supports the pattern

$(f \mathbin{\diamondsuit\!\!\$} p1 \mathbin{\diamondsuit\!\!*} ... \mathbin{\diamondsuit\!\!*} pn)$

as left-associative nested application of a curried function

$(((f \mathbin{\diamondsuit\!\!\$} p1) \mathbin{\diamondsuit\!\!*} ...) \mathbin{\diamondsuit\!\!*} pn),$

whereas our $\mathbin{\diamondsuit\!\!*}$ supports the same pattern as right-associative tupling and application of an uncurried function

$(f \mathbin{\diamondsuit\!\!\$} (p1 \mathbin{\diamondsuit\!\!*} (... \mathbin{\diamondsuit\!\!*} pn))).$

by appropriately changing the associativity and relative priority of the $\mathbin{\diamondsuit\!\!\$}$ and $\mathbin{\diamondsuit\!\!*}$ operators.

For normal functors, the pairing variant and the currying variant of $\mathbin{\diamondsuit\!\!*}$ are inter-derivable (McBride and Paterson 2008), but for *Iso* functors it makes a real difference.

We abstract over the class of functors supporting $\mathbin{\diamondsuit\!\!*}$ by introducing the *ProductFunctor* typeclass.

class *ProductFunctor f* **where**
  $(\mathbin{\diamondsuit\!\!*})::f\ \alpha \to f\ \beta \to f\ (\alpha,\beta)$

ProductFunctor does not have any superclasses, so that it can be used together with the new *IsoFunctor* type class or together with the ordinary *Functor* type class. *ProductFunctor* and its $\mathbin{\diamondsuit\!\!*}$ method form the second component of the language interface for our language of syntax descriptions.

3.3 Expressing choices and the $\mathbin{\diamondsuit\!\!|}$ operator

In the *parseMany* code shown above, alternatives are expressed using the $\mathbin{\diamondsuit\!\!|}$ combinator of type $\mathit{Parser}\ \alpha \to \mathit{Parser}\ \alpha \to \mathit{Parser}\ \alpha$. This combinator is used to compose parsers for the variants of a datatype into a parser for the full datatype. The $\mathbin{\diamondsuit\!\!|}$ combinator has been generalized in the standard *Alternative* type class. But *Alternative* declares a superclass constraint to *Applicative*, which is not suitable for syntax descriptions. We therefore need a version of *Alternative* which is superclass independent.

class *Alternative f* **where**
  $(\mathbin{\diamondsuit\!\!|})::f\ \alpha \to f\ \alpha \to f\ \alpha$
  $\mathit{empty}::f\ \alpha$

This class can be readily instantiated with *Parser*. The $\mathbin{\diamondsuit\!\!|}$ combinator will typically try both parsers, implementing a backtracking semantics. The *empty* function is a parser which always fails. For *Printer*, $\mathbin{\diamondsuit\!\!|}$ will try to print with the left printer. If this is not successful, it will print with the right printer instead. The *empty* function is the printer which always fails.

3.4 The class of syntax descriptions

So far, we have provided the combinators $\mathbin{\diamondsuit\!\!\$}$, $\mathbin{\diamondsuit\!\!*}$ and $\mathbin{\diamondsuit\!\!|}$ to combine smaller syntax descriptions into larger syntax descriptions, but we still have to provide a means to describe elementary syntax descriptions. We use two elementary syntax descriptions: *token* and *pure*. The *token* function relates each character with itself. The *pure* function takes an α and the resulting parser/printer will relate the empty string with that α value. A *pure x* parser returns x without consuming any input, while a *pure x* printer silently discards values equal to x. The $\mathit{Eq}\ \alpha$ constraint on the type *pure* is needed so that a printer can check a value to be discarded for equality to x.

Together with the typeclasses already introduced, these functions are sufficient to state the language interface that unifies parsing and prettyprinting. The type class *Syntax* pulls in the $\mathbin{\diamondsuit\!\!\$}$, $\mathbin{\diamondsuit\!\!*}$, and $\mathbin{\diamondsuit\!\!|}$ combinators via superclass constraints, and adds the *pure* and *token* functions.

class $(\mathit{IsoFunctor}\ \delta, \mathit{ProductFunctor}\ \delta, \mathit{Alternative}\ \delta)$
  $\Rightarrow \mathit{Syntax}\ \delta$ **where**
    -- $(\mathbin{\diamondsuit\!\!\$})::\mathit{Iso}\ \alpha\ \beta \to \delta\ \alpha \to \delta\ \beta$
    -- $(\mathbin{\diamondsuit\!\!*})::\delta\ \alpha \to \delta\ \beta \to \delta\ (\alpha,\beta)$
    -- $(\mathbin{\diamondsuit\!\!|})::\delta\ \alpha \to \delta\ \alpha \to \delta\ \alpha$
    -- $\mathit{empty}::\delta\ \alpha$
  $\mathit{pure}::\mathit{Eq}\ \alpha \Rightarrow \alpha \to \delta\ \alpha$
  $\mathit{token}::\delta\ \mathit{Char}$

With this typeclass, we can now state a function *many* which unifies *parseMany* and *prettyMany* as follows:

$\mathit{many}::\mathit{Syntax}\ \delta \Rightarrow \delta\ \alpha \to \delta\ [\alpha]$
$\mathit{many}\ p$
  $= \mathit{nil} \mathbin{\diamondsuit\!\!\$} \mathit{pure}\ ()$
  $\mathbin{\diamondsuit\!\!|}\ \mathit{cons} \mathbin{\diamondsuit\!\!\$} p$
    $\mathbin{\diamondsuit\!\!*} \mathit{many}\ p$

This implementation looks essentially like the implementation of *parseMany*, but instead of constructors *Nil* and *Cons*, we use partial isomorpisms *nil* and *cons*. Note that we do not have to use *const nil*, because our partial isomorphisms treat constructors without arguments like constructors with a single () argument. Unlike the code for *parseMany*, which was usable only for parsing, this implementation of *many* uses the polymorphically embedded language of syntax descriptions, which can be instantiated for both parsing and printing.

4. Implementing syntax descriptions

In the last section, we derived a language interface for syntax descriptions to unify parsers and printers *syntactically*. For example, at the end of the section, we have shown how to write *parseMany* and *printMany* as a single function *many*. To support our claim that *many* really implements both *parseMany* and *printMany semantically*, we now have to implement the language of syntax descriptions twice: First for parsing and then for printing.

An implementation of the language of syntax descriptions consists of a parametric data type with instances for *IsoFunctor*,

ProductFunctor, *Alternative* and *Syntax*. In this paper, we present rather inefficient proof-of-concept implementations for both parsing and pretty printing, but appropriate instance declarations could add more efficient implementations (see Sec. 4.4 for a discussion).

4.1 Implementing parsing

In our implementation, a *Parser* is a function from input text to a list of pairs of results and remaining text.

```
newtype Parser α
  = Parser (String → [(α, String)])
```

A value of type *Parser* α can be used to parse an α value from a string by applying the function and filtering out results where the remaining text is not empty. The *parse* function returns a list of α's because our parser implementation supports nondeterminism through the list monad, and therefore can return several possible results.

```
parse :: Parser α → String → [α]
parse (Parser p) s = [x | (x, "") ← p s]
```

We now provide the necessary instances to use *Parser* as an implementation of syntax descriptions. A parser of the form *iso* $\langle\$\rangle$ *p* is implemented by mapping *apply iso* over the first component of the value-text-tuples in the returned list, and silently ignoring elements where *apply iso* returns *Nothing*. Note that failed pattern matching (in this case: *Just y*) in a list comprehension is filtering out that element.

```
instance IsoFunctor Parser where
  iso <$> Parser p
    = Parser (λs → [ (y, s')
                   | (x, s') ← p s
                   , Just y ← [apply iso x]])
```

A parser of the form $(p \langle * \rangle q)$ is implemented by threading the remaining text through the applications of *p* and *q*, and tupling the resulting values.

```
instance ProductFunctor Parser where
  Parser p <*> Parser q
    = Parser (λs → [ ((x, y), s'')
                   | (x, s')  ← p s
                   , (y, s'') ← q s'])
```

A parser of the form $(p \langle | \rangle q)$ is implemented by concatenating the result lists of the two parsers. The empty parser returns no results.

```
instance Alternative Parser where
  Parser p <|> Parser q
    = Parser (λs → p s ++ q s)
  empty = Parser (λs → [])
```

Finally, the elementary parsers *pure* and *token* are implemented by returning the appropriate singleton lists. *pure x* always succeeds returning *x* and the full text as remaining text. *token* fails if there is no more input text, and returns the first character of the input text otherwise.

```
instance Syntax Parser where
  pure x = Parser (λs → [(x, s)])
  token  = Parser f where
    f []     = []
    f (t:ts) = [(t, ts)]
```

This concludes our proof-of-concept implementation of the language interface of syntax descriptions with parsers.

4.2 Implementing printing

Our implementations of pretty printers are partial functions from values to text, modelled using the *Maybe* type constructor.

```
newtype Printer α = Printer (α → Maybe String)
```

This is different from the preliminary *Printer* type we presented in Sec. 3, where we used *Doc* instead of *String*, and did not mention the *Maybe*. Here, we are using *String* because we are only interested in a simple implementation, and do not want to adapt an existing pretty printing library with a first-order *Doc* type to our interface. We are dealing with partial functions because a *Printer* α should represent a pretty printer for a subset of the extension of α. We then want to use the $\langle | \rangle$ combinator to combine pretty printers for several subsets into a pretty printer of all of α. This allows us to specify syntax descriptions for algebraic data types one constructor at a time, instead of having to specify a monolithic syntax description for the full data type at once.

A value of type *Printer* α can be used to pretty print a value of type α simply by applying the function.

```
print :: Printer α → α → Maybe String
print (Printer p) x = p x
```

We now provide the necessary instances to use *Printer* as an implementation of syntax descriptions. A printer of the form *iso* $\langle\$\rangle$ *p* is implemented by converting the value to be printed with *unapply iso* before printing it with *p*, silently failing if *unapply iso* returns *Nothing*.

```
instance IsoFunctor Printer where
  iso <$> Printer p
    = Printer (λb → unapply iso b >>= p)
```

A printer of the form $(p \langle * \rangle q)$ is implemented by monadically lifting the string concatenation operator $+\!\!+$ over the results of printing the first component of the value to be printed with *p*, and the second component with *q*. This returns *Nothing* if one or both of *p* or *q* return *Nothing*, and returns the concatenated results of *p* and *q* otherwise.

```
instance ProductFunctor Printer where
  Printer p <*> Printer q
    = Printer (λ(x, y) → liftM2 (++) (p x) (q y))
```

A printer of the form $p \langle | \rangle q$ is implemented by using *p* if it succeeds, and using *q* otherwise. The empty printer always fails.

```
instance Alternative Printer where
  Printer p <|> Printer q
    = Printer (λs → mplus (p s) (q s))
  empty = Printer (λs → Nothing)
```

A printer of the form *pure x* is implemented by comparing the value to be printed with *x*, returning the empty string if it matches, and *Nothing* otherwise. Finally, *token* is implemented by always returning the singleton string consisting just of the token to be printed.

```
instance Syntax Printer where
  pure x = Printer (λy → if x ≡ y
                         then Just ""
                         else Nothing)
  token = Printer (λt → Just [t])
```

This concludes our proof-of-concept implementation of the language interface of syntax descriptions with printers. We have shown that it is possible to implement syntax descriptions with both parsers and printers.

4.3 What syntax descriptions mean

A syntax description denotes a relation between abstract and concrete syntax. We can represent such a relation as its graph, i.e., as a list of pairs of abstract and concrete values. Since our interface design allows us to add a new meaning to the interface by corresponding instance declarations, we formulate our semantics as a set of type class instances in Haskell, too. This instance declaration is not useful as an executable implementation because it will generate and concatenate infinite lists. Rather, it should be read as a declarative denotational semantics.

An abstract value in this relation is of some type α, while a concrete value is of type *String*.

```
data Rel α = Rel [(α, String)]
```

To provide a semantics for syntax descriptions, we have to implement the methods of *Syntax*. The $\langle\$\rangle$ operator applies the first component of the partial isomorphism to the abstract values, filtering out abstract values which are not in the domain of the partial isomorphism.

```
instance IsoFunctor Rel where
  iso f g <$> Rel graph
    = Rel [(a', c)
      | (a, c) ← graph
      , Just a' ← return (f a)]
```

The $\langle*\rangle$ operator returns the cross product of the graphs of its arguments, tupling the abstract values, but concatenating the concrete values.

```
instance ProductFunctor Rel where
  Rel graph <*> Rel graph'
    = Rel [((a, a'), c ++ c')
      | (a, c) ← graph
      , (a', c') ← graph']
```

The $\langle|\rangle$ operator returns the union of the graphs, and *empty* is the empty relation, i.e. the empty graph.

```
instance Alternative Rel where
  Rel graph <|> Rel graph'
    = Rel (graph ++ graph')
  empty = Rel []
```

Finally, *pure x* is the singleton graph relating x to the empty string, and *token* relates all characters to themselves.

```
instance Syntax Rel where
  pure x = Rel [(x, "")]
  token = Rel [(t, [t]) | t ← characters]
    where characters = [minBound .. maxBound]
```

This denotational semantics of syntax descriptions can be used to describe the behavior of printing and parsing in a declarative way. Printing an abstract value x according to a syntax description d means to produce a string s so that (x,s) is an element of the graph of d. Parsing a concrete string s according to a syntax description d means to produce an abstract value x so that (x,s) is an element of the graph of d. Both printing and parsing are under-specified here, because it is not specified *how* to choose the s or the x to produce.

Understanding syntax descriptions as relations also allows us to compare our approach to logic programming, where relations (defined via predicates) can also theoretically be used "both ways", since each variable in a logic rule can operationally be used as both input and output. In practice, however, most predicates work only in one direction, because "unpure" features (such as cuts or primitive arithmetic) and the search strategy of the solver often require a clear designation of input and output variables.

Using a syntax description in both ways requires more work than in logic programming, since explicit instance declarations for each direction have to be specified. They have to be specified once only, though, and then inversion in that direction works for any syntax description. The instance declarations also provide more control than the fixed DFS strategy of typical logic solvers, which means that in contrast to logic programming invertibility can actually be made to work in practice.

4.4 Adapting existing libraries

The implementations of syntax descriptions for parsing and printing in the previous subsections are proofs-of-concept, lacking many features available in "real-world" parsers and pretty printers. The parser implementation also suffers from an exponential worst-case complexity and a space leak due to unlimited backtracking, which limits its applicability to large inputs.

The former problem is a problem of *any* interface design. We could add more features to our interfaces, but this would also limit the number of parsers and pretty printers that can implement this interface. This is, for example, also a problem of the existing designs of the *Applicative* and *Alternative* type classes in Haskell.

We propose two different strategies to deal with this problem. One strategy is to extend the interfaces via type class subclassing and then write additional instance declarations for more sophisticated parsers and pretty printers. Another strategy is to split a grammar specification into those parts that can be expressed with the *Syntax* interface and its derived operations alone, and those parts that are specific to a fixed parser or pretty-printer implementation. In this case, the automatic inversion still works for the first part, and manual intervention is necessary to invert the second part.

The latter problem can be solved by instantiating *Syntax* for more advanced parser combinator and/or pretty printer approaches, such as …, which exhibit better time or memory behavior. However, such existing parser/pretty printer libraries may not match the semantics expected by syntax descriptions. We have identified two categories of such semantic mismatches.

Firstly, an existing library may not provide combinators with the exact semantics of the combinators in the language interface for syntax descriptions, but only combinators with a similar semantics. For example, Parsec provides a $\langle|\rangle$ combinator, but its semantics implements predictive parsing with a look ahead of 1, whereas our implementation supports unlimited backtracking. This means that with Parsec, $p \langle|\rangle q$ may fail, even if q would succeed, whereas the syntax description $p \langle|\rangle q$ should not be empty if q is nonempty. If one would use the Parsec $\langle|\rangle$ to implement the *Syntax* $\langle|\rangle$, then syntax descriptions have to be written with the Parsec semantics in mind.

The design of an interface that is rich enough to specify efficient and sophisticated parsers and pretty printers without committing to a particular implementation is in our point of view an open research (and standardization) question and part of our future work. However, our design of syntax descriptions can serve as a common framework for such interfaces which combine several parsing and pretty printing libraries, similar to how the *Applicative* and *Alternative* classes provide a common framework for parsing.

5. Programming with partial isomorphisms

Since our language of syntax descriptions is based upon the notion of partial isomorphisms, programming with partial isomorphisms is an important part of programming with syntax descriptions. In this section, we evaluate whether programming with partial isomorpisms is practical. The abstractions developed in this section are reused in the next section as the basis for some derived syntax combinators.

Every partial isomorphism expressible in Haskell can be written by implementing both directions of the isomorphism independently, and combining them using the *Iso* constructor. However, this approach is neither safe nor convenient. It is not safe because it is not checked that the two directions are really inverse to each other, and it is not convenient because one has to essentially program the same function twice, although in two different directions. We call such a partial isomorphism implemented directly with *Iso* a *primitive* partial isomorphism, and we hope to mostly avoid having to define such primitives.

Instead of defining every partial isomorphism of interest as a primitive, we provide elementary partial isomorphisms for the constructors of algebraic datatypes, and an algebra of partial isomorphism combinators which can be used to implement more complex partial isomorphisms. We call such a partial isomorphisms implemented in terms of a small set of primitives a *derived* partial isomorphism, and we hope to implement most partial isomorphisms of interest as derived isomorphisms.

5.1 An algebra of partial isomorphisms

An algebra of partial isomorphisms can be implemented using primitives. The specification and implementation of a full algebra of partial isomorphisms is beyond the scope of this paper. However, we present sample elementary partial isomorphisms and partial isomorphism combinators to show how the development of such an algebra could reflect well-known type isomorphism and categorical constructs.

We have already seen the implementation of the $\circ$ and *id* combinators in the *Category* instance declaration in Sec. 3.1.

```
id :: Iso α α
(∘) :: Iso β γ → Iso α β → Iso α γ
```

Other categorical constructions can be reified as partial isomorphisms as well. For example, the product type constructor $(,)$ is a bifunctor from $Iso \times Iso$ to *Iso*, so that we have the bifunctorial map $\times$ which allows two separate isomorphisms to work on the two components of a tuple.

```
(×) :: Iso α β → Iso γ δ → Iso (α, γ) (β, δ)
i × j = Iso f g where
  f (a, b) = liftM2 (,) (apply i a) (apply j b)
  g (c, d) = liftM2 (,) (unapply i c) (unapply j d)
```

We reify some more facts about product and sum types as partial isomorphisms. Nested products associate.

```
associate :: Iso (α, (β, γ)) ((α, β), γ)
associate = Iso f g where
  f (a, (b, c)) = Just ((a, b), c)
  g ((a, b), c) = Just (a, (b, c))
```

Products commute.

```
commute :: Iso (α, β) (β, α)
commute = Iso f f where
  f (a, b) = Just (b, a)
```

$()$ is the unit element for products.

```
unit :: Iso α (α, ())
unit = Iso f g where
  f a = Just (a, ())
  g (a, ()) = Just a
```

element x is the partial isomorphism between $()$ and the singleton set which contains just *x*. Note that this is an isomorphism only up to the equivalence class defined by the *Eq* instance, as discussed in Sec. 3.1.

```
element :: Eq α ⇒ α → Iso () α
element x = Iso
  (λa → Just x)
  (λb → if x ≡ b then Just () else Nothing)
```

For a predicate *p*, *subset p* is the identity isomorphism restricted to elements matching the predicate.

```
subset :: (α → Bool) → Iso α α
subset p = Iso f f where
  f x | p x = Just x | otherwise = Nothing
```

Numerous more partial isomorphisms primitives could be defined, reflecting other categorical constructions or type isomorphisms. However, the primitives defined so far are sufficient for the examples in this paper. Therefore, the following subsections are devoted to the derivation of a non-trivial partial isomorphism using the primitives implemented so far.

5.2 Folding as a small-step abstract machine

We will need left-associative folding resp. unfolding as a partial isomorphism in the implementation of left-associative binary operators. Instead of defining folding and unfolding as primitives, we show how it can be defined as a derived isomorphism in terms of the already defined primitives.

To see how to implement folding and unfolding in a single program, we consider the straightforward implementation of *foldl* from the standard Haskell prelude.

```
foldl :: (α → β → α) → α → [β] → α
foldl f z [] = z
foldl f z (x : xs) = foldl f (f z x) xs
```

Since partial isomorphisms do not support currying very well, we uncurry most of the functions.

```
foldl :: ((α, β) → α) → (α, [β]) → α
foldl f (z, []) = z
foldl f (z, x : xs) = foldl f (f (z, x), xs)
```

This implementation of *foldl* is a big-step abstract machine with state type $(\alpha, [\beta])$, calling itself in tail-position and computing the result in a monolithic way. We want to break this monolithic computation into many small steps by transforming *foldl* into a small-step abstract machine. A big-step abstract machines can be transformed into small-step abstract machines by a general-purpose program transformation called *light-weight fission* (see Danvy 2008, for this and related transformations on abstract machines).

We decompose *foldl* into a step function and a driver. *step* computes a single step of *foldl*'s overall computation, and *driver* calls *step* repeatedly. *step* is actually a partial function, represented with a *Maybe* type. If no more computation steps are needed, *step* returns *Nothing*, so that *driver* stops calling *step* and returns the current state. *driver* is implemented independently from *foldl*.

```
driver :: (α → Maybe α) → (α → α)
driver step state
  = case step state of
      Just state' → driver step state'
      Nothing     → state
```

Since we are only interested in the α part of the final state, *foldl* drops the second component of the state after running the abstract machine.

```
foldl :: ((α, β) → α) → (α, [β]) → α
foldl f = fst ∘ driver step where
  step (z, []) = Nothing
  step (z, x : xs) = Just (f (z, x), xs)
```

We have transformed *foldl* into a small-step abstract machine to break its monolithic computation into a series of smaller steps. The next step towards the implementation of *foldl* as a partial isomorphism will be to enable this abstract machine to run backwards.

5.3 Running the abstract machine backwards

To convert *foldl* into a partial isomorphism combinator of type $Iso\ (\alpha,\beta)\ \alpha \to Iso\ (\alpha,[\beta])\ \alpha$, we have to convert both *driver* and *step* into partial isomorphisms. We could then run *foldl* forwards by composing a sequence of steps, and we could run *foldl* backwards by composing a reversed sequence of inverted steps.

The partial isomorphism analogue to *driver* is implemented as a primitive in terms of *driver*. We call it *iterate*, since it captures the iterated application (resp. unapplication) of a function.

```
iterate :: Iso α α → Iso α α
iterate step = Iso f g where
  f = Just ∘ driver (apply step)
  g = Just ∘ driver (unapply step)
```

Note that the type of *iterate* does not mention *Maybe* anymore. Instead, the partial isomorphism *step* is applied (resp. unapplied) until it fails, showing once more the usefulness of *partial* isomorphisms.

It remains to implement the parametric partial isomorphism *step* in terms of the primitives introduced earlier in this subsection. It has the following type.

$$Iso\ (\alpha,\beta)\ \alpha \to Iso\ (\alpha,[\beta])\ (\alpha,[\beta])$$

We start with a value of type $(\alpha,[\beta])$, and want to use the partial isomorphism i we have taken as an argument. Since i takes a single α, we have to destruct the $[\beta]$ into a first element β and the remaining elements $[\beta]$. The α should not be changed for now. The destruction is performed by the inverse of the *cons* partial isomorphism, and $(\times)$ is used to apply it to the second component of the input.

$$id \times inverse\ cons :: Iso\ (\alpha,[\beta])\ (\alpha,(\beta,[\beta]))$$

We can now restructure our value by using the fact that products are associative.

$$associate :: Iso\ (\alpha,(\beta,[\beta]))\ ((\alpha,\beta),[\beta])$$

The partial isomorphism i is now applicable to the first component of the tuple.

$$i \times id :: Iso\ ((\alpha,\beta),[\beta])\ (\alpha,[\beta])$$

We arrive at a value of type $(\alpha,[\beta])$, and are done. These snippets can be composed with $\circ$ to implement *step* as a partial isomorphism.

```
step i = (i × id)
  ∘ associate
  ∘ (id × inverse cons)
```

We can now implement *foldl* in terms of *iterate* and *step*. In the version of *foldl* as a small-step abstract machine, we used *fst* to return only the first component of the tuple, ignoring the second component. In this reversible small-step abstract machine, we are not allowed to just ignore information. However, we know from the definition of *step*, that the second component of the abstract machine's state will always contain $[\,]$ after the machine has been run. Therefore, we can use the inverse of the *nil* partial isomorphism to deconstruct that $[\,]$ into $()$, which can be safely ignored using the *unit* primitive.

```
foldl :: Iso (α,β) α → Iso (α,[β]) α
foldl i = inverse unit
  ∘ (id × inverse nil)
  ∘ iterate (step i)
```

As a partial isomorphism, this definition of *foldl* is invertible. It can be applied as left-associative folding, but it can also be *un*applied as left-associative *un*folding. By rewriting the step function of a small-step abstract machine to use the combinators for partial isomorphisms, we have effectively inverted the implementation of *foldl* into an implementation of *unfoldl*.

In this section, we have evaluated the practicability of programming with partial isomorphisms. We have seen that the automatic generation of partial isomorphisms for constructors of algebraic datatypes together with a small set of primitives suffices to derive an advanced combinator like left-associative folding, which can then be automatically inverted to yield left-associative unfolding.

6. Describing the syntax of a language

Using the partial isomorphism combinators from the last section, we can now evaluate our approach to syntax descriptions by applying it to an example of a small formal language which features keywords and identifiers, nested infix operators with priorities, and flexible whitespace handling.

6.1 Derived operations

Before introducing our example language, we implement some general-purpose combinators, mostly adopted from the usual parser and pretty-printer combinators.

We can define the dual of the $\oplus$ combinator, using the following injections into $Either\ \alpha\ \beta$.

```
$(defineIsomorphisms ''Either)

(⊕) :: Syntax δ ⇒ δ α → δ β → δ (Either α β)
p ⊕ q = (left $ p) <|> (right $ q)
```

The $\oplus$ operator can be used as an alternative to $\langle|\rangle$ when describing the concrete syntax of algebraic data types. Instead of providing a partial isomorphism for every constructor of the algebraic data type, and using $\langle|\rangle$ to combine the branches for the constructors, we provide a single partial isomorphism between the data type and its sum-of-product form written with $(,)$ and *Either*, and combine the branches for the constructors with $\oplus$.

The *many* combinator shown earlier can be implemented in this style as follows.

```
many' :: Syntax δ ⇒ δ α → δ [α]
many' p
  = listCases $ (text "" ⊕ p ⊛ many' p)
```

The partial isomorphism *listCases* can be implemented as follows, or the Template Haskell code in Appendix A could be extended to generate this kind of partial isomorphisms as well.

```
listCases :: Iso (Either () (α,[α])) [α]
listCases = Iso f g
  where
    f (Left ()) = Just []
    f (Right (x,xs)) = Just (x:xs)
    g [] = Just (Left ())
    g (x:xs) = Just (Right (x,xs))
```

text parses/prints a fixed text and consumes/produces a unit value.

```
text :: Syntax δ ⇒ String → δ ()
text [] = pure ()
text (c:cs) = inverse (element ((),()))
  $ (inverse (element c) $ token)
  ⊛ text cs
```

The following two operators are variants of $\circledast$ that ignore their left or right result. In contrast to their counterparts derived from the

Applicative class, the ignored parts have type $\delta\ ()$ rather than $\delta\ \beta$ because otherwise information relevant for pretty-printing would be lost.

$(\ \ast\!\!\rangle\ ) :: Syntax\ \delta \Rightarrow \delta\ () \to \delta\ \alpha \to \delta\ \alpha$
$p\ \ast\!\!\rangle\ q = inverse\ unit \circ commute \diamond\!\!\!\$\!\!\!\diamond\ p \diamond\!\!\!\ast\!\!\!\diamond\ q$
$(\ \langle\!\!\ast\ ) :: Syntax\ \delta \Rightarrow \delta\ \alpha \to \delta\ () \to \delta\ \alpha$
$p\ \langle\!\!\ast\ q = inverse\ unit \diamond\!\!\!\$\!\!\!\diamond\ p \diamond\!\!\!\ast\!\!\!\diamond\ q$

The *between* function combines these operators in the obvious way.

$between :: Syntax\ \delta \Rightarrow \delta\ () \to \delta\ () \to \delta\ \alpha \to \delta\ \alpha$
$between\ p\ q\ r = p\ \ast\!\!\rangle\ r\ \langle\!\!\ast\ q$

Even sophisticated combinators like *chainl1* can be directly implemented in terms of syntax descriptions and appropriate partial isomorphisms. The *chainl1* combinator is used to parse a left-associative chain of infix operators. It is implemented using *foldl* from Sec. 5.3 and *many* from 3.4.

```
chainl1
  :: Syntax δ ⇒ δ α → δ β → Iso (α,(β,α)) α → δ α
chainl1 arg op f
  = foldl f ◈$◈ arg ◈*◈ many (op ◈*◈ arg)
```

We have implemented some syntax description combinators along the lines of the combinators well-known from parser combinator libraries. We will now use these combinators to describe the syntax of a small language.

6.2 Abstract Syntax

The abstract syntax of the example language is encoded with abstract data types.

```
data Expression
  = Variable String
  | Literal Integer
  | BinOp Expression Operator Expression
  | IfZero Expression Expression Expression
  deriving (Show, Eq)
data Operator
  = AddOp
  | MulOp
  deriving (Show, Eq)
```

The Template Haskell macro *defineIsomorphisms* is used to generate partial isomorphisms for the data constructors.

```
$(defineIsomorphisms "Expression)
$(defineIsomorphisms "Operator)
```

6.3 Expressing whitespace

Parsers and pretty printers treat whitespace differently. Parsers specify where whitespace is allowed or required to occur, while pretty printers specify how much whitespace is to be inserted at these locations. To account for these different roles of whitespace, the following three syntax descriptions provide fine-grained control over where whitespace is allowed, desired or required to occur.

- *skipSpace* marks a position where whitespace is allowed to occur. It accepts arbitrary space while parsing, and produces no space while printing.
- *optSpace* marks a position where whitespace is desired to occur. It accepts arbitrary space while parsing, and produces a single space character while printing.
- *sepSpace* marks a position where whitespace is required to occur. It requires one or more space characters while parsing, and produces a single space character while printing.

```
skipSpace, optSpace, sepSpace :: Syntax δ ⇒ δ ()
skipSpace = ignore []    ◈$◈ many (text " " )
optSpace  = ignore [()] ◈$◈ many (text " " )
sepSpace  = text " " ⟨*  skipSpace
```

```
ignore :: α → Iso α ()
ignore x = Iso f g where
  f _ = Just ()
  g () = Just x
```

ignore is again not a strict partial isomorphism, because all values of α are mapped to ().

6.4 Syntax descriptions

The first character of an identifier is a letter, the remaining characters are letters or digits. Keywords are excluded.

```
keywords = ["ifzero", "else"]

letter, digit :: Syntax δ ⇒ δ Char
letter = subset isLetter ◈$◈ token
digit  = subset isDigit ◈$◈ token

identifier
  = subset (∉ keywords) ∘ cons ◈$◈
    letter ◈*◈ many (letter ◈<>◈ digit)
```

Keywords are literal texts but not identifiers.

```
keyword :: Syntax δ ⇒ String → δ ()
keyword s = inverse right ◈$◈ (identifier ◈+◈ text s)
```

Integer literals are sequences of digits, processed by read resp. show.

```
integer :: Syntax δ ⇒ δ Integer
integer = Iso read' show' ◈$◈ many digit where
  read' s = case [x | (x, "") ← reads s] of
              [] → Nothing
              (x:_) → Just x
  show' x = Just (show x)
```

A parenthesized expressions is an expression between parentheses.

```
parens = between (text "(") (text ")")
```

The syntax descriptions *ops* handles operators of arbitrary priorities. The priorities are handled further below.

```
ops = mulOp ◈$◈ text "*"
    ◈<>◈ addOp ◈$◈ text "+"
```

We allow optional spaces around operators.

```
spacedOps = between optSpace optSpace ops
```

The priorities of the operators are defined in this function.

```
priority :: Operator → Integer
priority MulOp = 1
priority AddOp = 2
```

Finally, we can define the *expression* syntax description.

```
expression = exp 2 where

exp 0 = literal    ◈$◈ integer
      ◈<>◈ variable ◈$◈ identifier
      ◈<>◈ ifZero   ◈$◈ ifzero
      ◈<>◈ parens (skipSpace ⟩*  expression ⟨*  skipSpace)
```

```
exp 1 = chainl1 (exp 0) spacedOps (binOpPrio 1)
exp 2 = chainl1 (exp 1) spacedOps (binOpPrio 2)

ifzero = keyword "ifzero"
       *> optSpace  *> parens (expression)
       <*> optSpace  *> parens (expression)
       <*> optSpace  *> keyword "else"
       *> optSpace  *> parens (expression)
binOpPrio n
  = binOp ∘ subset (λ(x,(op,y)) → priority op ≡ n)
```

This syntax description is correctly processing binary operators according to their priority during both parsing and printing. Similar to the standard idiom for expression grammars with infix operators, the description of *expression* is layered into several *exp i* descriptions, one for each priority level. The syntax description combinator *chainl1* parses a left-recursive tree of expressions, separated by infix operators. Note that the syntax descriptions *exp* 1 to *exp* 2 both use the same syntax descriptions *ops* which describes all operators, not just the operators of a specific priority. Instead, the correct operators are selected by the *binOpPrio n* partial isomorphisms. The partial isomorphism *binOpPrio n* is a subrelation of *binOp* which only accepts operators of the priority level *n*.

While parsing a high-priority expressions, the partial isomorphism will reject low-priority operators, so that the parser stops processing the high-priority subexpression and backtracks to continue a surrounding lower-priority expression. When the parser encounters a set of parentheses, it allows low-priority expressions again inside.

Similarly, during printing a high-priority expression, the partial isomorphism will reject low-priority operators, so that the printer continues to the description of *exp* 0 and inserts a matching set of parentheses.

All taken together, the partial isomorphisms *binOpPrio n* not only control the processing of operator priorities for both printing and parsing, but also ensure that parentheses are printed exactly where they are needed so that the printer output can be correctly parsed again. This way, correct round trip behavior is automatically guaranteed.

The following evaluation shows that operator priorities are respected while parsing.

```
> parse expression "ifzero (2+3*4) (5) else (6)"
[IfZero (BinOp (Literal 2) AddOp
           (BinOp (Literal 3) MulOp (Literal 4)))
        (Literal 5)
        (Literal 6)]
```

And this evaluation shows that needed parentheses are inserted during printing.

```
> print expression
    (BinOp (BinOp (Literal 7) AddOp
                  (Literal 8)) MulOp
           (Literal 9))
Just "(7 + 8) * 9"
```

By implementing whitespace handling and associativity and priorities for infix operators, we have shown how to implement two nontrivial aspects of syntax descriptions which occur in existing parsers and pretty printers for formal languages. We have shown how to implement well-known combinators like *between* and *chainl1* in our framework, which enabled us to write the syntax descriptions in a style which closely resembles how one can program with monadic or applicative parser combinator libraries.

7. Related and Future Work

7.1 Parsing and Pretty Printing

Parser combinator libraries in Haskell are often based on a monadic interface. The tutorial of Hutton and Meijer (1998) shows how this approach is used to implement a monadic parser combinator library on top of the same type *Parser* as we used in Sec. 4. Both applicative functors (McBride and Paterson 2008) and arrows (Hughes 2000) have been proposed as alternative frameworks for the structure of parser combinator libraries.

We have designed our language of syntax descriptions to allow a similar programming style as with parser combinator libraries based on applicative functors. This decision allows to more easily adopt programs written for monadic or applicative parser combinator libraries into our framework. However, the definition of a $\langle\diamond\rangle$ combinator for curried function application can, for instance, be found in the tutorial by Fokker (1995).

Alternative approaches are based on arrows. Jansson and Jeuring (1999) implement both an arrow-based polytypic parser and an arrow-based polytypic printer in parallel with a proof that the parser is the left inverse of the printer. They implement a generic solution to serialization which is directly applicable to a wide range of types using polytypic programming. However, since they do not aim to construct human-readable output, they are not concerned with *pretty* printing, and since they cover multiple datatypes using polytypic programming, they do not provide an interface to construct more printers and parsers which are automatically inverse.

Alimarine et al. (2005) introduce bi-arrows as an embedded DSL for invertible programming based on arrows. Similar to our notion of partial isomorphisms, a bi-arrow can be inverted and run backwards. A number of combinators for bi-arrows are introduced, and a simple parser and pretty printer is implemented as a single program. While their bi-arrows resemble our partial isomorphisms, there is an important difference in the role these constructs play in the respective approaches. Alimarine et al. implement a parser and pretty printer directly as a bi-arrow, while we have defined the language of syntax descriptions as a functor on top of partial isomorphisms. Therefore, their parsers and printers resemble the parsers in EDSLs based on arrows, while our syntax descriptions resemble the parsers in EDSLS based on applicative functors.

Furthermore, their pretty printer does not handle advanced features like operator priorities and the automatic inserting of parentheses in the same general way as we do, but requires information about the location of parentheses to be contained in the abstract syntax tree. Generally, their work suffers from the methodically questionable decision to define a *BiArrow* type class as a subclass of the *Arrow* type class, even if some methods of *Arrow* could never be implemented for bi-arrows. These methods are defined to throw errors at runtime instead. On the other hand, Alimarine et al. present some transformers for bi-arrows. This approach could possibly be adapted to our notion of partial isomorphisms.

7.2 Functional unparsing

There has been some work on type-safe variants on the C *printf* and *scanf* functions. The standard variants of these functions take a string and a variable number of arguments. How these arguments are processed, and how many of them are accessed at all, is controlled by the formatting directives embedded into the string. This dependence of the type of the overall function on the value of first argument seemingly requires dependent types.

But Danvy (1998) has shown how to implement a type-safe variant of *printf* in ML by replacing the formatting string with an embedded DSL. The DSL is implemented using function composition and continuation passing style (CPS). The use of CPS allows Danvy to circumvent the fact that *Printer* is contravariant. However, in

Danvy's approach, it is not obvious how to define an abstraction like *Printer* as a parametric type constructor. More recently, Kiselyov (2008) implements type-safe *printf* and *scanf* so that the formatting specifications can be shared. Asai (2009) analyzes Danvy's solution, and shows that it depends on the use of delimited continuations to modify the type of the answer. The same can be done in direct style using the control operators *shift* and *reset*.

The work on type-safe *printf* and *scanf* shares some of the goals and part of the implementation method with the work presented in this article. In both approaches, an embedded DSL is used to allow a type-safe handling of formatting specifications for printing, parsing, and in Kiselyov's implementation, even for both printing and parsing at once. However, these approaches differ in the interface presented to the user, and in the support for recursive and user-defined types. *printf* and *scanf*'s continuation take a variable number of arguments depending on the formatting specification, but our *parse* and *print* functions take resp. return only one argument. Instead, we support more complicated arguments by using datatypes, and we support recursive types by building recursive syntax descriptions. It is not clear how user-defined datatypes and recursive syntax descriptions are supported in the *printf* and *scanf* approach. We allow to use a well-known Haskell idiom for parsing to be used for printing.

Hinze (2003) implements a type-safe *printf* in Haskell without using continuations, but instead composing functors to modify the type of the environment. The key insight of Hinze's implementation is that each of the elementary formatting directives specify a functor so that the type of *printf* is obtained by applying the composition of all the functors to *String*. Functor composition is implemented with multi-parameter typeclasses and functional dependencies. While we implement *Printer* resp. *Parser* as a single functor from an unusual category, Hinze implements his formatting directives as several functors and functor compositions.

7.3 Invertible functions

Mu et al. (2004) present an combinator calculus with a relational semantics, which can express only invertible functions. Programming in their "injective language for reversible computation" is based on a set of combinators quite similar to the algebra of partial isomorphisms in Sec. 5.1, but their language also contains a union operator to combine two invertible functions with disjoint domains and codomains. In our work, the $\diamondsuit$ operator plays a similar role on the level of syntax descriptions. Mu et al.'s language has a relational semantics, implemented by a stand-alone interpreter, while partial isomorphisms are implemented as an embedded DSL in Haskell.

Somewhat related to partial isomorphisms, functional lenses (Foster et al. 2008, 2005) can be described as functions which can be run backwards. However, functional lenses and partial isomorphisms use different notions of "running backwards". Running a lens forwards projects a part of a data structure into some alternative format. Running it backwards combines a possibly altered version of the alternative format with the original structure into an possibly altered version of the original structure. This is different from partial isomorphisms, where running backwards is not dependent on some original version of data. However, results about partial lenses may be applicable to partial isomorphisms. It is part of our future work to analyze their relationship.

Program inversion is concerned with automatically or manually inverting existing programs, while our approach for partial isomorphisms is based on the combination of primitive invertible building blocks into larger programs. Abramov and Glück (2002) give an overview over the field of program inversion.

Future work could try to combine our technique of running abstract machines backwards, and existing techniques for the transformation of semantic artifacts (Danvy 2008), into a technique for program inversion.

7.4 Categories other than *Hask*

In Sec. 3.1, we had to introduce the *IsoFunctor* class to abstract over functors from *Iso* to *Hask*, because Haskell's ordinary *Functor* does not support Functors involving categories different from *Hask*. Instead of introducing yet another category-specific functor class like *IsoFunctor*, one could use a more general functor class which allows to abstract over functors between arbitrary categories. Kmett (2008) supports such a "more categorical definition of Functor than endofunctors in the category Hask" in his $category-extras$ package.

$$\begin{aligned}&\textbf{class}\ (Category\ r, Category\ s)\\&\quad \Rightarrow CFunctor\ f\ r\ s \mid f\ r \to s, f\ s \to r\ \textbf{where}\\&\quad cmap :: r\ a\ b \to s\ (f\ a)\ (f\ b)\end{aligned}$$

Kmett declares a symbolic name for the category *Hask*, where the arrows are just Haskell functions.

$$\textbf{type}\ Hask = (\to)$$

The *CFunctor* type class is a strict generalization of Haskell's standard *Functor* class. While all instances of Haskell's standard *Functor* class can be declared instances of *CFunctor Hask Hask*, there are instances of *CFunctor* which cannot be expressed as *Functor*. For example, instances of *IsoFunctor* can be declared instances of *CFunctor Iso Hask*. Similarly, if the standard *Alternative* typeclass would have been parametric in the source and target categories of the applicative functor, we could have reused it directly, instead of duplicating its methods into our version of *Alternative*. Combinators and generic algorithms expressed in terms of the standard *Alternative* class would then be readily available for our functors from *Iso* to *Hask*.

This unnecessary need for code duplication suggests that the Haskell standard library could benefit from a redesign along the lines of Kmett's *CFunctor* class.

7.5 Other

Oury and Swierstra (2008) present the embedding of data definition languages as a use case of dependently typed programming and the use of universes in Agda. While their proposal has a somewhat monadic flair, Oury and Swierstra do not discuss functoriality of their type constructor. Furthermore, their prototype does not support user-defined datatypes, or recursive data types. In contrast, our implementation supports user-defined data types and (iso-) recursive types through the device of partial isomorphic functions. It would be interesting to see how the invariants of *Iso* values could be encoded in a dependently typed language.

Brabrand et al. (2008) define a stand-alone DSL for the specification of the connection between an XML syntax, and a non-XML syntax for the same language. Their implementation statically checks that a specified transfromation between two syntaxes is reversible by approximating a solution to the ambiguity problem of context-free grammars.

Hofer et al. (2008) describe a general methodology to embed DSLs in such a way that programs written in the DSL are polymorphic with respect to their interpretation. We have adopted their Scala-based design to Haskell using type classes.

8. Conclusion

We have described the language of syntactic descriptions, with which both parser and pretty-printer can be described as a single program. We have shown that sophisticated languages with keywords and operator priorities can be described in this style, resulting in useful parsers and pretty-printers. Finally, we have seen that

partial isomorphisms are a promising abstraction that goes beyond parsing and pretty-printing; functions such as fold/unfold can be described in a single specification.

Acknowledgments

We thank the anonymous reviewers for their insightful and encouraging comments.

References

Sergei Abramov and Robert Glück. Principles of inverse computation and the universal resolving algorithm. In *The essence of computation: complexity, analysis, transformation*, pages 269–295. Springer LNCS 2566, New York, 2002.

Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. There and back again: arrows for invertible programming. In *Proceedings of the workshop on Haskell (Haskell '05)*, pages 86–97, New York, 2005.

Kenichi Asai. On typing delimited continuations: three new solutions to the printf problem. *Higher-Order and Symbolic Computation*, 22(3): 275–291, September 2009.

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4-5):385–406, 2008.

Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.

Olivier Danvy. From reduction-based to reduction-free normalization. In *Advanced Functional Programming*, pages 66–164. Springer LNCS 5832, 2008.

J. Fokker. Functional parsers. In J.T. Jeuring and H.J.M. Meijer, editors, *Advanced Functional Programming, First International Spring School*, number 925 in LNCS, pages 1–23, 1995.

J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. In *Proceeding of the International Conference on Functional Programming (ICFP '08)*, pages 383–396, New York, 2008.

Nathan J. Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *Proceedings of the symposium on Principles of Programming Languages (POPL '05)*, pages 233–246, New York, 2005.

Ralf Hinze. Formatting: a class act. *Journal of Functional Programming*, 13(5):935–944, 2003.

Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of DSLs. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE '08)*, pages 137–148, New York, October 2008.

John Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, pages 53–96. Springer LNCS 925, 1995.

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, 1998.

Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In *European Symposium on Programming*, pages 273–287. Springer LNCS 1576, 1999.

Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

Oleg Kiselyov. Type-safe functional formatted IO. Available at http://okmij.org/ftp/typed-formatting/, 2008.

Edward A. Kmett. category extras: Various modules and constructs inspired by category theory. Available at `http://hackage.haskell.org/package/category-extras`, 2008.

Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In *Proceedings of the International Conference on Mathematics of Program Construction (MPC '04)*. Springer Verlag, 2004.

Nicolas Oury and Wouter Swierstra. The power of pi. In *Proceedings of the International Conference on Functional Programming (ICFP '08)*, pages 39–50, New York, 2008.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, 2002.

A. Generation of partial isomorphisms using Template Haskell

This appendix contains the implementation of the *constructorIso* and *defineIsomorphism* Template Haskell macros.

```
constructorIso c = do
  DataConI n _ d _              ← reify c
  TyConI ((DataD _ _ _ cs _)) ← reify d
  let Just con = find (λ(NormalC n' _) → n ≡ n') cs
  isoFromCon con

defineIsomorphisms d = do
  TyConI (DataD _ _ _ cs _) ← reify d
  let rename n
        = mkName (toLower c : cs) where c : cs = nameBase n
      defFromCon con@(NormalC n _)
        = funD (rename n)
            [clause [] (normalB (isoFromCon con)) []]
  mapM defFromCon cs

isoFromCon (NormalC c fs) = do
  let n     = length fs
  (ps, vs) ← genPE n
  v        ← newName "x"
  let f     = lamE [nested tupP ps]
                [| Just $(foldl appE (conE c) vs) |]
  let g     = lamE [varP v]
                (caseE (varE v)
                  [match (conP c ps)
                     (normalB [| Just $(nested tupE vs) |]) []
                  ,match (wildP)
                     (normalB [| Nothing |]) []])
  [| Iso $f $g |]

genPE n = do
  ids ← replicateM n (newName "x")
  return (map varP ids, map varE ids)

nested tup []     = tup []
nested tup [x]    = x
nested tup (x:xs) = tup [x, nested tup xs]
```