Polytypic Syntax Tree Operations

Arjen van Weelden, Sjaak Smetsers, and Rinus Plasmeijer

Institute for Computing and Information Sciences
Radboud University Nijmegen
The Netherlands
{A.vanWeelden, S.Smetsers, R.Plasmeijer}@cs.ru.nl

Abstract. Polytypic functional programming has the advantage that it can derive code for generic functions automatically. However, it is not clear whether it is useful for anything other than the textbook examples, and the generated polytypic code is usually too slow for real-life programs. As a real-life test, we derive a polytypic parser for the Haskell 98 syntax and look into other front-end compiler syntax tree operations. We present a types–as–grammar approach, which uses polytypic programming (in both Generic Haskell and Clean) to automatically derive the code for a parser based on the syntax tree type, without using external tools. Moreover, we show that using polytypic programming can even be useful for data–specific syntax tree operations in a (functional) compiler, such as scope checking and type inference.
Simple speed tests show that the performance of polytypic parsers can be abominable for real-life inputs. However, we show that much performance can be recovered by applying (extended) fusion optimization on the generated code. We actually have a derived parser whose speed is close to one generated by a specialized Haskell parser generator.

1 Introduction

The construction of complex software often starts by designing suitable data types to which functionality is added. Some functionality is data type specific, other functionality only depends on the structure of the data type. Polytypic programming is considered an important technique to specify such generic functionality. It enables the specification of functions on the structure of data types, and therefore, it is characterized as type dependent (type indexed) programming. The requested overall functionality is obtained by designing your data types such that they reflect the separation of specific and generic functionality. By overruling the polytypic instantiation mechanism for those parts of the data type that correspond to specific functionality, one obtains the desired overall behavior. In essence, a programmer only has to *program the exception* and a small polytypic scheme, since polytypic functions automatically work for the major part of the data types. Examples of such generic operations are equality, traversals, pretty-printing, and serialization.

The number of such generic operations in a specific program can be quite small, and hence the applicability of polytypic programming seems limited. Polytypic functions that are data specific only make sense if the involved data types

themselves are complex or very big. Otherwise, the definition of the polytypic version of an operation requires more effort than defining this operation directly. Moreover, the data-dependent functionality should be restricted to only a small portion of the data type, while the rest can be treated generically.

This paper investigates the suitability of polytypic programming as a general programming tool, by applying it to (a part of) compiler construction. Compilers involve both rich data structures and many, more or less complex, operations on those data structures. We focus on the front-end of compilers: parsing, post-parsing, and type inference operations on the syntax tree. There exist many special tools, e.g., parser generators and compiler compilers, that can be used for constructing such a front-end. We show that polytypic programming techniques can also be used to elegantly specify parsers. This has the advantage that the polytypic functional compiler can generate most of the code. Another advantage it that one can specify everything in the functional language itself, without synchronization issues, e.g., between the syntax tree type and the grammar definition, with external tools.

We have implemented polytypic parsers in both Generic Haskell [1] (a preprocessor for Haskell [2]) and Clean [3]. We use (Generic) Haskell to present our implementation in this paper, the Clean code is very similar. The polytypic parser we use in this paper differs from those commonly described in papers on polytypic programming [4, 5]. Our parser is based on the *types–as–grammars* approach: the context-free syntax of the language to parse is specified via appropriate data type definitions. The types–as-grammar approach was previously used to construct a new version of the Esther shell originally described in Weelden and Plasmeijer [6]. The shell uses polytypic programming to specify the parser and post-parsing operations on expressions the size of a single command-line. This paper tackles larger inputs and grammars, including the Haskell syntax.

Apart from its expressiveness, a programming technique is not very useful if the performance of the generated code is inadequate. The basic code generation schema that is used in the current implementations of polytypic systems produces inefficient code. We asses the efficiency of both the Generic Haskell and the Clean implementations and compare them with the code generated by an optimization tool by Alimarine and Smetsers [7]. This tool takes a polytypic Clean program as input and produces a Haskell/Clean-like output without the polytypic overhead.

To summarize, the main contributions of this paper are:

- We show that polytypic programming, introduced in Sect. 2, is not only suited for defining more or less inherently generic operations, but also for specifying data specific functionality.
- We describe a technique that allows us to derive a parser for context-free languages automatically from the definition of a syntax tree in Sect. 3. The technique is based on the idea to interpret types as grammar specifications.

– We show that the same technique applies to several related syntax tree operations in Sect. 4. As operations become more data specific, we gain less from using polytypic programming. However, we show that it is not totally unsuitable for non-generic algorithms.
– As most polytypic programmers know, polytypic programs (including our parsers) have serious performance problems. Fortunately, we show in Sect. 5 that an appropriate optimization tool recovers a lot of efficiency, and that our parsers can approach the speed of parsers generated by external tools.

Related work is discussed in Sect. 6, and we conclude in Sect. 7.

2 Polytypic Programming

Specifying polytypic functions is a lot like defining a type class and its instances. The main difference is that a polytypic compiler can derive most of the instances automatically, given a minimal fixed set of instances for three or four (generic) types. The programmer can always overrule the derived instance for a certain type by specifying the instance manually. This powerful derivation scheme even extends to kinds (the types of types), which we will neither use nor explain in this paper.

The fact that polytypic functions can be derived for most types is based on the observation that any (user defined) algebraic data type can be expressed in terms of eithers, pairs, and units. This generic representation, developed by Hinze [12], is used by Generic Haskell and is encoded there by the following Haskell types:

```
data Sum a b = Inl a | Inr b   — either/choice between (In)left and (In)right
data Prod a b = a :*: b        — pair/product of two types, left associative
data Unit = Unit               — the unit type
```

A data type and its generic representation are isomorphic. The corresponding isomorphism can be specified as a pair of conversion functions. E.g., for lists the generic representation and automatically generated associated conversion functions are as follows.

```
type GenericList a = Sum (Prod a [a]) Unit

fromList :: [a] → GenericList a        toList :: GenericList a → [a]
fromList (x:xs) = Inl (x :*: xs)       toList (Inl (x :*: xs)) = x:xs
fromList []     = Inr Unit             toList (Inr Unit)       = []
```

Note that the generic representation type `GenericList` is not recursive and still contains the original list type. A polytypic function instance for the list type can be constructed by the polytypic system using the generic representation. The derived instance for the list type uses the given instances for `Sum`, `Prod`, `Unit`, and once again the currently deriving instance for lists. This provides the recursive call, which one would expect for a recursive type such as lists.

To define a polytypic function, the programmer has to specify its function type, similar to a type class, and only the instances for the generic types (`Prod`, `Sum`, and `Unit`) and non-algebraic types (like `Int` and `Double`). The polytypic instances for other types that are actually used inside a program are automatically derived. Polytypic functions are, therefore, most useful if a large collection of (large) data types is involved, or if the types change a lot during development.

To illustrate polytypic programming we use the following syntax tree excerpt:

```
data Expr = Apply Expr Expr              | Lambda Pattern Expr
          | Case Expr [(Pattern, Expr)] | Variable String
          | If Expr Expr Expr           | ···

data Pattern = Var String | Constructor String [Pattern] | ···

data ···
```

We define a Generic Haskell function `print` of type $\texttt{a} \to \texttt{String}$ that is polytypic in the type variable `a`, similar to Haskell's `show` of type $\texttt{Show a} \Rightarrow \texttt{a} \to \texttt{String}$ that is overloaded in `a`. Instead of instances for the `Show` class, we define type instances for `print` using the special parentheses $\{\!|\ |\!\}$.

```
print {| a |} :: a → String

print {| Int |}       i          = show i            — basic type instance

print {| Unit |}      Unit       = ""                — unit instance

print {| Sum a b |}   (Inl l)    = print {| a |} l   — left either instance
print {| Sum a b |}   (Inr r)    = print {| b |} r   — right either instance

print {| Prod a b |}  (l :*: r)                      — pair instance
        = print {| a |} l ++ " " ++ print {| b |} r

print {| Con d a |}   (Con x)                        — instance for constructors
        = "(" ++ conName d ++ " " print {| a |} x ++ ")"
```

To print the parameterized type `Sum`, and also `Prod`, `print` requires printing functions for the parameter types `a` and `b`. These are automatically passed under the hood by Generic Haskell, similar to dictionaries in the case of overloading. `print {| Sum a b |}` can refer to these hidden dictionary functions using `print {| a |}` and `print {| b |}`. Furthermore, the type `Con`, used in this example, was added to the set of generic types in Generic Haskell as well as in Clean. Run-time access to some information about the original data constructors is especially convenient when writing trace functions, such as `print`, for debugging purposes.

```
data Con a = Con a;          data ConDescr = { conName :: String, ··· }
```

When used in Generic Haskell, `Con` *appears* to get an additional argument `d`. This is not a type argument but a denotation that allows the programmer to access information about the constructor, which is of type `ConDescr`. In the example `print {| Con d a |}` applies `conName` to `d` to retrieve the name of the constructor.

Observe that this polytypic print function does not depend on the structure of the syntax tree type. If this type definition changes during development, the underlying system will automatically generate a proper version of `print`. This implementation of `print` is quite minimal, with superfluous parentheses and spaces. It is easy to adjust the definition to handle these situations correctly, see for example Jansson and Jeuring [4].

It is not difficult to specify the polytypic inverse of the print function. Using a monadic parser library, with some utility functions such as `symbol(s)` and `parseInt` that take care for low-level token recognition, one could specify a polytypic parse function (similar to Haskell's `read`) as follows:

```
type Parser a = ··· — some monadic parser type

parse {| a |} :: Parser a

parse {| Unit |}     = return Unit

parse {| Sum a b |}  = mplus (parse {| a |} >>= return . Inl)
                             (parse {| b |} >>= return . Inr)

parse {| Prod a b |} = do    l ← parse {| a |}
                             r ← parse {| b |}
                             return (l :*: r)

parse {| Con d a |}  = do    symbol '('
                             symbols (conName d)
                             symbol ' '
                             x ← parse {| a |}
                             symbol ')'
                             return (Con x)
parse {| Int |}      = parseInt
```

Such a simple parser follows the print definition very closely and is easy to understand. `parse` is obviously `print`'s inverse, and it can only parse input generated by the print function, including redundant spaces and parentheses.

3 Polytypic Parsing of Programming Languages

This section introduces the types–as–grammar approach to polytypically derive a parser. This parser builds on a small layer of monadic parser combinators, to abstract from the lower level token recognition machinery. We use very naive parser combinators (shown below) because they are easy to use and explain.

To abstract from the parsing issues at the lexical level, we assume a separated scanner/lexer and that the parser will work on a list of tokens. Later in Sect. 5, we will test the efficiency of the polytypic parser using also a set of continuation parser combinators that improve the error messages. The naive monadic parser, using the `Maybe` monad, is implemented as follows.

```
newtype Parser a = Parser { parser :: [Token] → Maybe (a, [Token]) }

data Token = IdentToken String | LambdaToken | ArrowToken
           | IfToken | ThenToken | ElseToken | ···          — all tokens

token :: Token → Parser Token
token tok = Parser (λts → case ts of
                            (t:ts') | t == tok → Just (t, ts')
                             _                 → Nothing

instance Monad Parser where
    return x = Parser (λts → Just (x, ts))                  — success parser
    l >>= r  = Parser (λts → case parser l ts of            — sequence parser
                               Just (x, ts') → parser (r x) ts'
                               Nothing       → Nothing      )

instance MonadPlus Parser where
    mzero     = Parser (λts → Nothing)                      — fail parser
    mplus l r = Parser (λts → case parser l ts of           — choice parser
                                Just (x, ts') → Just (x, ts')
                                Nothing       → parser r ts  )
```

The `mplus` instance above defines a deterministic (exclusive) choice parser: if the left argument of `mplus` parses successfully, the right argument is never tried. This is done out of speed considerations and, if the parsers are written in the right way, it does not matter for deterministic grammars. Algebraic data constructors have unique names, which makes the grammar deterministic. This is also reflected in the `Parser` type, i.e., the parser returns a `Maybe` result, which shows that it returns at most one result.

To parse real programming languages we should not parse the constructor names that occur in the syntax tree type. Instead, we should parse all kinds of tokens such as **if**, λ, and →. This requires writing most of the instances for the polytypic function `parse` by hand. Another option is adding these tokens to the abstract syntax tree, which becomes a non-abstract, or rich, syntax tree. Since we instruct the polytypic parser using types, we cannot reuse the (constructors of the) `Token` data type. Instead, we specify each token as a separate data type. This gives us the ability to parse our own tokens, without the constructors getting in the way. We can now define, for example, a nicer parser for lists that uses the `[]` and `_:_` notation.

```
data List a = Cons a ColonToken (List a) | Nil EmptyListToken
```

```
data ColonToken      = ColonToken
data EmptyListToken = EmptyListToken

parse {| ColonToken |}      = symbol ':'    >>  return ColonToken
parse {| EmptyListToken |} = symbols "[]" >>  return EmptyListToken

parse {| Con d a |}         = parser {| a |}   >>= return . Con

intListParser = parse {| List Int |}     — automatically derived by the system
```

We partly reuse the `parse` definition from Sect. 2. We do not want to parse the constructor names, therefore, we replace the `Parse {| Con d a |}` alternative from Sect. 2 with the one shown above. Not parsing constructor names means that the order of alternatives is important. Since `parse {| Sum a b |}` uses the exclusive `mplus`, it gives priority to the `Inl`(eft) alternative over the `Inr`(ight) alternative. Therefore, the textual order of the constructors of an algebraic data type determines the order of parsing, which is similar to function definitions with multiple alternatives in Haskell and Clean.

One can parse any context-free syntax by specifying the grammar using algebraic data types. The grammar below is an excerpt of a small functional programming language. It uses the convention that N*type* represents the non-terminal *type* and T*type* represents a terminal symbol *type*.

```
data Nexpression = Apply Nexpression Nexpression
                 | Lambda Tlambda Nvariable Tarrow Nexpression
                 | If Tif Nexpression Tthen Nexpression
                                      Telse Nexpression
                 | Variable Nvariable
                 | Value Nvalue

data Nvariable = Variable String

data Nvalue = Integer Int | Boolean Bool

data Tlambda = Tlambda;   data Tarrow = Tarrow
data Tif     = Tif;       data Tthen  = Tthen;    data Telse = Telse

parse {| Con d a |}  = parse {| a |}       >>= return . Con
parse {| String |}   = identifierToken >>= λ(IdentToken s) → return s

parse {| Tlambda |}  = token LambdaToken >> return Tlambda
parse {| Tarrow |}   = token ArrowToken  >> return Tarrow
parse {| Tif |}      = token IfToken     >> return Tif
parse {| Tthen |}    = token ThenToken   >> return Tthen
parse {| Telse |}    = token ElseToken   >> return Telse
```

If we remove all constructors from the type definitions above, we end up with something that looks very similar to the following grammar description in BNF notation:

```
<expression> ::= <expression> <expression>
               | "λ" <variable> "→" <expression>
               | "if" <expression> "then" <expression>
                                  "else" <expression>
               | <variable>
               | <value>

<variable>   ::= String

<value>      ::= Int | Bool
```

It is also easy to support extended BNF (EBNF) notation by introducing some auxiliary data types: `Plus` to mimic $(\cdots)^+$, `Option` to mimic $[\cdots]$, and `Star` to mimic $(\cdots)^\star$. The parsers for all of them can be derived automatically.

```
data Plus a       = Plus a (Plus a) | One a
type Star a       = Option (Plus a)
type Option a     = Maybe a

data Nexpression = ···
                 | Lambda Tlambda (Plus Nvariable) Tarrow Nexpression
                 | ···
```

The use of parameterized data types, such as `Plus`, can make the definition of the syntax tree type very concise. It is similar to two-level or van Wijngaarden grammars [8]. We can now specify a lambda expression with multiple arguments using `Plus` as shown above. Clearly, this corresponds to the following EBNF grammar:

```
<expression> ::= ···
               | "λ" <variable>+ "→" <expression>
               | ···
```

An issue with this *types–as–grammar* approach is left-recursive type definitions. Most parser combinator libraries do not support left-recursive parser definitions and run out of heap or stack space. Recently, Baars and Swierstra developed parser combinators [9] that detect and remove left-recursion automatically . Our current solution is manually removing the (few occurrences of) left-recursion by splitting the left-recursive type, as shown below. Only `Nexpression` is (mutually) left-recursive because it has no argument of type `T`*token* before the `Nexpression` arguments. We write a small parser for the left-recursive part, making sure that most of the parser is still derived automatically.

```
data Nexpression = Apply Nexpression Nexpression
                 | Term Nterm                  — separate non-recursive part
```

```
data Nterm        = Lambda Tlambda (Plus Nvariable) Tarrow Nexpression
                  | ···

parse {| Nexpression |} = parse {| Plus Nterm |} >>= return . app
 where
    app (One t)      = Term t
    app (Plus t ts) = app' (Term t) ts
    app' acc (One t)     = Apply acc t
    app' acc (Plus t ts) = app' (Apply acc t) ts
```

We extended this example to a basic functional language grammar, to test our generated parser. Moreover, as a larger test, we converted Haskell's grammar to types and derived a parser for it. The results of those tests appear in Sect. 5.

4 Other Polytypic Syntax Tree Operations

Polytypic parsing and several other polytypic syntax tree operations are used in the current version of the Esther shell [6], which is written using Clean's generics. The Esther shell offers a basic lazy functional language as shell syntax. Its grammar is specified as a type, using the approach of Sect. 3. This section uses excerpts from the Esther shell to give an impression about how data specific syntax tree operations, written using polytypic programming techniques, improve conciseness, modularity, and allow easy changes to the syntax by adding and rearranging types.

4.1 Restructuring Infix Expressions

A common syntax tree operation is re-parsing expressions that contain user defined infix operators. Because they are user defined, they cannot be correctly parsed during the first parse. The usual solution is to restructure the syntax tree after parsing, once the precedence and associativity information is available.

```
data FixityInfo = ··· — precedence and associativity information

fixInfix {| a | m |} :: (Functor m, Monad m) ⇒ a → FixityInfo → m a

fixInfix {| Int |}       i       ops = return i
fixInfix {| Unit |}      Unit    ops = return Unit
fixInfix {| Sum a b |}   (Inl l)  ops = do   l' ← fixInfix {| a |} l ops
                                            return (Inl l')
fixInfix {| Sum a b |}   (Inr r)  ops = do   r' ← fixInfix {| b |} r ops
                                            return (Inr r')
fixInfix {| Prod a b |}  (l :*: r) ops = do  l' ← fixInfix {| a |} l ops
                                            r' ← fixInfix {| b |} r ops
                                            return (l' :*: r')
```

```
fixInfix {| Nexpression |} (Term t) ops = do
                                           t' ← fixInfix {| Nterm |} t ops
                                           return (Nterm t)
fixInfix {| Nexpression |} (Apply e1 e2) ops = ···— rebuild expression tree
```

We overloaded `fixInfix` with the `Monad` class because this operation can fail due to conflicting priorities. Generic Haskell requires mentioning this type variable `m` at the left side of the function type definition. The polytypic restructuring `fixInfix` function can be derived for all types except `Nexpression`, which is where we intervene to restructure the syntax tree. Note that manually removing the left-recursion and splitting the `Nexpression` type allows us to override the polytypic function derivation at exactly the right spot. We lack the space to show exactly how to restructure the expression tree. This can be found in the current version of the Esther shell [6].

The traversal code in the instances for the generic representation types is a common occurring pattern. This shows that we can elegantly and concisely specify a syntax tree operation that operates on a very specific part of the tree. There is no need to specify traversal code for any other type in the syntax tree, these are all automatically derived.

4.2 Adding Local Variable Scopes

Another common operation is checking variable declarations in the context of local scope. Scope can easily be added into the syntax tree using polytypic programming. We simply define the `Scope` data type below and inject it into the syntax tree where appropriate.

```
data Scope a  = Scope a

data Nterm    = LambdaWithScope (Scope Nlambda)
              | ···
data Nlambda  = Lambda Tlambda (Plus Npattern) Tarrow Nexpression
data Ncase    = Case Tcase Nexpression Tof
                                    (Plus (Scope Nalt, Tsemicolon))
data Nalt     = Alternative (Plus Npattern) Tarrow Nexpression
data Npattern = ···
              | VariablePattern Nvariable
```

We overrule the derived polytypic code for `chkVars` at the following positions in the syntax tree types: `Nvariable` is an applied occurrence, except for occurrences after a `VariablePattern` constructor (part of the `Npattern` type), where it is a defining occurrence. Furthermore, we override the polytypic instance for `Scope`, which ends a variable scope after lambda expressions and case alternatives.

```
chkVars {| a | m |} :: (Functor m, Monad m) ⇒ a → [String] → m [String]

chkVars {| Unit |}      _      vs = return vs
chkVars {| Int |}       _      vs = return vs
```

```
chkVars {| Prod a b |} (l :*: r) vs = chkVars {| a |} l vs >>= chkVars {| b |} r
chkVars {| Sum a b |}  (Inl l)   vs = chkVars {| a |} l vs
chkVars {| Sum a b |}  (Inr r)   vs = chkVars {| b |} r vs

chkVars {| Nvariable |} (Variable v) vs
        | v `elem` vs = return vs
        | otherwise   = fail ("unbound variable: " ++ v)

chkVars {| case VariablePattern |} (VariablePattern (Variable v)) vs
        = return (v:vs)          -- polytypic instance for a single constructor

chkVars {| Scope a |} (Scope x) vs = chkVars {| a |} x vs >> return vs
```

We make use of a Generic Haskell feature in the `chkVars` example above, which is not found in Clean: overriding the generic scheme at the constructor level. Instead of writing code for all constructors of the `Npattern` type, we only specify the semantics for the `VariablePattern` (hence the use of the **case** keyword) and let Generic Haskell derive the code for the other alternatives of the type.

4.3 Type Inference

As the compilation process proceeds, syntax tree operations tend to be less generic and more data specific. Program transformations and code generation, but also type checking, usually require writing polymorphic instances for almost all types, since each type must be treated differently. At first sight, it seems as if polytypic programming is no longer useful to implement such operations. In this section, we will show that even for more data specific functions a polytypic definition improves modularity because it splits the specification per type, even if there is little profit from the automatic derivation mechanism. As an example, we specify a type inference algorithm in a polytypic way. Type inference is much more data specific than any other example in this paper, nevertheless, it illustrates the way to polytypically specify syntax operations that occur later in the compilation process.

The algorithm is based on the idea of strictly separating type inference into the generation of constraints (in the form of type equations), and solving these constraints by standard unification. We restrict ourselves to the generation part, which is usually done by traversing the syntax tree and collecting constraints corresponding to each syntactical construct. Such an algorithm not only takes the syntax tree as input but also an environment containing type declarations for functions, constructors, and variables. Moreover, during the generation process we sometimes need fresh type variables, e.g., to instantiate a function's type scheme or to create local type variables used to express dependencies between derivations. Therefore, we supply the generation function with a heap data structure and we use an accumulator to collect type equations. This leads to the following polytypic function type and auxiliary type definitions.

```
data Type = TVar String        | TTVar VHeap    | TBasic TBasic
          | TApp String [Type] | TArr Type Type | TAll [String] Type

data TBasic = TBool | TInt
data Equ    = Equ Type Type

type TypeState a = State (VHeap, [Equ]) a    -- a state monad

gtype {| t |} :: t → Envs → TypeState Type
```

The `VHeap` is used to allocate fresh type variables. Mostly it suffices to generate unique integers to distinguish different type variables. These fresh variables are represented by the `TTVar`-alternative in the definition of `Type`. The other alternatives are used to represent type variables, basic types, type constructor applications, arrow types, and type schemes, respectively.

The type equations are represented as a list of `Equ` elements. Together with the `VHeap`, they form the state of the polytypic function. For convenience, the implementation of the polytypic `gtype` function is based on the standard `State` monad. For creating fresh variables, and for extending the list of type equations we introduce the following functions.

```
freshVar :: TypeState VHeap
freshVar = State {runState = λ(vh, eqs) → (vh, (vh+1, eqs))}

newEqu :: Type → Type → TypeState ()
newEqu dt ot = State
                 {runState = λ(vh, eqs) → ((), (vh, Equ dt ot:eqs))}
```

The polytypic instance declarations are straightforward. We chose to interpret a `Prod` of two terms as an application of the first to the second. The advantage is that we can derive the instance for the type `Nexpression` automatically.

```
gtype {| Sum a b |}  (Inl l)  env = gtype {| a |} l env
gtype {| Sum a b |}  (Inr r)  env = gtype {| b |} r env
gtype {| Prod a b |} (x :*: y) env = do   tx ← gtype {| a |} x env
                                          ty ← gtype {| b |} y env
                                          fv ← freshVar
                                          newEqu (TArr ty (TTVar fv)) tx
                                          return (TTVar fv)
```

Clearly, there are not many other types for which we use the polytypic version; most of the instances have to be given explicitly. E.g., for `TfunctionId` we can use the following definition:

```
gtype {| TfunctionId |} (FunctionId name) env
       = freshType name (fun_env env)
```

The overall environment has three separate environments: for functions, for constructors, and for type variables.

```
type Env  = String → Type
data Envs = Envs { fun_env :: Env, cons_env :: Env, var_env :: Env }
```

The function `freshType` takes care of the instantiation of the environment type. It can be defined easily, using the `freshVar` function, for type variables introduced by a `TAll` type. Another example is the alternative for `Nif`. Again, its definition is straightforward.

```
gtype {| Nif |} (If Tif c Tthen t Telse e) env = do
        tc ← gtype {| Nterm |} c env
        newEqu tc (TBasic TBool)
        tt ← gtype {| Nterm |} t env
        te ← gtype {| Nterm |} e env
        newEqu tt te
        return tt
```

Although we have to specify many instances explicitly, it is not inconvenient to use a polytypic specification: it splits the implementation into compact polytypic instances, which are easy to write while the resulting structure of the algorithm remains clear.

Concluding this section, we want to remark that polytypic programming allowed easy changes to the syntax by adding and rearranging types. Usually, this was done by *adding* types and instances to polytypic functions, instead of *rewriting* existing instances.

5 Performance of Polytypic Parsers

In this section we investigate the efficiency of the generated parsers for two different grammars/languages. Our elegant types–as-grammar technique is of little practical use if the resulting programs perform poorly because of the automatically derived code by the polytypic system. Who cares about the advantage of not having to use an external tool, when the polytypic parsers performs an order of magnitude worse than parser generator based parsers.

5.1 A Basic Functional Language Parser

The first example is the derived parser for the basic functional language from Sect. 3. Since we are not interested in lexical analysis, we have tokenized the test input for the parser manually resulting in a list of 663 tokens representing 45 small functions in this language. The programs under test copy the input list of tokens 100 times and parse the resulting list 100 times. The results are shown in Table 1. For Haskell we used Generic Haskell (GH) 1.42, which requires the GHC 6.2.2 compiler. For Clean we used the Clean 2.1.1 distribution.

All programs were run with a heap size of 256MB. It's remarkable to see that the Haskell version used only a quarter of the heap allocated by the Clean version. At first glance, it might not be clear that the generated executables are very slow and consume huge amounts of memory. Both Generic Haskell

	Execution time (s)	Garbage collection (s)	Total time (s)	Total heap allocation (MB)
GH+GHC	27.2	1.4	28.6	3,500
Clean	45.0	6.7	51.8	11,600

Table 1. Performance figures for the derived basic functional language parser, using `Maybe` parsers.

and Clean have some built-in specific optimization techniques to improve the performance of the derived functions. Moreover, these derived functions also benefit from standard optimizations, such as dictionary elimination, higher-order removal, etc. However, it appears that this is insufficient to obtain any acceptable performance.

5.2 Improving the Automatically Derived Code

In [7] Alimarine and Smetsers present an optimization technique, called *fusion*, of which they claim that it removes all the overhead introduced by the compilation scheme for polytypic functions (developed by Hinze [12]) that is used both in Generic Haskell and in Clean. Like *deforestation*, fusion aims at removing intermediate data used for passing information between function calls. This is done by combining nested pairs of *consumer* and *producer* calls into a single function application, making the construction of intermediate data structures from the producer to the consumer superfluous.

Fusion is not implemented in the Clean compiler, but incorporated in a separate source–to–source translator. The input language for this translator is a basic functional language extended with syntactical constructs for specifying polytypic functions. The translator first converts polytypic definitions into ordinary function definitions and optimizes these generated functions, by eliminating data conversions that are necessary to convert each object from and to its generic representation. The optimized output is both Clean and Haskell syntax compatible, so it was easy to include performance figures using both compilers as a back–end. These figures are shown in Table 2.

	Execution time (s)	Garbage collection (s)	Total time (s)	Total heap allocation (MB)
Fusion+GHC	4.3	0.03	4.5	340
Fusion+Clean	6.3	0.4	6.7	1,500

Table 2. Execution times for the optimized basic functional language parser, using `Maybe` parsers.

The programs ran under the same circumstances as those shown in Table 1. Each test yields a syntax tree consisting of approximately 300,000 constructors per iteration. In the optimized Haskell version this leads to an allocation of 12 bytes per node. Representing a similar syntax tree in an imperative language would require approximately the same number of bytes per node.

5.3 Using Continuation Based Parser Combinators

A nice aspect of our approach, is that the polytypic specification of the parser in Sect. 3 and the underlying parser combinator library are independent: we are free to choose different combinators, e.g., combinators that produce better error messages, without having to adjust the polytypic definitions. To illustrate this, we replaced the simple `Maybe`-combinators, by a set of continuation based parser combinators, which collect erroneous parsings. These are similar to the combinators by, e.g., Koopman [10] or Leijen and Meijer [11]. Although the error reporting technique itself is simple, it appears that the results are already quite accurate. Of course, one can fine-tune these underlying combinators or even switch to an existing set of advanced combinators, e.g., Parsec [11], without having to change the polytypic parser definition itself.

	Execution time (s)	Garbage collection (s)	Total time (s)
GH+GHC	137.9	10.2	148.2
Clean	77.3	20.0	97.3
Fusion+GHC	18.6	0.41	19.0
Fusion+Clean	55.5	8.74	64.2

Table 3. Execution times for the derived and optimized basic functional language parser, using continuation based parsers.

We have tested the unoptimized as well as the optimized version of the continuation based parser, see Table 3. This time, the figures are more difficult to explain, in particular if you compare them with the execution times from the previous tables. In the literature, continuation passing parsers are often presented as an efficient alternative for the naive combinators. However, our measurements do not confirm this. The polytypic, as well as the optimized versions, are much slower than the corresponding parser from the first test set, up to a factor of ten. One might believe that the additional error information causes this overhead. However, the loss in efficiency is almost the same when this information is not included. Apparently, the gain that is obtained by avoiding explicit constructors and pattern matching is completely undone by the use of continuations and, therefore, higher-order applications.

5.4 A Haskell 98 Parser

As a second test we have implemented a (nearly) complete Haskell parser, simply by deriving polytypic parser instances for the Haskell syntax specified as a collection of algebraic data types. These data types were obtained by a direct conversion of the Haskell syntax specification as given in section 9.5 of the Haskell 98 Report [2]. Again, we have compared the results for Generic Haskell and Clean for both the `Maybe` and the continuation passing combinators. We also optimized the generic code and compared the performance of all different versions. The results are shown in Table 4. The parsers were run on an example input consisting of approximately 500 again manually tokenized lines of Haskell code, 2637 tokens

An optimization that replaces update-frames with indirections was added to the Clean run-time system, reducing both heap and stack usage enough too complete the tests on a 1.5Ghz 512MB Windows PC.

	GH+GHC (s)	Clean (s)	Fusion+GHC (s)	Fusion+Clean (s)
`Maybe`	20.6	17.6	0.03	2.30
CPS	182	15.2	1.12	5.40

Table 4. Performance figures for the derived and optimized Haskell 98 parser, using both `Maybe` and continuation bases parsers.

These execution times are quite revealing. We can conclude that Generic Haskell as well as Clean generate extremely inefficient polytypic code. It is doubtful whether these polytypic language extensions are really useful for building serious applications. However, the optimization tool changes this completely, at least for Haskell. The performance gain for the `Maybe`-parsers is even a factor of 700. This test indicates once more that the continuation passing parsers are less efficient. It is strange to see that for Haskell the difference is much bigger than for Clean: a factor of 35 and 2, respectively. We do not have an explanation for the factor of 75 between GHC and Clean for the optimized `Maybe`-parsers.

We have also compared the efficiency of the optimized parsers with a Haskell parser generated with the *Happy* tool [13]. This parser is included in the libraries of the GHC Haskell compiler we used. The result is surprising: its execution time is exactly the same as our Fusion+GHC `Maybe`-parser! To get more significant results we ran both with 100 times the input ($50,000$ lines of Haskell code, using a 4MB heap). Our parser is five percent faster, but does not have a lexer or decent error messages. Nonetheless, we believe that this shows that fusion is really needed and that fusion works for polytypic parsers.

6 Related Work

Parsers are standard examples for polytypic programming (see Jansson and Jeuring [4], Hinze [14]). However, the common definition gives a parser that can only recognize expressions that can be defined in the corresponding programming language itself. This is very natural because the type definitions in a programming language can be regarded as a kind of grammar defining legal expressions in the corresponding programming language. We have shown that this also works for any context-free grammar.

It has also been shown how a parser for another language can be constructed from a grammar description. Atanassow, Clarke, and Jeuring [15] construct parsers for XML from the corresponding DTD description. To the best of our knowledge this paper is the first that describes the use of algebraic data type definitions as a grammar for deriving polytypic parsers for arbitrary languages.

There exist other (lazy, functional) parser generator tools and combinator libraries [13, 16, 9–11], which may generate better parsers than our approach, due to grammar analysis or handwritten optimizations. What makes our approach appealing, is that the tool used to generate the parser is part of the language. This removes the need to keep your syntax tree data structures synchronized with an external tool: one can do it within the polytypic functional language, and efficiently too, using extended fusion.

7 Conclusions

With this paper we have illustrated that polytypic programming techniques, as offered by the Generic Haskell preprocessor and the Clean compiler, can effectively be used for compiler construction. Additionally, we hope to have illustrated that the technique is interesting for programming in general.

Polytypic functions are type driven, it is therefore important to know what can be expressed in a type. In this paper we have shown that context-free grammars can be encoded in a straightforward way using algebraic data types. We have defined a polytypic parser using a types–as–grammar approach. Using such a polytypic definition, a parser for an arbitrary context-free language can be derived automatically. The polytypic function is defined in terms of parser combinators, and one can easily switch from one library to another.

Moreover, we have shown how other convenient polytypic post-parsing operations on the resulting rich syntax tree can be defined, even if not all syntax tree operations gain much from the polytypic programming style. It gives you the flexibility of moving data types within larger type structures, mostly by adding polytypic instances without having to change (much of) the existing code.

Finally, we have shown that optimizations that remove the polytypic overhead are really necessary to make polytypic programs usable. Currently, polytypic programming, in either Generic Haskell or Clean, may be suitable for toy examples and rapid prototyping but the derived code is definitely not efficient enough for larger programs. Using the extended fusion optimization technique,

the parser's efficiency came close to a parser generated by Happy. We believe that fusion makes polytypic programming for real-world applications possible.

References

1. Löh, A., Clarke, D., Jeuring, J.: Dependency-style Generic Haskell. In: Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming ICFP'03, ACM Press (2003) 141–152
2. Peyton Jones, S.: Haskell 98 language and libraries: the Revised Report. Cambridge University Press (2003) http://www.haskell.org/definition/.
3. Alimarine, A., Plasmeijer, R.: A Generic Programming Extension for Clean. In Arts, T., Mohnen, M., eds.: The 13th International workshop on the Implementation of Functional Languages, IFL'01, Selected Papers. Volume 2312 of LNCS., Älvsjö, Sweden, Springer (2002) 168–186
4. Jansson, P., Jeuring, J.: Polytypic compact printing and parsing. In Swierstra, S.D., ed.: Proceedings 8th European Symposium on Programming, ESOP'99, Amsterdam, The Netherlands, 22–28 March 1999. Volume 1576., Berlin, Springer-Verlag (1999) 273–287
5. Jansson, P., Jeuring, J.: Polytypic data conversion programs. Science of Computer Programming **43**(1) (2002) 35–75
6. van Weelden, A., Plasmeijer, R.: A functional shell that dynamically combines compiled code. In Trinder, P., Michaelson, G., eds.: Selected Papers Proceedings of the 15th International Workshop on Implementation of Functional Languages, IFL'03, Heriot Watt University, Edinburgh (2003)
7. Alimarine, A., Smetsers, S.: Improved fusion for optimizing generics. In Hermenegildo, M., Cabeza, D., eds.: Proceedings of Seventh International Symposium on Practical Aspects of Declarative Languages. Number 3350 in LNCS, Long Beach, CA, USA, Springer (2005) 203 – 218
8. van Wijngaarden, A.: Orthogonal design and description of a formal language. Technical Report MR 76, Mathematisch Centrum, Amsterdam (1965)
9. Baars, A.I., Swierstra, S.D.: Type-safe, self inspecting code. In: Proceedings of the ACM SIGPLAN workshop on Haskell, ACM Press (2004) 69–79
10. Koopman, P., Plasmeijer, R.: Layered Combinator Parsers with a Unique State. In Arts, T., Mohnen, M., eds.: Proceedings of the 13th International workshop on the Implementation of Functional Languages, IFL'01, Ericsson Computer Science Laboratory (2001) 157–172
11. Leijen, D., Meijer, E.: Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Departement of Computer Science, Universiteit Utrecht (2001) http://www.cs.uu.nl/∼daan/parsec.html.
12. Hinze, R.: Generic Programs and proofs (2000) Habilitationsschrift, Universität Bonn.
13. Gill, A., Marlow, S.: Happy: The parser generator for Haskell (2001) http://www.haskell.org/happy/.
14. Hinze, R., Peyton Jones, S.: Derivable Type Classes. In Hutton, G., ed.: 2000 ACM SIGPLAN Haskell Workshop. Volume 41(1) of ENTCS., Montreal, Canada, Elsevier Science (2001)
15. Atanassow, F., Clarke, D., Jeuring, J.: Scripting XML with Generic Haskell. Technical report uu-cs-2003, University of Utrecht (2003)
16. Hutton, G.: Higher-order Functions for Parsing. Journal of Functional Programming **2**(3) (1992) 323–343