Liskell

Haskell Semantics with Lisp Syntax

Clemens Fruhwirth
⟨clemens@endorphin.org⟩

ABSTRACT

This paper introduces Liskell, a new syntax for Haskell. Liskell belongs to the Lisp family of computer programming language when judged by its syntax, but is mostly Haskell when it comes to language semantics.

We argue that meta-programming in Haskell has not found wide-spread adoption because of the disparity between the abstract syntax tree and its visual appearance in source code form. Liskell uses an extremely minimalistic parse tree and shifts syntactic classification of parse tree parts to a later compiler stage to give parse tree transformers the opportunity to rewrite the parse trees being compiled. These transformers can be user supplied and loaded dynamically into the compiler to extend the language.

This paper introduces the Liskell syntax and serves as first draft for a language definition. We conclude the paper with a demonstration of meta-programming capabilities ranging from quasiquotation to an embedded version of Prolog. We implement Liskell as syntax frontend for the Glasgow Haskell Compiler. The implementation is publicly available from `http://clemens.endorphin.org/liskell`

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors

Keywords

Liskell, Haskell, Lisp, meta-programming, parse tree transformation

1. INTRODUCTION

Haskell is a lazy, functional and pure, strongly and statically typed programming language evolved by a mostly academic community since 1987. It features curried functions in the tradition of the λ-calculus, Hindley-Milner type inference, and a type class system to enable abstraction mechanism similar to the object-oriented paradigm.

The Haskell 98 Report[8] is the defining work for Haskell. We often refer to this work because the runtime semantics of Liskell are purely Haskell. For the brevity of this paper, we must assume the reader to have basic knowledge of Haskell. For Haskell textbooks, we refer to [14]. The Glasgow Haskell Compiler (GHC) is an implementation of Haskell 98, and we use GHC as the foundation for the implementation of Liskell.

Although sharing the λ-calculus as common ancestor, the Lisp family of programming languages is fairly distinct from Haskell. The most visual difference is its full parenthesized syntax known as symbolic expressions. On the semantic side, Lisp allows state modifications and side effects and therefore falls into the impure half of functional programming languages. Lisp is usually strongly but not statically typed, making runtime type checks mandatory. Compilers are allowed to do type inference, but they are in a much weaker position to come up with useful results, as the language design does not guarantee comprehensive type inference.

Haskell stands a long tradition of modeling the writing customs of the logic and mathematics community. This is not surprising as Haskell has a strong academic background and therefore most researchers extend the Haskell syntax in a way they are used to in writing. As appealing this syntax is for reasoning, it falls short when one has to think of it as an abstract syntax tree. This is rarely necessary for regular programming activities, but indispensable for meta-programming.

Meta-programming is available in Lisp via macros. We seldom find average sized an Lisp program that does not make use of macros. We think that the reason for the widespread adoption of meta-programming in Lisp and the slow acceptance of Template Haskell in the Haskell community lies within their syntactic differences.

Simplicity and uniformity. When the data structure for parse trees is simplified, writing parse tree transformers becomes easier in two aspects: Analysing the input parse tree, and generating the output parse tree.

Compared to Lisp, Haskell has a fairly complicated abstract syntax tree. Many specialized sub trees are needed to model a regular Haskell program. Template Haskell, Haskell's meta-programming facility, features 12 sub-tree types, with 48 data constructors.[1] In contrast, the Liskell parse tree structure has only one type with 2 constructors, namely one for a node and one for a leaf.

Template Haskell (TH) tries to hide its complexity by providing a language construct known as Quasiquotation, see [13]. Quasiquoting generates parts of the abstract syntax tree from regular Haskell syntax snip lets. This works smoothly as long as the programmer does not exceed the capability of this construct, such as providing an output with a variable number of parts. In such cases, the programmer has to build the syntax tree manually. Also quasiquotation only helps with generating but not with analysing parse trees.

Skills for free. Suppose we have a Lisp and a Haskell programmer that are intimate with their programming language. The Lisp programmer has no difficulties to see the Lisp parse tree behind the code he regularly writes, while the Haskell programmer has to deal with syntax tree objects, he is not used to from regular programming. Haskell meta-programming skills must be learned separately before they can be put to use, while for the Lisp programmer they come mostly for free.

Seamless integration. Lisp macros are not marked up differently compared to regular language constructs. In contrast, TH requires explicit markup for expressions evaluated at compile-time and forces the programmer to show details to the subsequent reading programmer that are potentially undesired. This mental distinction between regular code, compile-time code and built-in language constructors are mostly unnecessary and at worst confusing to the reader. Once the reader has understood the new syntax concepts added with meta-programming, a different syntactic annotation of meta-programming fragments prevents the reader from perceiving the new syntax as naturally embedded into the language.

Data equals code. While the "skills for free" argument depends clearly on a simple syntax tree, the seamless integration of TH as unannotated syntax could work too. The problem in that case is not so much with making it unambiguous when to invoke compile-time code generation, but in what form the arguments should be delivered to the code generator.

Often Lisp macros belong to a special class of code generators, namely code transformers that manipulate an arbitrary piece of code. These macros may be supplied with a mix of data – to guide the compile-time code generation – and an unevaluated code for verbatim inclusion in the generated code.

Assuming these compile-time macros receive both data and code, the compiler has to know at parse-time what parts of the parse tree are data and what parts are code. The consistence of this mix might eventually decided by macros itself, giving the parser no chance – as it rans before macro expansion – to deliver the macro with code elements parsed as code and data delivered raw.

[1]GHC internally uses 57 types and 227 constructors to represent a parsed Haskell syntax tree.

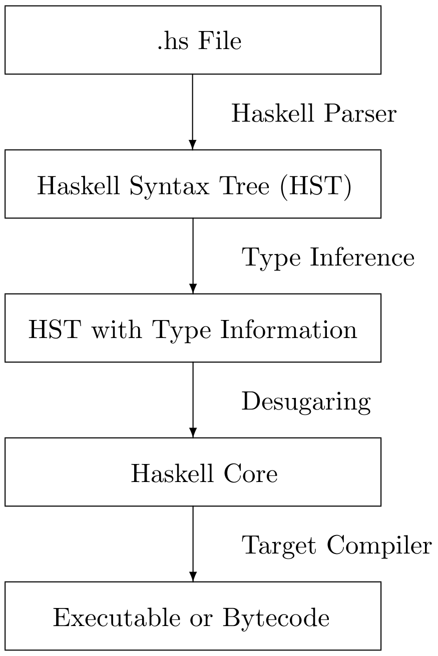

Figure 1: Inside a Haskell compiler

TH solves this by using markup for code parts, but this prevents seamless integration. In Lisp and in Liskell, the problem does not exist because the syntactic markup of data and code is identical.

2. DESIGN PRINCIPLES

Liskell is implemented in GHC as a lexer, parser and an in-compiler parse tree transformer.[2] The lexer and parser are extremely simple due to the simple nature of Liskell syntax. The parse tree transformer converts a Liskell parse tree into a Haskell syntax tree as it would be produced by the Haskell source parser. Due to this, Liskell can take advantage of all compiler facilities that have been written for the Haskell syntax tree.

Figure1 lays out the regular path a Haskell source takes through a Haskell compiler. The result from parsing is a Haskell syntax tree that is put through type inference and type checking. After type checking, the syntax tree is transformed into a syntax tree for a more primitive language called Haskell Core. The Core language is basically a fully-type annotated implementation of the second-order lambda calculus also known as System F or polymorphic lambda calculus. Backends in GHC produce the appropriate target code from the Core syntax tree, such as assembler code, C code, or bytecode.

To reuse this work flow within the Haskell compiler, Liskell targets the Haskell syntax tree as output, see Figure 2. But before the Liskell primitive transformations turn a Liskell parse tree (LPT) into a Haskell syntax tree, the user is able to provide parse tree transformers to rewrite LPTs arbitrarily.

[2]As part of GHC, Liskell falls under GHC's simple permissive MIT-styled license.

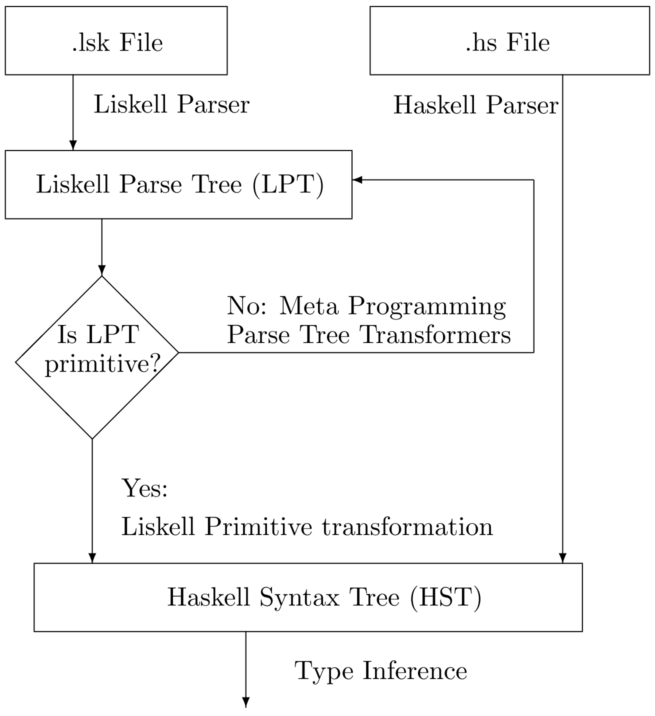
.lsk File
.hs File
Liskell Parser
Haskell Parser
Liskell Parse Tree (LPT)
Is LPT primitive?
No: Meta Programming Parse Tree Transformers
Yes: Liskell Primitive transformation
Haskell Syntax Tree (HST)
Type Inference

Figure 2: The Liskell source path

We realize Liskell meta-programming through the transformation loop for parse trees. Section 3 describes the Liskell primitive transformations for different syntax forms. Section 4 introduces the Liskell meta-programming model.

2.1 The common and the rare

Most Liskell syntax forms can be given as an atom or as a list. An atom is either a literal – character, string, integer or rational – or it is a symbol. Literals have the usual Lisp form: characters are prefixed with `#\`, strings are enclosed into double quotes, integers are written as a set of base 10 digits, rationals might contain a dot for their decimal part.

An atom usually retains its semantic meaning when wrapped in a singleton list. Whenever

$$atom \rightarrow (atom)$$

is true for a syntax form, we say that atoms of this syntax forms are auto-wrapped. Notice that this informal rule works only one way. There is no auto-unwrapping although there are syntax forms where over-parenthesized atoms happen to be semantically equivalent (as in expressions f.i.).

Syntax forms given as lists usually provide the opportunity to give more details, while a single atom is usually only a short form of the more elaborate list syntax. For instance in patterns, the atom `Nothing` has the same meaning as the singleton list `(Nothing)`. Both forms map to a pattern match with the zero-argument prefix constructor `Nothing`. When we need to give a prefix constructor pattern with at least one argument, we can not give the pattern as an atom but have to choose the list form as for instance in

```
(Tree left-leaf right-leaf)
```

The way strings are handled in most programming languages is a good example of "make the common case fast". The common case, namely writing characters, is made easy by the string syntax, while seldom used special or control characters are accessible with a special prefix `\`, the backslash character. A character sequence follows the backslash specifying a control or special character, for instance `\n` for line breaking. Usually the prefix character in its raw form is also available via a character sequence, in most cases `\\`. We employ the same concept for designing Liskell.

Expressions, patterns and types have specials forms that can be accessed via its list syntax. Whether a list is a special form or not is judged by the first element of the list, the list head. If the form is not special, all syntax forms have default meanings. For instance, the expression

```
(filter odd xs)
```

is not a special form, as its list head `filter` is not listed in Table 1. Therefore it matches the generic function application syntax, which is default for a list in the context of an expression.

The form

```
(if x False True)
```

is a special form resembling the semantics of `(not x)`. Its list head `if` is special and consulting Table 1 reveals that this expression is a conditional.

For the rare case, when specials shadow a desired default meaning, we provide – analogous to `\\` – a special that gives access to the shadowed syntax forms. A function application of *args* to a function named `lambda` can not be done by `(lambda` *args*`)`, because `lambda` is the special for closures. We have to use the special `app` to force function application: `(app lambda` *args*`)`.[3]

3. LANGUAGE DEFINITION

This section is a rather dry and dense definition of Liskell. Knowledge of the Haskell language definition [8] is recommended for this section, although we try to briefly explain the Haskell semantics where feasible.

This section defines the primitive Liskell transformations. All restrictions mentioned in this sections are with respect to a Liskell variant without meta-programming. We can lift any syntactic restriction whenever we are able to rewrite the offending parse tree into a conforming primitive parse tree via meta-programming.

3.1 The Lexer and The Parser

As any Lisp, Liskell is very easy to lex and parse. An Alex[10] generated lexer combined with a Happy[9] generated parser accounts for less then 150 lines of code, while

[3]The same effect can be achieved by referring to `lambda` in its qualified form as in `MyModule.lambda`, because specials only match unqualified symbols.

Form	See [8]	Semantic Meaning
lowid	3.2	Variable
Upid	3.2	Data constructor
literal	3.2	String, character, integer or fractional literal
(*expr* *expr**)	3.3	Function application, left-associative
(`lambda` (*pats**) *expr*)	3.3	Closure
(`if` *expr* *expr* *expr*)	3.4	Conditional, expr. order: *conditional*, *then*, *else*
(`[]` *expr**)	3.7	List
(`let` (*E -binder**) *expr*)	3.12	Let binding, see binding syntax 3.4
(`case` *expr* *matches**)	3.13	Case analysis, see match syntax 3.3
(`::` *expr* *type*)	3.16	Type cast
(`,` *expr**)	3.22	Tuple with *expr**
(`app` *expr* *expr**)	none	Explicit application

Table 1: Expression Syntax

GHC needs about 3400 lines of Alex and Happy code to parse Haskell.

Haskell distinguishes between *varid* and *conid* symbols. The former starts with a lowercase character, while *conid* starts with a capital letter (see [8] 2.4). Liskell keeps this simple identifier distinction.

Liskell is more liberal when it comes to identifiers for symbols. The set of identifiers is larger than the set of possible identifiers for Haskell, and so a part of the Liskell identifier set can only be reused in Liskell modules. An example for this would be identifiers containing dashes that are quite popular among Lisp programmers. Identifiers such as `print-object` can not be used in Haskell, because the Haskell parser sees an infix minus operator here.

We use the meta-variable *symbol* to refer to the whole range of symbols, *Upid* to refer to an identifier starting with a capital letter or a colon. *lowid* denotes all non-*Upid* identifiers.

A symbol can be qualified by putting the module name separated by a dot in front of its identifier. The semantics of qualified names are defined in Chapter 5 of [8].

3.2 Expressions

Expressions defined in Table 1 are valid for all meta-variables named *expr*. Most specials are documented properly in the Haskell Report [8]. The syntax forms given in tables 1-6 are always matched starting the most specialized to the least specific on top of the tables.

An expression is either an atom or a list. Atoms are integral and fractional numbers, strings, characters and symbols. With respect to overloading, all atoms that are not symbols are treated in the same way as in Haskell, see 6.4.1 in [8].

A list is usually an application of a function to a series of arguments, unless the head of the list is special. Because functions are fully curried in Haskell, a multi-argument function application is actually a series of left-associatively constructed sequence of applications. Hence

$$\texttt{(f a b c)} \rightarrow \texttt{(((f a) b) c)}$$

which is not true for most Lisp dialects.

In contrast to Lisp, `(fun)` does not cause a function invocation of `fun` with zero arguments. Haskell knows no functions without arguments. `(fun)` is best thought of as `(id fun)`, the application `fun` to `id`.

3.3 Patterns

A unification algorithm answers the questions whether two meta-variable containing terms can be made equal by substituting these meta-variables for arbitrary terms. Either unification fails or the answer is a set of substitutions. Matching is a restricted form of unification where only one term is allowed to contain meta-variables. Pattern matching is mainly used to destructure algebraic data types (ADT).

In the pattern syntax, the "variable pattern" denotes a meta-variable. When pattern matching succeeds, the result is not a substitution set, but more conveniently, variables with the same identifiers as the meta-variables are bound in the lexical scope to the respective substitution term.

The semantics of pattern matching in Liskell are directly derived from Haskell semantics. See Section 3.17.2 of [8] on the informal semantics of pattern matching.

In case statements, we use the minor syntax form *match*. It is the mere aggregation of a pattern *pat* and an expression *expr* into a list (*pat* *expr*) with no form shorter than that.

3.4 Bindings

Binders are a minor syntax form that is used in top-level binders and in the lexically scoped binder `let`. There are two syntax forms for binders: *E-binders* and *ET/TE-funbinders*. The latter is a restricted version of an *E-binder* that is only able to bind functions but in exchange for that allows type annotation for functions.

3.4.1 The generic E-binder

An *E-binder* is a syntax form that assigns values to identifiers. The following syntax

(*binding* *expr*)

creates a binder for *binding* bound to *expr* in the lexical environment when used with `let` and in the global environment when used with `define`.

Form	§3.17.2 [8]	Semantics
_	3	Wildcard pattern
lowid	1	Variable pattern
literal	7	Literal pattern
Upid	4, 5	Constructor pattern with zero arguments
(*Upid* *pat**)	4,5	Prefix constructor pattern
(@ *lowid* *pat*)	9	As-pattern with named *lowid*, and *pat* as payload pattern
(~ *pat*)	2	Lazy pattern
([] *pat**)	–	List pattern
(, *pat**)	–	Tuple pattern
(Con *symbol* *pat**)	4,5	Constructor pattern with prefix *symbol*

Table 2: Pattern Syntax

lowid	Type variable
Upid	Type constructor with no arguments
(*symbol* *type**)	Application (left associative)
(-> *type* *type+*)	Function type (right associative)
(=> *context* *type*)	Constrained type
([] *type*)	List type
(, *type**)	Tuple type
(! *type*)	Bang type
(app *symbol* *type**)	Application (left associative)

Table 3: Type Syntax

Haskell knows two kinds of binders, function binders and pattern binders. All patterns shown in the last section can be part of a pattern binder except the variable pattern. A primitive variable is modeled by a zero-argument function binder. Liskell differentiates between the binder types by the syntactic form of their *binding* part.

- If *binding* is a single symbol, it is interpreted as binding for a primitive variable. Emit a function binder.
- If *binding* is recognized as a pattern (see 3.3), emit a pattern binder.
- Otherwise emit a function binder. This can only happen when *binding* has the form

  $$(lowid\ \ p_1\ \ \ldots\ \ p_2)$$

  This binds a function named *lowid* with $p_1 \ldots p_n$ arguments, where p_i are patterns.

3.4.2 ET/TE funbinder

In some cases, we need to type annotate the binding. We introduce two binders that are only valid for function binders and are syntactic variants of each other. They can be used to explicitly give signatures to top-level functions or to class methods.

We use the meta-variable *ET-funbinder* for the first variant

(*binding* *expr* [*type*])

which is practically an *E-binder* extended with an optional type. The second variant, the *TE-funbinder*, just flips the positions of the type and the expression making the latter optional,

(*binding* *type* [*expr*])

In both variants *binding* must match the last section's rules 1 or 3 for function binders.

The reason for providing two variants is that there are cases where we want to omit the type and others where we want to omit the expression. Examples for the former are functions and instance declarations, examples for the latter are class definitions.

3.5 Types

In the context of types, a parse tree consisting of a symbol represents a type variable or a type constructor. Type variables are symbols with *lowid* while type constructors are written as *Upid*. Liskell interprets a list as left-associative type application. The specials for types are listed in Table 3. They include the right-associative type constructor for functions, i.e.

(-> t1 t2 t3) → (-> t1 (-> t2 t3)))

3.5.1 Constrained type forms

Haskell syntax contains a number of type forms that are syntactically restricted. They are defined in [8] 4.1-4.3. The same restrictions apply to their Liskell counterparts but for a few forms the syntax is different. This section aims to give a brief description of these differences without explaining the concepts below. Understanding these concepts is only necessary for programming with type classes.

The syntactic appearance of *class* and *simpleclass* predicates as well as *simpletypes*, *newconstr* and *qtycls* does not change. *constr* is only allowed to appear in its prefix form, and any occurrence of strictness as in !*atype* has to be replaced by

`(defmodule` *Upid exports* `(`*import**`))`	Defines a module
`(define .` *E-binder*`)`	Value definition including patterns.
`(define .` *ET-funbinder*`)`	Value definition and/or signature definition
`(defwithsig .` *TE-binder*`)`	Signature definition and/or value definition
`(deftype` *simple-type type*`)`	Type synonym
`(defnewtype` *simple-type*$\|_{context}$ *newconstr*`)`	newtype definition
`(defdata` *simple-type*$\|_{context}$ `(`*constr**`))`	Algebraic Data Type
`(defclass` *simple-type*$\|_{scontext}$ *TE-funbinder**`)`	Class definition
`(definstance (`*qtycls inst*`)`$\|_{scontext}$ *ET-funbinder**`)`	Instance declaration

Table 4: Top level specials

`(!` *atype*`)`. *inst* is only allowed in its type application form or as *gtycon*.

A context is a collection of predicates given as list. Predicates are either represented in the *class* or *simpleclass* syntax. If the context consists only of a single predicate, the predicate is auto-wrapped to form a single-element context. Due to the restrictions on the forms of predicates this is an unambiguous transformation, as the first element has to be an *Upid* by the syntax of *scontext* and *context*.

The two type forms *simpletype* and *inst* might be constrained by an explicit context. As there are two context variants, *scontext* and *context* we indicate which one is allowed by typesetting the context bar-separated in subscript, as for instance *simpletype*$|_{context}$.

3.6 Top Level

At top-level, we see only special syntax forms. There is no default interpretation of a list with an unknown list head. Symbols are invalid at top-level.

3.6.1 Modules

Every Liskell source file starts with a `defmodule` statement, defining the module's name, its exports and its imports. The objects for import and export are specified with the *iename* syntax.

exports is either _ or `(`*iename**`)`. In the former case, the wildcard pattern symbol indicates that the module exports its entire variable, function, type and class content. In latter case, the list of *iename* enumerates all objects for export.

The forms of *import* are given in Table 6. Either it is an *Upid* naming a module which should be imported entirely, or it is a list naming the objects that should be imported from *Upid*. The *iename** list can start with a flag list. Possible import flags are `hiding`, `qualified`, and `(as` *Upid*`)` with semantics as given in Chapter 5 of [8].

3.6.2 Definitions

define, defwithsig. Both forms define bindings for identifiers. `define` gives an expression with an optional type signature, while for `defwithsig` the type signature is mandatory and the expression is optional.

deftype, defnewtype, defdata. These constructs define type synonym, new types, algebraic data types (ADT). All of these constructs inherit the semantics of Haskell. An ADT is

lowid	variable or function
Upid	type/class
`(`*Upid* _`)`	class/type with all methods/constructors
`(`*Upid lowid**`)`	class with method named *lowid**
`(`*Upid UpId**`)`	type *Upid* with type constructors *Upid**
`(module` *UpId*`)`	module content (export only)

Table 5: *iename* Syntax

a type that can be instantiated with one of possibly many constructors. Every constructor is wrapping zero or more payload types. The only way to access the payload is by unwrapping it via pattern matching. An ADT can also be described as a sum type over product types.

defclass, definstance. Class declarations define the methods a member type must have. Usually only method signatures are given in the declaration, while default methods are optional. *TE-funbinder*s fit in perfectly here. For `definstance`, we use *ET-funbinder*s as with them expressions are mandatory but type annotations are optional.

3.7 Examples

The previous sections defined the core of Liskell, its primitive syntax. The primitive syntax forms are directly translated into GHC fragments of the Haskell syntax tree.

This section serves as show case to give an impression how Liskell looks for simple programming examples.

```
(defmodule Main _ ())  ; export all, no imports

(define main
  (print "Hello World"))

(define (fact n)
  (if (== n 0)
      1
      (* n (fact (- n 1)))))

(define (quicksort xs)
  (case xs
    ([]) ([])           ; nil in gives nil out
    ((: x xs)
     (++ (quicksort (filter (< x) xs))
         (: x
            (quicksort (filter (>= x) xs)))))))
```

Notice the recursive definition of the Fibonacci numbers.

```
(define fibs (++ ([] 0 1)
                 (zipWith + fibs (tail fibs))))
```

Upid	all symbols of module *Upid*
(*Upid* *iename**)	*iename** symbols from module *Upid*
(*Upid* (`flags` *import-flags**) *iename**)	as above with import flags

Table 6: *import* Syntax

Figure 3 gives a computer player for the game Tic Tac Toe. The full source is available as part of the Liskell regression test suite.[4]

Figure 4 features a direct comparison of one of Tic Tac Toe's functions: `add-move`. The semantic of the Haskell – on the left – and Liskell code – in the middle – are identical. Both use pattern matching to destructure a tuple in the function arguments, carry out a case analysis on the player argument given as character, and reassemble a tuple with either the first or the second tuple part modified by the `insert` function.

The Common Lisp (CL) variant on the right works similarly, with the exception that it does not use a tuple but an untyped cons cells, and that it has to use accessors functions as CL lacks destructuring in function arguments.

4. META-PROGRAMMING IN LISKELL

Meta-programming is the strategy of writing code that writes code. The development of high level programming languages such as Haskell can be seen as a form of meta-programming on top of backend languages such as assembler or C. More visible applications of meta-programming are parser generators, object persistence mapping tools, and remote procedure call stub generators.

We seldom find languages that feature meta-programming within the language itself. The Lisp family of computer programming languages is an exception to that and is by far the most successful language family to employ meta-programming in daily programming.

Liskell brings meta-programming to the Haskell world by introducing the concept of parse tree transformers (PTT). This concept is more general than the meta-programming facilities `defmacro` and `symbol-macrolet` in Common Lisp. Liskell provides a single hook into the compiler, namely `envlet`.

4.1 Liskell Parse Trees

Before we can talk about parse tree transformers (PTTs), it is good to get familiar with the data structures of parse trees. We define a simplified version of the Liskell parse tree for demonstration purposes:

```
(defdata SParseTree
  (SSym String)
  (SList ([] SParseTree)))
```

This defines an algebraic data type called `SParseTree` with the constructors `SSym` for `String`-carrying leaves and `SList` for nodes with a list of child nodes.

[4]`http://clemens.endorphin.org/testsuite-liskell/tests/liskell/testprogs/TicTacToe.lsk`

The expression

```
(define (double x) (+ x x))
```

has the parsetree form

```
(SList ([] (SSym "define")
           (SList ([] (SSym "double")
                      (SSym "x")))
           (SList ([] (SSym "+")
                      (SSym "x")
                      (SSym "x")))))
```

Every symbol corresponds to an instance of `SSym` while every parse tree list is represented by `SList`. The next section describes how to operate with this data structure.

4.2 Parse Tree Transformers

A parse tree transformer is a function that maps from one parse tree to another. Using our simplified version of the parse tree, the function signature of a transformer is

```
(-> SParseTree SParseTree)
```

To give a practical example, we introduce a PTT that enables us to use syntax for `cond`, a multi-conditional version of `if`. In tradition of CL [2], we want to use `cond` in the following way:

```
(cond ((< a b) LT)
      ((> a b) GT)
      (True EQ))
```

The PTT should rewrite this `cond` statement into a series of `if`s. For the example above, the result should be:

```
(if (< a b)
    LT
    (if (> a b)
        GT
        (if True
            EQ
            undefined)))
```

The first objective of the PTT is to look for syntax with the shape `(cond ...)`. When we pattern match the given parse tree against

```
(SList (: (SSym "cond") clauses))
```

we not only find this specific form, but also extract the list of clauses into the variable `clauses`.

Every clause is composed of a guard and an action, and for each we generate the following code expansion

(`if` *guard action rest-of-expansion*)

For an empty set of clauses, we emit `(SSym "undefined")`, which stands for the undefined symbol in Haskell that when evaluated will cause a run-time error.

```
(define (computer-player state ego)
  (case (game-over? state)
    ((Just a) (, undefined a))
    (Nothing (let ((results (map (lambda (move)
                                   (let ((new-state (board-add-move state move ego))
                                        ((, _ winner) (computer-player new-state
                                                                       (opponent ego))))
                                     (, move winner)))
                                 (Data.Set.toList (board-valid-moves state))))
                   (rate-with (case ego
                                (#\a rate-outcome-a)
                                (#\b (- rate-outcome-a))))
                   (sorted-results (sortBy (lambda (result1 result2)
                                             (compare (rate-with result2)
                                                      (rate-with result1)))
                                           results)))
               (head sorted-results)))))   ;; return the best move
```

Figure 3: Tic Tac Toe computer player in Liskell

```
add_move (stateA, stateB)
         move
         player =
 case player of
  'a' -> (insert move stateA,
          stateB)
  'b' -> (stateA,
          insert move stateB)
```

```
(define (add-move (, stateA
                     stateB)
                  move
                  player)
  (case player
    (#\a (, (insert move
                    stateA)
            stateB))
    (#\b (, stateA
            (insert move
                    stateB)))))
```

```
(defun add-move (state move player)
  (case player
    (a (cons (cons move
                   (car state))
             (cdr state)))
    (b (cons (car state)
             (cons move
                   (cdr state))))))
```

Figure 4: Haskell vs. Liskell vs. Common Lisp

```
(define (cond-ptt pt)
  (let (((trf-cond ([]) (SSym "undefined"))
        ((trf-cond (: (SList ([] guard action))
                      rest)
           `(if ,guard
                ,action
                ,(trf-cond rest))))
        ((trf-cond _) (error "Invalid cond")))
    (case pt
      ((SList (: (SSym "cond") rest))
       (trf-cond rest)))
      (_ pt))
```

This code expands all `cond` forms and passes along all other forms unmodified. Notice that we use a new facility in the `cond-ptt` function, namely backquoting. Basically backquoting is syntax sugar to give us a convenient code template mechanism. The statement

```
`(if ,guard ,action ,(trf-cond rest))
```

is equivalent to

```
(SList ([] (SSym "if")
           guard
           action
           (trf-cond rest)))
```

Section 5.4 introduces the backquoting facility in detail.

4.3 The real model

A real Liskell PTT is not that simple. The Liskell compiler environment contains entry points to four PTTs for the four major syntax classes. The Liskell compiler invokes the appropriate function in the compiler environment whenever it has to transform an expression, a pattern, a type or a top-level declaration. The programmer is able to modify these entry points. The regular approach is to replace one entry point by a user-supplied function that calls the old entry point when it finishes its parse tree transformation.

In Liskell, a PTT is obligated to communicate whether it transformed the parse tree or not. The reason is that for modified parse trees the parse tree transformation must be restarted, so embedded syntax sugar is expanded further. PTTs do not have such a simplistic signature, as the one given in the last section. Parse tree transformation happens in continuation-passing style. A user supplied PTT is invoked with two continuations, one for successful transformations, and one continutation the PTT has to call when it has not modified the parse tree. As the transformation proceeds, the syntax tree becomes more expanded and finally converges to primitive Liskell syntax forms.

The Liskell primitives are PTTs themselves with the distinction that they do not transform a Liskell parse tree into another Liskell parse tree as user supplied PTTs would do, but they transform primitive Liskell syntax into Haskell syntax trees.

4.4 Environment Transformers

To become an active part of the compiler, a PTT has to be hooked into the Liskell compiler environment. At the moment, Liskell knows four transformers: expression-, pattern-, type- and top-level declaration-transformers.

To add a transformer to the compiler environment there are two special forms, `defenv` and `envlet`, while `defenv` is only syntax sugar for the latter. The former modifies the compiler environment of all following top-level declarations, while the

`(defenv` *fun*`)`	redefines the current compiler environment.
`(envlet` *fun decls/expr*`)`	redefines the compiler environment for the enclosed declarations or the enclosed expression.

Table 7: Meta-programming forms

latter modifies the environment only for the enclosed declarations. `envlet` can also be used in an expression context, while in this case the enclosed form must be an expression.

The Liskell compiler environment is stored in the simple product type

`(LskEnv` *exprtrf pattrf typetrf decltrf*`)`

where the all transformation functions *trf* have the signature

$$\texttt{(-> ParseTree }\underbrace{\texttt{(-> ParseTree a)}}_{\text{success continuation}}\ \underbrace{\texttt{(-> ParseTree a)}}_{\text{failure continuation}}\texttt{ a)}$$

4.4.1 *cond-ptt revised*

The previous `cond` PTT needs a few modifications to work with the real Liskell compiler model. The modifications are typeset in *italic*. First, we need to call continuations for succeeded and failed transformations.

```
(define (cond-ptt-1 pt ks kf)
  (let ...
    (case pt
      ((SList (: (SSym "cond") clauses))
       (ks (trf-cond clauses))))
      (_ (kf pt)))
```

To activate this PTT, we use the following compiler environment transformer

```
(defenv (lambda ((LskEnv e p t d))
          (return (LskEnv cond-ptt-1 p t d))))
```

Now only one problem remains, namely that the old entry point `e` is discarded and that the only active PTT is the `cond` PTT. Usually, we want more than one PTT to be active, and so we need to create a chain of PTTs as mentioned before. We can achieve this by replacing the entry point with a function partially applied to the old entry point. This way the new PTT gets a reference to the old entry point which it can call for failed transformations. The failure continuation is not called directly by the new PTT and only passed through to the old PTT.

```
(define (cond-ptt-2 kn pt ks kf)
  (let ...
    (case pt
      ((SList (: (SSym "cond") clauses))
       (ks (trf-cond clauses)))
      (_ (kn pt ks kf)))))
```

The invocation of `defenv` changes to

```
(defenv (lambda ((LskEnv e p t d))
          (return (LskEnv (cond-ptt-2 e) p t d))))
```

5. THE LISKELL PRELUDE

The language defined by the Liskell primitives is very sparse and not convenient for daily programming. The Liskell Prelude provides the programmer with a set of convenience functions and syntax sugar that eases programming. Similiar to the Haskell Prelude – which is imported too – the Liskell Prelude's module `LskPrelude` is imported with every Liskell source implicitly.

5.1 Simple List

The simple list PTT transforms lists prefixed with a procent sign % into an explicit list.[5] After injecting this PTT, we can write `([] a b c)` in a shorter form

```
%(a b c)
```

The simple list syntax sugar also rewrites the symbol `nil` into `([])`. This is transformer can be used for patterns too. Be aware that when you see `nil` as symbol in a pattern match, it might not denote a meta-variable, but in fact a match for the empty list.

5.2 The Dispatcher Namespace

In Liskell terminology, the PTT part which decides whether to modify a parse tree or not is called dispatcher. Invocations of `defenv` or `envlet` always hook the dispatcher part into the compiler environment. The Liskell Prelude tries to keep dispatchers in a different namespace than regular functions.

The dispatcher namespace can be access by wrapping a symbol into a "d" prefixed list such as `d(dispatcher)`. The Prelude defines two top-level declarations for the dispatcher namespace:

- `(define-dspr (`*function args* ...`)` *body*`)` is equal to `define` except that a new dispatcher function is defined in the dispatcher namespace.
- `(add-dspr (`*syntax-class dispatcher*`)`***`)` – `add-dspr` is syntax sugar for `defenv`. It adds multiple dispatchers to the compiler environment. The statement

  ```
  (add-dspr (expression simple-list)
            (pattern simple-list)
            (declaration defmacros))
  ```

  adds the `simple-list` syntax sugar to expressions and patterns, and `defmacro` syntax to toplevel declarations. `expression`, `pattern`, `type` and `declaration` are valid choices for *syntax-class*.

5.3 Quoting

Liskell does not modify the Haskell evaluation semantics and as Haskell does not know a quote operator, Liskell has to introduce its own. It provides the convenience function `quote` that turns a `ParseTree` into another `ParseTree` that when evaluated results in the original `ParseTree`.

The code fragment `(a b)` is represented by the parse tree

```
(SList ([] (SSym "a") (SSym "b")))
```

[5] % was a totally arbitrary choice.

In fact, we just quoted the expression `(a b)` by giving an expression that when evaluated results in a parse tree representing `(a b)`.

Let us look what we just did. For the symbols `a b`, we created an application of `SSym` to its identifier. We can do this easily with the function:[6]

```
(define (qSSym (SSym id))
  (SList %((SSym "SSym") (SSym (show id)))))
```

The use of `show` adds doulbe quotes around `id` marking it a string. For lists, we have to create an application of `SList` to a list containing quoted parse tree elements. With the help of the function `quote` – defined in a second – we can realize quoting for lists with:

```
(define (qSList (SList lst))
  (SList %((SSym "SList")
           (SList (: (SSym "[]")
                     (map quote lst))))))
```

With `qSSym` and `qSList` we covered all possible syntax forms of `SParseTree` and can write the function `quote` that dispatches to these two helper functions:

```
(define (quote pt)
  (case pt
    ((SSym _) (qSSym pt))
    ((SList _) (qSList pt))))
```

The evaluation of `(quote (a b))` is equivalent to the evaluation of:

```
(SList
  ([] (SSym "SList")
      (SList
        ([] (SSym "[]")
            (SList ([] (SSym "SSym")
                       (SSym "\"a\""))
            (SList ([] (SSym "SSym")
                       (SSym "\"b\""))))))))
```

5.4 Quasiquotation

With the basic form of quoting defined, the Liskell Prelude is able to provide one of the most important convenience facility to meta-programming: quasiquotation also known as backquoting. In [4], Bawden describes quasiquotation as "... a parametrized version of ordinary quotation, where instead of specifying a value exactly, some holes are left to be filled in later". Backquoting can be seen as a template mechanism for code that needs to produce a parse tree as output. In most cases, we need bits of the output parse tree to be variable.

In general, all parse tree elements prefixed via backquoting are quoted with the `quote` function from the last section. Unquoting creates an exception from that – it generates the "holes" Bawden refers to. Comma-prefixed parse trees are not subject to quoting. The most elementary form of unquoting – a very pathological case – is the immediate unquoting of a backquoted symbol such as `‘,a` which is equivalent to `a`. More commonly, we see unquoting inside a backquoted list such as

[6]The function for symbol quoting found in `LskPrelude` is different from the one given, as the syntax for `SList` and `SSym` is defined later in the Liskell Prelude and hence can not be used.

```
‘(if ,guard ,action ...)
```

This returns a parse tree with `,guard` and `,action` replaced by the parse trees snip lets contained in the variables `guard` and `action`.

Backquoting also supports an operator that allows splicing parse tree lists, namely `,@`.

```
(let ((pt-lst ‘(b c d)))
  ‘(a ,@pt-lst e))
```

returns a result equivalent to `‘(a b c d e)`. While the result of

```
‘(a ,pt-lst e)
```

is equivalent to `‘(a (b c d) e)`.

Backquoting can be nested, and by repeated use of unquoting, we can create complete expressions such as:

```
(let ((level0 ‘c))
  ‘(let ((level1 ‘d))
     ‘(a b c ,,level0 ,level1 ,,(if level0-var
                                    ‘‘(a b)
                                    ‘‘(b a)))))
```

We define the backquoting operator as follows:

Definition 1. `‘`*pt* is equal to `(quote` *pt*`)` except for two cases:

- `‘,`*pt* is equal to *pt*
- when *pt* is a list. Then `‘(`*pt1 pt2 pt3* `..)` is equal to `(SList (concat ([]` {*pt1*} {*pt2*} {*pt3*}`)))`.

In the definition we us the a new curly-bracket operator that is defined as follows:

Definition 2. {*pt*} is equal to singleton list `([] ‘`*pt*`)` except in one case, namely {`,@`*pt*} is equal to `(pt_list` *pt*`)`[7].

A sketch of the `LskPrelude` backquoting implementation is given in Figure 5[8]. The implementation of backquoting makes use of the following properties for the `Maybe` monad type, which has the zero-argument constructor `Nothing` and the one-argument constructor `Just` *value*:

- `(>>= Nothing` *fun*`)` ≡ `Nothing`.
- `(>>= (Just` *value*`)` *fun*`)` ≡ `(`*fun value*`)`.

[7]`pt_list` returns the parse tree list wrapped by `SList` and `PList`

[8]Paradoxically the backquote operator seems to use itself in its definition as it contains backquoted symbol. The reason why this works, and is contained in LskPrelude as it is written here, is that prior to this definition we defined a primitive version of backquoting that works only for symbols. We defined syntax sugar to ease the definition of syntax sugar.

```
(define (bq-quote ks pt)
  (let ((pt-expand (bq-dispatch (lambda (pt ks kf) pt) pt id undefined)))
    (Just (ks (fromMaybe (case pt-expand
                           ((PList loc lst pp)
                            (qPList loc
                                    (SList %('concat
                                             (SList (: '[] (map bq-bracket lst)))))
                                    pp))
                           (_ (quote pt-expand)))
                         (comma? pt-expand)))))))

(define (bq-dispatch kn pt ks kf)
  (fromMaybe (kn pt ks kf)
             (>>= (quoted? pt)
                  (bq-quote ks))))

(define (bq-bracket pt)
  (fromMaybe (SList %('[] (bq-quote pt)))
             (comma-at? pt)))
```

Figure 5: Backquoting implemented as Parse Tree Transformer

- `(fromMaybe` *default* `Nothing)` ≡ *default*.
- `(fromMaybe` *default* `(Just` *value*`))` ≡ *value*.

A computation in the Maybe monad might fail, and the combination operator `>>=` only carries on with the computation when no failure (`Nothing`) has been encountered. `fromMaybe` provides wraps a Maybe monad and provides a default value for failed computations.

The functions `quote?`, `comma?` and `comma-at?` are predicates that either return a parse tree suitable for further processing or the predicate fails by returning `Nothing`. `quote?` tests whether the parse tree it gets is backquoted, and if yes, returns a parse tree stripped from the quote. `comma?` and `comma-at?` check for a comma or a comma-at prefix.

The `bq-dispatch` function only dispatches on parse tree elements that are accepted by `quoted?`, otherwise the parse tree is sent to the `kn` continuation. The backquoted parse tree is handed to `bq-quote` without its prefix, which further expands any backquoting by recursively calling `bq-dispatch` with identity continuations.

`bq-quote` is moderated by the `comma?` predicate. If `comma?` returns `Nothing`, either `bq-bracket` processes all list elements, if the parse tree is a list, or if it is not, `bq-quote` returns the parse tree `quoted`. `bq-quote` always calls the given success continuation and signals a success by handing the return value of `ks` wrapped in `Just` to its caller.

`bq-bracket` works similarly, but only for the curly bracket operator, and the list splicing operator `,@`. Curly bracket elements are either returned quoted in a singleton list, or when list splicing is detected by `comma-at?`, the respective unquoted element is returned.

5.4.1 The Parseable Type Class

The predicates `comma?` and `comma-at?` not only test for comma prefixed elements, but also return parse trees that are fit for the inclusion in the output code of the backquote PTT. Both predicates wrap the returned parse tree in

`(toParsetree` *pt*`)`

The method `toParseTree` is part of the type class `Parseable`. This type class requires a member type t to define a method `toParseTree` that has the type signature

`(-> t ParseTree)`

namely a function that converts the member type t into a parse tree. Liskell defines the types `Integer`, `Char`, `Rational`, and `ParseTree` and lists of `Parseable`s to be part of the type class `Parseable` providing a method `toParseTree` to convert these type into a `ParseTree`.

With this facility, all `Parseable` types can be embedded into a backquoted list without wrapping, such as this integer example

```
(let ((a 1)) '(+ ,a ,a))
```

Users can provide `Parseable` instances for their types to enable automatic and unannotated lifting into backquoted expressions.

5.5 defmacro

The Liskell prelude contains syntax sugar for defining macros similar to CL macros. The syntax for `defmacro` is

`(defmacro (`*macro-name pts*`)` *body*`)`

and the result is that a dispatcher with the name *macro-name* is defined that dispatches all parse trees of the form (*macro-name* ...) to the macro's *body*, binding *pts* to everything that follows *macro-name* in the matched parse tree. We will see examples of `defmacro` in the next section.

5.6 Derive – Type Class Instance Generation

The Haskell 98 standard defines syntax sugar to automatically generate instances of algebraic data types in types classes. But this syntax sugar is limited to 6 type classes. Liskell does not include any `deriving` syntax in its primitive syntax, but adds this comfort via its Prelude. For derive syntax sugar, PTTs work in a team. The `defdata-deriving` syntax extracts the deriving directive included within a `defdata` statement into separate top level declarations.

To give an example,

```
(defdata ZenTree Leaf (Node ([] ZenTree))
  (deriving Eq Show))
```

is split and normalized by the `defdata-deriving` PTT into

```
(defdata ZenTree (Leaf) (Node ([] ZenTree)))
(derive Eq
  ZenTree (Leaf) (Node ([] ZenTree)))
(derive Show
  ZenTree (Leaf) (Node ([] ZenTree)))
```

For every instance template, there is a PTT producing the template code. For example, `derive-eq` dispatches on top-level declarations of the form

```
(derive Eq ..)
```

and replaces this top-level declaration by the appropriate `definstance` declaration. The user can supply PTTs that dispatch on any arbitrary second symbol in the `(derive ..)` top-level declaration, hence he can extend this deriving mechanism easily.

5.7 Field Syntax: sugar for updates

Haskell provides syntax sugar for updating certain parts of ADTs. The Liskell Prelude provides a facility to define syntax for field like updates.[9] The `defdataf` top-level declaration is identical to `defdata` with the exception that the *constr* meta-variable is substituted by

(*Upid* (*field-name type*) *)

where *Upid* is the constructor name – as before – but the payload types of the constructor have to be paired with *lowid* field identifiers. `defdataf` defines a PTT with the name identical to the type that dispatches on the form

(`update` *type-name obj* (*field-name new-value*) *)

The semantics of the parse tree generated in place of this `update` form is modelled along Haskell field syntax.

5.8 Infix to multi-argument prefix

Binary infix operators are convenient because they can unambiguously combine a complex expression solely moderated by precedence levels and associativity declarations. The expression $a + b + c + d$ must be written in Liskell with the cumbersome use of parenthesized sub-expressions

```
(+ a (+ b (+ c d))
```

CL provides an uncurried function calling syntax which allows multi-argument functions. Liskell, as it is based on the curried function convention of Haskell, can not work this way. But we can reintroduce multi-argument syntax sugar with PTTs. This way, we regain the flexibility of associativity from binary operators by transforming them into a multi-argument prefix macro.

Assume there is a PTT that rewrites `(+ a b c d)` into the correctly parenthesized version as given above. PTTs can process parse tree lists of variable length, and therefore they can also implement macros with multiple arguments.

[9]At the time of this writing, this has not yet been implemented.

```
(defmacro (++ pts)
  (case pts
    (%(one two) `(+ ,one ,two))
    ((: head tail) `(+ ,head (++ ,@tail)))))
```

This macro provides a new multi-argument prefix macro `++` that right associatively generates a sequence of additions. Notice that the macro for `++` emits a parse tree containing a reference to itself.

In general, we can define multi-argument versions of all binary infix operators. This leads us straight to higher order parse tree transformers, PTTs that generate PTTs. With the higher order PTT, shown in Figure 6, we can lift every Haskell infix operator to a multi-argument Liskell prefix macro.

```
(def-binary-fun-as-prefix + left)
(def-binary-fun-as-prefix / left)
(def-binary-fun-as-prefix : right)
```

6. APPLICATIONS OF META-PROGRAMMING

6.1 Convenience

In the last section we introduced a simplified version of the Liskell parse tree named `SParseTree`. This structure does not exist, but when you look into the source of the LskPrelude, you encounter the use of `SSym` and `SList` just in the manner, as we defined it in the last section. In fact, `SSym` and `SList` are macros providing the illusion of a simple data structure on top of the slightly more flexible underlying data structure `ParseTree`. `ParseTree` includes an additional data structure `SrcSpan` which indicates the point of definition of this parse tree element. This information is of almost no value to meta-programming, but must still be present in the Liskell internal parse tree to generate the Haskell syntax tree with source location information for meaningful error reporting. `ParseTree` also includes a tuple of two strings for lists that contain the strings prefixing or postfixing the list brackets as in `prefix(1 2 3)postfix`. These strings are used seldom – although we have seen a user, namely the backquoting PTT – and so they are only optional for the `SList` syntax sugar.

The following code snip let provides the convenience macro for `SSym`:

```
(defmacro (SSym %(pt))
  `(PSym noSrcSpan ,pt))
```

6.2 Language Extension

Haskell does not provide a language construct that allows the programmer to determinate (in short an simple manner) whether a specific pattern matches. We can define a short macro that gives us a new syntax form (`~=` *pat*) for such cases.

```
(defmacro (~= %(pat))
  `(let (((comp ,pat) True)
         ((comp _) False))
     comp))
```

6.3 Shifting computations to compile-time

Shifting computation to compile time leads to an increase in run-time performance. Partial evaluation is a powerful technique in this area but it is not yet available in standard compilers. Meta-programming can accommodate the programmer when this kind of optimization is required.

```
(defmacro-decl (def-binary-fun-as-prefix %(function associativity))
  (let ((function* (mapSymId function (lambda (s) (++ s "*")))))
    %(`(defmacro (,function-new-name pts)
         (case (length pts)
           (0 (error "Empty infix lift"))
           (1 (error (++ "Singular infix lift" (show pts))))
           (2 `(,,(quote function) ,@pts))
           (_ ,(case associativity
                 ((SSym "right")
                  ``(,,(quote function) ,(head pts) (,,(quote function*) ,@(tail pts))))
                 ((SSym "left")
                  ``(,,(quote function) (,,(quote function*) ,@(init pts)) ,(last pts))))))))))
```

Figure 6: def-binary-fun-as-prefix

The following example resembles the demonstration of Bezier curve optimization given in [7] Section 13.2.

A cubic Bezier curve is defined by a start and an end point, as well as two control point. The following equation defines the x coordinates of all points on the Bezier curve:

$$x = (x_3 - 3x_2 + 3x_1 - x_0)u^3 + (3x_2 - 6x_1 + 3x_0)u^2 + (3x_1 - 3x_0)u + x_0$$

The equation for y is identical with x_n replaced by y_n All points of the Bezier curve lie on (x, y) for variations of u in the interval $[0, 1]$. A usual approach for drawing Bezier curves in drawing applications is to choose the number of points to sample for the Bezier curve and connect these sampled points with lines. If the number of samples n is known in advance, a considerable amount of computation can be shifted to compile-time by precomputing the powers of u for the sample points $0, \frac{1}{n-1}, \frac{2}{n-1}, \ldots, \frac{n-2}{n-1}, 1$. The Liskell regression test suite contains two implementations of Bezier curve sampling, once as a generic function and once as macro embedding values of u into expanded code. Benchmarking these two implementations against each other shows 45% reduction of run-time for the macro implementation.

6.4 PTTs with side-effects

As PTTs run within the `TransformationMonad` that contains an `IO` monad as inner monad, PTTs can lift their operations into the `IO` monad. We show two applications of `IO` side effects, namely private storage and file access.

6.4.1 Private Storage: Interning

The environment transformer also runs within the `IO` monad. Whenever a PTT needs to maintain state, the programmer is able to generate new `IORef`s in the environment transformer and put the partially applied PTT into the transformation chain.

```
(defenv (lambda (LskEnv e p t d)
          (>>= newIORef
               (lambda (ref)
                 (return (LskEnv (ptt ref e)
                                 p t d))))))
```

A viable application for PTTs with side effects is an interning mechanism. Assume you want syntax sugar to rewrite `String` into a different string representation. GHC internally uses `FastString` to manage strings. Other applications might want to keep a list strings in C string representation.

An example interning mechanism for string could work as follows: Whenever `fs("test-string")` is seen by the expression transformer[10], it checks whether "test-string" is already part of its interning table. If it is not, it generates a symbol and puts that symbol associated with the string into its interning table. The interning table is extended with a tuple of (*new-sym*, `test-string`). The expression transformer then replaces the syntax sugar `fs("test-string")` either by the freshly generated symbol *new-sym* or by the symbol found in the interning table.

In cooperation with a declaration transformer, the interning mechanism can hold on to its promise and generate the symbols of its interning table as top-level binding at the end of the file. So, all interned strings that are identical point to the same symbol, and the object file potentially shrinks in size.

6.4.2 Using external resources

Parse tree transformers can consult external resources as they run within the `IO` monad.[11] An application of this might be stub code generation for remote procedure calls out of RPC definition files such as Corba's IDL or SUN's rpcgen files. Hibernate for Java uses XML files for object description and the Java code generator produces stub code according to these XML files. In Liskell, there is no need for an external preprocessing stage as with Hibernate, Corba or SUN RPC. An appropriate PTT can generate code at compile-time by reading the mapping definition file. For persistence mapping, a level of integration is feasible that might not be desired because it could lead to subtle bugs, namely generating data types and stub code by inspecting a running SQL data base.

An unusual, but truly powerful application of using external resources would be auto-embedding of foreign language files. A parse tree transformer could inject a transcription of C/Python/Perl code into Liskell code. We could also generate foreign function bindings from C header files.

[10]This is a singleton parse tree list containing the string "fs" as prefix. Pre- and postfix strings are omitted from the simplified parse tree `SParseTre` but contained in `ParseTree`.

[11]This is a powerful feature, but introduces an unexpected attack vector. Never compile untrusted Liskell code. This might not lead to new practical restrictions, as the sole purpose of code compilation is code execution, and we all agree that we do not execute untrusted code.

```
(defmacro-decl (def-memoized-function
                %(function expr))
  (let ((tabe (mapSymId (head function)
                        (lsect ++ "-table"))))
        (args (tail function)))
  `((define ,tabe
            (unsafePerformIO
              (>>= (HashTable.new == hasher)
                   newIORef))))
    (define ,function
      (unsafePerformIO
       (>>= (readIORef ,tabe)
            (lambda (ht)
              (case (lookup ht ([] ,@args))
                (Nothing
                 (let ((r (,expr ,@args)))
                       (>> (insert ht
                                   ([] ,@args)
                                   r)
                           (return r))))
                ((Just r) (return r))))))))))
```

Figure 7: def-memoized-function

6.5 Brevity: defmemoize

The attention span of humans is limited. The sensory registers can hold massive amount of data, but to make use of this information, the brain has to filter relevant parts of this sensory bombardment and capture it in short term memory. A bad useful/useless information ratio makes this filtering task harder and in the case of source code might prevent the reader from successfully focusing on the key parts.

Abstraction allows the programmer to think in terms of high level concepts, but we seldom find examples of abstractions that make the code longer. In the representation of high level combinations of concepts and components, brevity is needed to mentally handle programming by abstraction efficiently. We demonstrate how parse tree transformers help to unclutter the source code from low-level details, and once such a concept is understood, it can be signaled to programmer merely by a single keyword.

Assume the user of the Tic Tac Toe computer player function given in the previous section does not have any insight on how the computer player works. It might use heuristics, it might play randomly or it might compute the whole game tree to provide an optimal answer (which it happens to actually do). The game logic queries the computer player function after every move made by its opponent and the computer player recomputes its results. With computer players re-enumerating the game tree completely, this is inefficient as the answer must have been already computed in previous queries. By caching the results we can deliver the answer straight from the cache for subsequent queries.

In Lisp, there is a macro toolkit called `memoize` [5] that provides the programmer with a new function-defining top-level form `def-memoized-function`. It works exactly as the regular CL's `defun` with the exception that input-output pairs are cached, and whenever an already computed input is seen the cached result is returned.

We can easily write `def-memoized-function` as parse tree transformer in Liskell as shown in Figure 7. To work as a drop-in replacement for `define`, we allow ourselves to use `unsafePerformIO` to cache the results. This does not violate referential transparency. Instead of defining the function itself we define a wrapper around the function's body which is contained in `expr`. The wrapper performs a hash table lookup for the function's arguments contained in `args`. The first `define` form declares this hash table that goes by the name of the function suffixed with `"-table"`. We could also use the fresh name source `gensym` which is provided by `TransformationMonad`. In the event of an unsuccessful lookup, the wrapper adds the result `r` to the hash table before returning it. For the wrapper, we assume that `hasher` is a hash function according to the requirements of the `Data.HashTable` implementation.

After defining this macro, all we have to do is to replace

```
(define (computer-player state ego) ..)
```

with

```
(def-memoized-function (computer-player state ego)
                       ..)
```

Once the programmer has embraced the concept of memoization, he does not have to search for the valuable functions body which would be hidden in a deeply nested `let` of the wrapper. A single keyword efficiently communicates to the programmer the simple but powerful concept of memoization for functions. This successfully combines abstraction with brevity.

Other examples where brevity fosters productivity and hides distracting details are instance generation by the Haskell `deriving` syntax, field syntax, and Haskell Regular Patterns.

6.6 Embedded Languages: Prolog

Liskell can be used to embed any language that can be translated into typed lambda calculus. The Liskell regression test suite features a Prolog PTT that can be argue to be a Prolog compiler in its own right. It works by translating Prolog clauses into Liskell functions guided by the concepts introduced by Claessen and Ljunglöf in their paper on Prolog embedding in Haskell[6].

The Prolog file ends with the use of the `path` finding clause in a graph described by `edge`:

```
(<- (path X X (: X [])))
(<- (path X Z (: X Nodes))
    (edge X Y)
    (path Y Z Nodes))
```

in the inference statement

```
(define main
  (putStrLn (show (with-inference (path a e Xs)
                  xs))))
```

The Prolog embedding compiler can be divided into three parts:

- Helper functions such as Prolog primitives, a unification algorithm and the Prolog syntax and data ADTs,

- Meta-programming: a parse tree transformer collecting all toplevel Prolog statement and turning them into functions. This process is modelled according to [6]. `with-inference` provides a convience macro to access Prolog inference.
- Source code defining Prolog predicates and using Prolog inference.

7. CONCLUSIONS AND FUTURE WORK

This paper demonstrates the successful application of meta-programming paradigms to a pure functional programming language by employing symbolic expressions as syntax form.

GHC. Most of future work consists of pragmatic improvements of GHC for Liskell programming. Error reporting has to be improved to aid regular programming as well as for meta-programming. One aspect of the Lisp family is that most of the implementations contain an interactive environment known as REPL – short for **r**ead-**e**val-**p**rint **l**oop. GHC includes GHC interactive, or short GHCi. When judged by the features of a regular REPL, GHCi seems inferior.

Liskell design. GHC contains a numerous of extensions that are not yet accessible by Liskell. It has to be completed in the following areas: foreign function interface declarations, Haddock documentation, tuple unboxing, GADT declarations, pattern guards, functional dependencies in type classes, existential quantification, rewrite rules, bang patterns, pragmas.

There are two obvious short comings in the Liskell design when it comes to meta-programming: `defmodule` is not subject to parse tree transformation. A possible solution is to introduce a malleable module information that can be changed throughout the compilation process. Further, dispatchers are not imported into the compiler-environment automatically. Adding all the dispatchers provided by Liskell Prelude is cumbersome and mostly template code. An elegant solution for these two problems still have to be found.

Development Tools. SLIME[1] provides an excellent CL development environment for Emacs. A current effort by Benedikt Schmidt is extending SLIME in his Shim project[12] to provide a better development environment for Haskell. As SLIME already understands symbolic expression syntax very well, and Shim closes the gap to GHC-API, there seems to be a good foundation to build a Liskell development environment.

7.1 Meta-programming

Denotational Semantics. Languages given with their denotational semantics can be translated quite easily. Scheme in Liskell should be ease to realize.

More general, we should be able find a uniform description language for denotational semantics. Denotation semantics can be seen as a set of transformation rules how to transform a source language into lambda calculus. We can create transformers that turn the denotation description language into a series of parse tree transformers that transform the source language into Haskell as Haskell is very close to the lambda calculus. At the end, we could capture the semantics of the denotational description language in denotational semantics too and create meta-circular transformer that takes itself as input and gives itself as output. But we leave this as an exercise to the hedonistic reader.

Dependent Types in Liskell. Defining translation rules for expressions is not new, denotation semantics publications exist since 30 years. A rather new area of research is simulating richer type systems on traditional type systems. The papers by McBride [11] as well as Apple and Weimer [3] lay out strategies on how to simulate a dependent type system with the Haskell type system. Applying meta-programming techniques to extend the type system of a language is a rather new approach and it might push the limits of meta-programming as research tool further.

8. REFERENCES

[1] SLIME: *The Superior Lisp Interaction Mode for Emacs.* `http://common-lisp.net/project/slime/`.

[2] American National Standards Institute and Information Technology Industry Council. *Programming language – Common LISP*, 1996.

[3] J. Apple and W. Weimer. *Simulating Dependent Types with Guarded Algebraic Datatypes.* `http://www.cs.virginia.edu/~jba5b/singleton/`.

[4] A. Bawden. *Quasiquotation in Lisp.* `http://citeseer.ist.psu.edu/bawden99quasiquotation.html`, 1999.

[5] T. Bradshaw. *Memoize Library.* `http://www.cliki.net/memoize`.

[6] K. Claessen and P. Ljungl. *Typed logical variables in Haskell.* `http://citeseer.ist.psu.edu/claessen00typed.html`, 2000.

[7] P. Graham. *On Lisp: advanced techniques for Common Lisp.* Prentice-Hall, `http://www.paulgraham.com/onlisp.html`, 1994.

[8] S. P. Jones. *Haskell 98: The Revised Report.* `http://research.microsoft.com/~simonpj/haskell98-revised/`, Jan 2003.

[9] S. Marlow. *Happy, the parser generator for Haskell.* `http://www.haskell.org/happy/`.

[10] S. Marlow. *Scanner Generator Alex.* `http://www.haskell.org/alex/`.

[11] C. McBride. *Faking it—simulating dependent types in Haskell.* `http://citeseer.ist.psu.edu/mcbride01faking.html`, 2001.

[12] B. Schmidt. *Shim.* `http://shim.haskellco.de/trac/shim`.

[13] T. Sheard and S. P. Jones. *Template metaprogramming for Haskell.* `http://research.microsoft.com/Users/simonpj/papers/meta-haskell/meta-haskell.ps`, 2002.

[14] Haskell Wiki. *Books and Tutorials on Haskell.* `http://haskell.org/haskellwiki/Books_and_tutorials`.