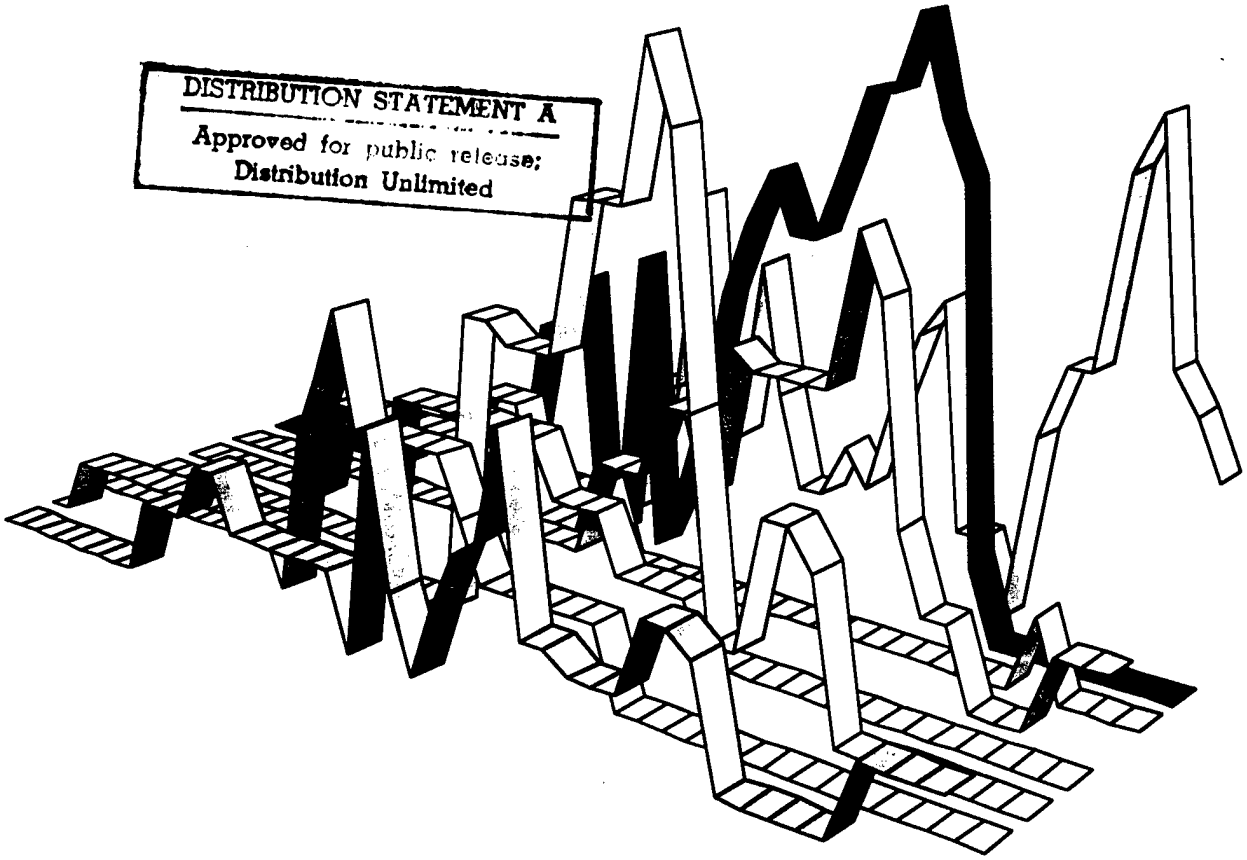
PACIFIC SOFTWARE RESEARCH CENTER
TECHNICAL REPORT

Initial Suite of Small Language Definitions and Implementations for DSDL

Pacific Software Research Center
April 22, 1998

CONTRACT NO. F19628-96-C-0161
CDRL SEQUENCE NO. [CDRL 0002.3]

Prepared for:
USAF



OREGON GRADUATE INSTITUTE OF SCIENCE & TECHNOLOGY

Initial Suite of Small Language Definitions and Implementations for DSDL

Pacific Software Research Center
April 22, 1998

CONTRACT NO. F19628-96-C-0161
CDRL SEQUENCE NO. [CDRL 0002.3]

Prepared for:
USAF
Electronic Systems Center/AVK

Prepared for:
Pacific Software Research Center
Oregon Graduate Institute of Science and Technology
PO Box 91000
Portland, OR 97291

Initial Suite of Seven Embedded Languages

Jeffrey R. Lewis and John Launchbury

April 22, 1998

Abstract

Domain-specific languages are small, special purpose languages created to describe computational solutions in a particular problem domain. Domain-specific languages have proven themselves useful many times over; however, the cost of defining and implementing a domain specific language can be high. An approach that avoids the overhead of domain-specific language definition is to define an *embedded* language—i.e. a collection of definitions in a sufficiently expressive host language. Embedding a domain-specific language places high demands on a host language. The host languge must be able to express the essence of the domain, while not sacrificing too much in syntax. This report, presents a suite of seven examples of embeddings using the functional programming language Haskell.

Contents

1 Introduction

Domain-specific languages are small, special purpose languages created to describe computational solutions in a particular problem domain. They vary greatly in presentation, level of abstraction and intended audience. They typically incorporate the fundamental abstractions of the problem domain directly, without requiring that concepts be encoded. For example, in the widely used parser generator yacc, grammar productions can be expressed directly. Contrast this with the typical hand-written parser where a programmer had to translate the grammar productions into a program in some general purpose language. The most effective DSLs use notations that are closely related to the notations already used by experts to communicate. In yacc, for example, ambiguous grammars can be annotated with precedence declarations to resolve ambiguity, reflecting standard practice in grammar specification.

An *embedded* language is a DSL that is implemented as a collection of definitions in a host language. Many DSLs are implemented as stand-alone languages, with their own interpreters, translators and/or compilers. However, the development of such tools may require a lot of effort and careful design. An embedded language offers an alternative approach that leverages off of existing tools and language designs. However, for an embedded language to work well, the host language must have sufficient expressive power such that the embedding is natural and doesn't distract the user with details irrelevant to the domain.

We can view the role of an embedded language as lying along the spectrum from recognition of the particular problem domain to canonization as a separate domain specific language. This final step, where the problem domain is captured as a stand-alone DSL, is the final mature stage in the life of a problem domain. The role of embedded languages plays a part in the middle of the life cycle, where details and properties of the problem space are still being explored. At this stage, the designer doesn't want to be bogged down by the difficult issues of language design; rather, he wants to leverage off the framework provided by the existing language to quickly test ideas and construct solutions that are both extensible and modifiable.

1.1 Embedded Languages and Stand-alone Languages

One of the interesting aspects of embedded languages is that they don't have their own syntax. Their syntax is entirely borrowed from the host language. In fact, to a certain degree, it is hard to distinguish the embedded language from the host language. Or rather, it's hard to say where the embedded language stops and the host language picks up. The lack of a fixed boundary defining the language gives us great flexibility. It is easy to add new "features" to the embedded language just by defining new types or operations.

The embedded language approach is often contrasted with the more traditional approach of defining a stand-alone language with its own syntax and semantics.

Advantages of embedded languages:

- greater flexibility
- language-level re-use (leverage off of the features of the host language)
- it is often desirable to have the ability to step outside of a domain specific language to the more general features of a general purpose language. Many domain specific languages are incrementally extended to general purpose languages, often with bad results (e.g. Tcl, Perl). The embedded language approach avoids this pitfall by starting with a well-designed general purpose language.
- No need to develop new logic for reasoning about domain specific programs. You borrow the well-developed logic of the host language.
- Programs can be optimized using existing facilities already developed for the host language. Partial evaluation would be particularly useful here.

Advantages of stand-alone languages:

- Direct expression of domain concepts—not encoded in syntax of general-purpose language
- complete control over interpretation of syntax. For example, in the parsing domain, we are typically limited to recursive descent parsers, forcing the programmer to write the grammar in such a manner as to avoid left recursion. Yacc is able to translate its input into a bottom-up parser, allowing the programmer to write grammars in a much more natural style.
- Programming errors can be interpreted with respect to the syntax and semantics of the language and their reporting tailored to language. For an embedded language, errors are reported by the host language, and may require expert knowledge to resolve.
- Users of a domain specific language may find the generality of a general purpose language to hinder more than help.
- A domain specific language can have a more refined type system than a general purpose host language.
- Arguing about the correctness of an embedded language can be difficult, because it's hard to say what the language is. Any reasoning must be done in the logic of the host language, without being able to take advantage of the narrowing afforded by a specific domain.

Of course, there's nothing to say that a mixture of embedded and stand-alone languages can't be employed in the same system. The Isabelle theorem prover, for example, expresses tactics as an embedded language in SML, while it expresses theories in a domain specific language interpreted by Isabelle.

2 The Suite of Embedded languages

We introduce the fundamental principles of the seven embedded languages. Full details may be found in the attached papers.

2.1 Parsing

In functional programming, a popular approach to building recursive descent parsers is to model them as functions, and to define higher-order functions (or combinators) that implement grammar constructions such as productions, and alternatives. The parser combinators of Hutton and Meijer is a library of functions for constructing parsers [Volume I].

The key abstraction of the parser combinators is that of an annotated BNF. Productions are annotated with values—when the given production is recognized, the result of parsing is the indicated value. However, parser combinators are much more flexible because we are not restricted to the traditional rigid model of productions, terminals and non-terminals. With the combinators, we construct values of type `Parser a`, where `a` is the type of the result. For example, a parser that parsed integer literals would be a value of type `Parser Int`. We can combine parsers in all the ways available to us in the language: they may be stored, passed as an argument, or returned as a result. This simplified view of parsers also makes unnecessary the traditional distinction between scanners and parser. We can combine the two in a single specification quite naturally. Regardless, BNF-style grammars are straightforward to express using the parser combinators and we will continue using the terminology of BNF as a convenient way of talking about them.

A final feature of note regarding the parser combinators is that they aren't restricted to a particular class of grammars, such as LALR(1) for yacc. An ambiguous grammar is implemented as a parser that yields multiple results. Laziness allows us to use the generality of non-deterministic parsing without necessarily having to explore all possible parses.

Productions are built using monadic `do` notation so that it's easy to bind a name to any part of a production. The following example parses an `if` statement, and constructs the appropriate abstract syntax out of the relevant parts of the `if` using `buildIf`.

```
pIf = do  symbol "if"
          b <- expr
          symbol "then"
          x <- expr
          symbol "else"
          y <- expr
          return (buildIf b x y)
```

By using the `do` notation, the programmer is able to name the results, `x` and `y` in this case, that will be used later, and easily discard the results of parses that don't return values of interest, such as `if`, `then` and `else`.

Alternative parses are indicated by the ++ combinator. For example, if we had another parser pCase that parsed case statements, we could form a parser that recognizes either if or case by joining the two as follows: pIf ++ pCase.

2.2 Pretty Printing

Pretty printing is another example of a domain long known to be well-suited to embedding. The pretty printing combinators of Hughes is a library of functions for constructing pretty printers [Volume II].

The key abstraction is that of an "intelligent document"—a document which can have a variety of textual layouts depending on its context (position on page, line width, ribbon width, etc). The combinators build intelligent documents that can be layed out horizontally, vertically, or selectively based on what fits on the line. In addition, there's also a combinator for indicating indentation in vertical layouts.

The pretty printing combinators can be summarized as follows:

- `text` construct a document consisting of a literal string
- `<>` put two documents next to each other
- `$$` put two documents over each other
- `sep` takes a list of documents and creates a single document, separated either by <> or $$, depending on which gives the best layout.
- `nest` increases the indentation of a document that's layed out vertically

context.

For example, we can layout the if statement parsed by the previous example:

```
ppIf (If (b, x, y)) = sep [text "if " <> ppExpr b,
                           nest 2 (text "then " <> ppExpr y),
                           nest 2 (text "else " <> ppExpr z)]
```

This can produce either of the following two layouts, depending on which one fits best.

```
if x then y else z
```

```
if x
 then y
 else z
```

2.3 Hawk

Hawk is a library for building executable specifications of microprocessors that concentrates on the level of micro-architecture. Hawk has been used to specify modern microarchitectures similar to the Pentium Pro with features such as super-scalar execution, out-of-order execution and register renaming [Volume III].

The key abstraction in Hawk is signals—discrete time-dependent values. Computer signals are typically binary (`Signal Bool`), although we can, for example, model a bundle of binary signals more conveniently as a (fixed-precision) integer (`Signal Int`), or model a control signal as an enumerated datatype (`Signal Reg`). Processing units are expressed as functions on signals. Circuits are formed out of mutually recursive definitions of processing units. For example, the following is the top-level definition of an simple processor

```
data Reg = R0 | R1 | R2 | R3
data OpCode = ADD | SUB | INC

simple :: (Signal OpCode, Signal Reg,
           Signal Reg, Signal Reg) ->
          (Signal Reg, Signal Int)
simple (opcode, destReg, srcRegA, srcRegB) =
       (destReg', aluOutput')
  where
    (aluInputA, aluInputB) = regFile (destReg', aluOutput')
                                     (srcRegA, srcRegB)
    aluOutput = alu opcode aluInputA aluInputB
    aluOutput' = delay 0 aluOutput
    destReg' = delay R0 destReg
```

2.4 Fran

Fran (Function Reactive Animation) is a library for composing interactive multimedia animations [Volume IV].

The two key abstractions in Fran are behaviors and events. A behavior is a value which varies over time. It could be a time-dependent number, such as a sine wave, or even a time-dependent image, such as an animation. An event is a value at a particular time, such as a button-press. Behaviors are used to describe animations, and events are used to describe how those animations react to events in the outside world.

The basic combinator for describing (2-D) animations is `moveXY`. For example, the following describes an image, called `charlotte`, that moves smoothly back and forth.

```
leftRightCharlotte = moveXY wiggle 0 charlotte
charlotte = importBitmap "charlotte.bmp"
```

`wiggle` is a real-valued behavior that oscillates between negative one and positive one and describes the desired behavior along the X axis. Animations can be combined using combinators such as `over`, which lays one animation on top of another. For example, we could define another animation that moves up and down.

```
upDownPat = moveXY 0 waggle pat
```

And then we could combine the two into a kind of dance:

```
charlottePatDance = leftRightCharlotte `over` upDownPat
```

The basic combinator for describing the interaction of behaviors with events is untilB. untilB pastes together two animations in time with the transition from one to the other indicated by an event. For example, the following indicates a red color attribute that will transition to a blue color attribute when the left mouse button is pressed.

```
red `untilB` (lbp -=> blue)
```

2.5 Haskore

Haskore is a collection of Haskell modules designed for expressing musical structures in the high-level, declarative style of functional programming [Volume V]. In Haskore, musical objects consist of primitive notions such as notes and rests, operations to transform musical objects such as transpose and tempo-scaling, and operations to combine musical objects to form more complex ones, such as concurrent and sequential composition.

The key abstractions in Haskore are those of a *score*, i.e. notated music, and a *player*, i.e. an interpretation of the score. The notion of a player is interpreted broadly, including both performers (like a MIDI-capable sound card) and the layout and printing of sheet music.

The basic building block in Haskore is a Note, which specifies pitch, duration and any attributes, such as accents, or grace notes. Melodies are composed by stringing together notes sequentially using :+:. Harmonies are composed by combining melodies in parallel using :=:.

For example, here's the first line of "Row row row your boat."

```
row1 =      c 5 qn []
       :+: c 5 qn []
       :+: c 5 (trn * 2) [] :+: d 5 trn []
       :+: e 5 qn []
```

Each note on the scale has an associated function of the same name. In the example above, we use the notes c, d and e. Each note takes three arguments: the first indicates which octave the note lies in, the second the duration of the note, and the third, a list of any attributes the note has (like dynamic marking or grace notes—this example contains no attributes). The durations used are qn for quarter note, trn for eighth-note triplet, and shortly we'll use hn for half note.

It's easy to make a round. If the whole melody is called row, the following function will form a round with n singers, each starting a half note after the last.

```
rown 0 = Rest 0
rown n = rown (n - 1) :=: delay ((n - 1) * hn) row
```

2.6 CGI

HTML, the language in which web pages are written, is a static layout language. However, many web pages need to respond to the actions of remote users, such as taking information for credit card orders. This interaction is done by the use of CGI scripts—programs running on the server that generate HTML in response to data sent by the remote browser. The CGI library provides combinators for generating HTML and a nicer interface to the awkward argument passing convention of CGI scripts [Volume VI].

The underlying abstraction is that of hypertext—text annotated with mark-up indicating the document structure, as well as the relationship with other documents (hyper-links). The abstraction for CGI scripts is essentially program generation in the simplified case where the programs are HTML.

The following is a script that gives positive reinforcement for choosing a radio button labelled `Haskell`, instead of one labelled `Java`.

```
script env =
    case lookup "language" env of
    Nothing -> page "Language Choice" [] [choice]
    Just "Haskell" ->
        page "Chose Haskell" [] [h1 ("You chose well!")]
    Just "Java" ->
        page "Chose Java" [] [h1 ("Well, If you insist.")]

choice =
    gui "choice.cgi" [ buttons,
                       submit "" "Submit",
                       reset "" "Reset" ]

buttons = (radio `group` "language")  ["Haskell", "Java"]
```

2.7 AgentScript

AgentScript is a library of functions for controlling MicroSoft agents, animated entities that interact with the user by gestures and synthesized speech [Volume VII]. Scripts for the agents are put together with parallel and sequential combinators that make specification of interactions with multiple agents particularly easy. This is because synchronization is taken care of by AgentScript, unlike how it is done in C++ or Visual Basic, where the action of each agent must be carefully synchronized by hand with the other agents—a fairly error-prone process.

The basic combinators are `<*>` for sequential composition (do one thing after another), and `<|>` for parallel composition (do them simultaneously). For example, the following is the top-level code for a demo involving three interacting agents. `seqAnim` composes the elements of the list using sequential composition.

```
demo erik simon daan =
    seqAnim [ erik  introduces,
              simon helps,
              daan  showUpsUp <|> simon looksAround
                              <|> erik moves,
              (erik wantsCompiler <*> daan hasDoneIt)
                  <|> simon looksAtRod,
              simon isPleased,
              erik  wantsAnimation,
              daan  explains,
              simon wantsReport,
              daan  isSurprised,
              daan writes  'while'
                  (erik searches  'while'
                      (simon goesHome)
                  ),
              daan looksGood <|> erik wavesGoodbye,
              erik goesSurfing,
              daan endsTheShow
            ]
```

3 Implementation Needs

The language Haskell offers a number of powerful features that aid in embedding languages. Here is a list of those that are exploiting by the examples in the previous section.

Higher-order functions give us the ability to abstract over common patterns of control, which allows us to hide unnecessary details and cleanly construct new components from old ones.

Polymorphism gives us the ability to abstract over common patterns of data.

Lazy evaluation allows us to abstract away from evaluation order. Part of what makes popular languages a poor choice for embedding languages is that they enforce a strict evaluation order that may not be appropriate for a given problem domain. This is most often the case for domains that deal with conceptually infinite concepts such as streams.

Type inference is another aid in reducing levels of detail, reducing the clutter by avoiding unnecessary declarations, and at the same time assuring that terms are used consistent to their interfaces.

Type classes in Haskell support a structured method for defining overloaded operators. They give two main benefits. First, overloading is natural, and is a notational occurance in many problem domains. Supporting it helps make embedding more natural. Further, type inference can be used to identify misuse of overloading. Second, it reduces clutter by providing a way of hiding and propagating implicit parameters.

Monad support aids in explicit management of computation. Most languages have fixed notions of computation, such as evaluation order, how exceptions are handled, and how I/O is done. How these features interact is also hard-coded into the language. It is awkward or impossible to escape the underlying model of computation. A poor match between the builtin computational model of language and the computational model of the problem domain spells trouble for an embedded language. A currently popular technique for managing computational details in functional programming languages is to use a monad, a structure borrowed from mathematics which encapsulates notions of computation. Successful use of monads, however, requires a fairly sophisticated language. Monadic programming is Haskell is supported by all of the features so far listed, plus a language construct, the `do` notation, for writing programs using monads.

Garbage collection aids in abstracting away from details of resource allocation. These details are a major distraction to C and C++ programmers, and a major source of errors.

Fixity control aids in providing syntax that looks more natural and avoids unnecessary use of parentheses.

Monadic Parser Combinators

Graham Hutton
University of Nottingham

Erik Meijer
University of Utrecht

Appears as technical report NOTTCS-TR-96-4,
Department of Computer Science, University of Nottingham, 1996

Abstract

In functional programming, a popular approach to building recursive descent *parsers* is to model parsers as functions, and to define higher-order functions (or *combinators*) that implement grammar constructions such as sequencing, choice, and repetition. Such parsers form an instance of a *monad*, an algebraic structure from mathematics that has proved useful for addressing a number of computational problems. The purpose of this article is to provide a step-by-step tutorial on the monadic approach to building functional parsers, and to explain some of the benefits that result from exploiting monads. No prior knowledge of parser combinators or of monads is assumed. Indeed, this article can also be viewed as a first introduction to the use of monads in programming.

Contents

1 Introduction

In functional programming, a popular approach to building recursive descent *parsers* is to model parsers as functions, and to define higher-order functions (or *combinators*) that implement grammar constructions such as sequencing, choice, and repetition. The basic idea dates back to at least Burge's book on recursive programming techniques (Burge, 1975), and has been popularised in functional programming by Wadler (1985), Hutton (1992), Fokker (1995), and others. Combinators provide a quick and easy method of building functional parsers. Moreover, the method has the advantage over functional parser generators such as Ratatosk (Mogensen, 1993) and Happy (Gill & Marlow, 1995) that one has the full power of a functional language available to define new combinators for special applications (Landin, 1966).

It was realised early on (Wadler, 1990) that parsers form an instance of a *monad*, an algebraic structure from mathematics that has proved useful for addressing a number of computational problems (Moggi, 1989; Wadler, 1990; Wadler, 1992a; Wadler, 1992b). As well as being interesting from a mathematical point of view, recognising the monadic nature of parsers also brings practical benefits. For example, using a monadic sequencing combinator for parsers avoids the messy manipulation of nested tuples of results present in earlier work. Moreover, using *monad comprehension* notation makes parsers more compact and easier to read.

Taking the monadic approach further, the monad of parsers can be expressed in a modular way in terms of two simpler monads. The immediate benefit is that the basic parser combinators no longer need to be defined explicitly. Rather, they arise automatically as a special case of lifting monad operations from a base monad m to a certain other monad parameterised over m. This also means that, if we change the nature of parsers by modifying the base monad (for example, limiting parsers to producing at most one result), then new combinators for the modified monad of parsers also arise automatically via the lifting construction.

The purpose of this article is to provide a step-by-step tutorial on the monadic approach to building functional parsers, and to explain some of the benefits that result from exploiting monads. Much of the material is already known. Our contributions are the organisation of the material into a tutorial article; the introduction of new combinators for handling lexical issues without a separate lexer; and a new approach to implementing the offside rule, inspired by the use of monads.

Some prior exposure to functional programming would be helpful in reading this article, but special features of Gofer (Jones, 1995b) — our implementation language — are explained as they are used. Any other lazy functional language that supports (multi-parameter) constructor classes and the use of monad comprehension notation would do equally well. No prior knowledge of parser combinators or monads is assumed. Indeed, this article can also be viewed as a first introduction to the use of monads in programming. A library of monadic parser combinators taken from this article is available from the authors, via the World-Wide-Web.

2 Combinator parsers

We begin by reviewing the basic ideas of combinator parsing (Wadler, 1985; Hutton, 1992; Fokker, 1995). In particular, we define a type for parsers, three primitive parsers, and two primitive combinators for building larger parsers.

2.1 The type of parsers

Let us start by thinking of a parser as a function that takes a string of characters as input and yields some kind of tree as result, with the intention that the tree makes explicit the grammatical structure of the string:

```
type Parser = String -> Tree
```

In general, however, a parser might not consume all of its input string, so rather than the result of a parser being just a tree, we also return the unconsumed suffix of the input string. Thus we modify our type of parsers as follows:

```
type Parser = String -> (Tree,String)
```

Similarly, a parser might fail on its input string. Rather than just reporting a run-time error if this happens, we choose to have parsers return a list of pairs rather than a single pair, with the convention that the empty list denotes failure of a parser, and a singleton list denotes success:

```
type Parser = String -> [(Tree,String)]
```

Having an explicit representation of failure and returning the unconsumed part of the input string makes it possible to define combinators for building up parsers piecewise from smaller parsers. Returning a list of results opens up the possibility of returning more than one result if the input string can be parsed in more than one way, which may be the case if the underlying grammar is ambiguous.

Finally, different parsers will likely return different kinds of trees, so it is useful to abstract on the specific type `Tree` of trees, and make the type of result values into a parameter of the `Parser` type:

```
type Parser a = String -> [(a,String)]
```

This is the type of parsers we will use in the remainder of this article. One could go further (as in (Hutton, 1992), for example) and abstract upon the type `String` of tokens, but we do not have need for this generalisation here.

2.2 Primitive parsers

The three primitive parsers defined in this section are the building blocks of combinator parsing. The first parser is `result v`, which succeeds without consuming any of the input string, and returns the single result `v`:

```
result  :: a -> Parser a
result v = \inp -> [(v,inp)]
```

An expression of the form `\x -> e` is called a λ-abstraction, and denotes the function that takes an argument `x` and returns the value of the expression `e`. Thus `result v` is the function that takes an input string `inp` and returns the singleton list `[(v,inp)]`. This function could equally well be defined by `result v inp = [(v,inp)]`, but we prefer the above definition (in which the argument `inp` is shunted to the body of the definition) because it corresponds more closely to the type `result :: a -> Parser a`, which asserts that `result` is a function that takes a single argument and returns a parser.

Dually, the parser `zero` always fails, regardless of the input string:

```
zero :: Parser a
zero  = \inp -> []
```

Our final primitive is `item`, which successfully consumes the first character if the input string is non-empty, and fails otherwise:

```
item :: Parser Char
item  = \inp -> case inp of
                  []     -> []
                  (x:xs) -> [(x,xs)]
```

2.3 Parser combinators

The primitive parsers defined above are not very useful in themselves. In this section we consider how they can be glued together to form more useful parsers. We take our lead from the BNF notation for specifying grammars, in which larger grammars are built up piecewise from smaller grammars using a *sequencing* operator — denoted by juxtaposition — and a *choice* operator — denoted by a vertical bar |. We define corresponding operators for combining parsers, such that the structure of our parsers closely follows the structure of the underlying grammars.

In earlier (non-monadic) accounts of combinator parsing (Wadler, 1985; Hutton, 1992; Fokker, 1995), sequencing of parsers was usually captured by a combinator

```
seq      :: Parser a -> Parser b -> Parser (a,b)
p `seq` q = \inp -> [((v,w),inp'') | (v,inp')  <- p inp
                                   , (w,inp'') <- q inp']
```

that applies one parser after another, with the results from the two parsers being combined as pairs. The infix notation `p `seq` q` is syntactic sugar for `seq p q`; any function of two arguments can used as an infix operator in this way, by enclosing its name in backquotes. At first sight, the `seq` combinator might seem a natural composition primitive. In practice, however, using `seq` leads to parsers with nested tuples as results, which are messy to manipulate.

The problem of nested tuples can be avoided by adopting a *monadic* sequencing combinator (commonly known as `bind`) which integrates the sequencing of parsers with the processing of their result values:

```
bind      :: Parser a -> (a -> Parser b) -> Parser b
p `bind` f = \inp -> concat [f v inp' | (v,inp') <- p inp]
```

The definition for `bind` can be interpreted as follows. First of all, the parser `p` is applied to the input string, yielding a list of (value,string) pairs. Now since `f` is a function that takes a value and returns a parser, it can be applied to each value (and unconsumed input string) in turn. This results in a list of lists of (value,string) pairs, that can then be flattened to a single list using `concat`.

The `bind` combinator avoids the problem of nested tuples of results because the results of the first parser are made directly available for processing by the second, rather than being paired up with the other results to be processed later on. A typical parser built using `bind` has the following structure

```
p1 `bind` \x1 ->
p2 `bind` \x2 ->
...
pn `bind` \xn ->
result (f x1 x2 ... xn)
```

and can be read operationally as follows: apply parser `p1` and call its result value `x1`; then apply parser `p2` and call its result value `x2`; ...; then apply the parser `pn` and call its result value `xn`; and finally, combine all the results into a single value by applying the function `f`. For example, the `seq` combinator can be defined by

```
p `seq` q = p `bind` \x ->
            q `bind` \y ->
            result (x,y)
```

(On the other hand, `bind` cannot be defined in terms of `seq`.)

Using the `bind` combinator, we are now able to define some simple but useful parsers. Recall that the `item` parser consumes a single character unconditionally. In practice, we are normally only interested in consuming certain specific characters. For this reason, we use `item` to define a combinator `sat` that takes a predicate (a Boolean valued function), and yields a parser that consumes a single character if it satisfies the predicate, and fails otherwise:

```
sat  :: (Char -> Bool) -> Parser Char
sat p = item `bind` \x ->
        if p x then result x else zero
```

Note that if `item` fails (that is, if the input string is empty), then so does `sat p`, since it can readily be observed that `zero `bind` f = zero` for all functions `f` of the appropriate type. Indeed, this equation is not specific to parsers: it holds for an arbitrary *monad with a zero* (Wadler, 1992a; Wadler, 1992b). Monads and their connection to parsers will be discussed in the next section.

Using `sat`, we can define parsers for specific characters, single digits, lower-case letters, and upper-case letters:

```
char  :: Char -> Parser Char
char x = sat (\y -> x == y)
```

```
digit :: Parser Char
digit  = sat (\x -> '0' <= x && x <= '9')

lower :: Parser Char
lower  = sat (\x -> 'a' <= x && x <= 'z')

upper :: Parser Char
upper  = sat (\x -> 'A' <= x && x <= 'Z')
```

For example, applying the parser `upper` to the input string `"Hello"` succeeds with the single successful result `[('H',"ello")]`, since the `upper` parser succeeds with `'H'` as the result value and `"ello"` as the unconsumed suffix of the input. On the other hand, applying the parser `lower` to the string `"Hello"` fails with `[]` as the result, since `'H'` is not a lower-case letter.

As another example of using `bind`, consider the parser that accepts two lower-case letters in sequence, returning a string of length two:

```
lower `bind` \x ->
lower `bind` \y ->
result [x,y]
```

Applying this parser to the string `"abcd"` succeeds with the result `[("ab","cd")]`. Applying the same parser to `"aBcd"` fails with the result `[]`, because even though the initial letter `'a'` can be consumed by the first `lower` parser, the following letter `'B'` cannot be consumed by the second `lower` parser.

Of course, the above parser for two letters in sequence can be generalised to a parser for arbitrary strings of lower-case letters. Since the length of the string to be parsed cannot be predicted in advance, such a parser will naturally be defined recursively, using a choice operator to decide between parsing a single letter and recursing, or parsing nothing further and terminating. A suitable choice combinator for parsers, `plus`, is defined as follows:

```
plus       :: Parser a -> Parser a -> Parser a
p `plus` q  = \inp -> (p inp ++ q inp)
```

That is, both argument parsers `p` and `q` are applied to the same input string, and their result lists are concatenated to form a single result list. Note that it is not required that `p` and `q` accept disjoint sets of strings: if both parsers succeed on the input string then more than one result value will be returned, reflecting the different ways that the input string can be parsed.

As examples of using `plus`, some of our earlier parsers can now be combined to give parsers for letters and alpha-numeric characters:

```
letter   :: Parser Char
letter    = lower `plus` upper

alphanum :: Parser Char
alphanum  = letter `plus` digit
```

More interestingly, a parser for words (strings of letters) is defined by

```
word :: Parser String
word  = neWord `plus` result ""
        where
           neWord = letter `bind` \x  ->
                    word   `bind` \xs ->
                    result (x:xs)
```

That is, `word` either parses a non-empty word (a single letter followed by a word, using a recursive call to `word`), in which case the two results are combined to form a string, or parses nothing and returns the empty string.

For example, applying `word` to the input `"Yes!"` gives the result `[("Yes","!"), ("Ye","s!"), ("Y","es!"), ("","Yes!")]`. The first result, `("Yes","!")`, is the expected result: the string of letters `"Yes"` has been consumed, and the unconsumed input is `"!"`. In the subsequent results a decreasing number of letters are consumed. This behaviour arises because the choice operator `plus` is *non-deterministic*: both alternatives can be explored, even if the first alternative is successful. Thus, at each application of `letter`, there is always the option to just finish parsing, even if there are still letters left to be consumed from the start of the input.

3 Parsers and monads

Later on we will define a number of useful parser combinators in terms of the primitive parsers and combinators just defined. But first we turn our attention to the monadic nature of combinator parsers.

3.1 The parser monad

So far, we have defined (among others) the following two operations on parsers:

```
result :: a -> Parser a
bind   :: Parser a -> (a -> Parser b) -> Parser b
```

Generalising from the specific case of `Parser` to some arbitrary type constructor `M` gives the notion of a monad: a *monad* is a type constructor `M` (a function from types to types), together with operations `result` and `bind` of the following types:

```
result :: a -> M a
bind   :: M a -> (a -> M b) -> M b
```

Thus, parsers form a monad for which `M` is the `Parser` type constructor, and `result` and `bind` are defined as previously. Technically, the two operations of a monad must also satisfy a few algebraic properties, but we do not concern ourselves with such properties here; see (Wadler, 1992a; Wadler, 1992b) for more details.

Readers familiar with the categorical definition of a monad may have expected two operations `map :: (a -> b) -> (M a -> M b)` and `join :: M (M a) -> M a` in place of the single operation `bind`. However, our definition is equivalent to the

categorical one (Wadler, 1992a; Wadler, 1992b), and has the advantage that `bind` generally proves more convenient for monadic programming than `map` and `join`.

Parsers are not the only example of a monad. Indeed, we will see later on how the parser monad can be re-formulated in terms of two simpler monads. This raises the question of what to do about the naming of the monadic combinators `result` and `bind`. In functional languages based upon the Hindley-Milner typing system (for example, Miranda[†] and Standard ML) it is not possible to use the same names for the combinators of different monads. Rather, one would have to use different names, such as `resultM` and `bindM`, for the combinators of each monad `M`.

Gofer, however, extends the Hindley-Milner typing system with an overloading mechanism that permits the use of the same names for the combinators of different monads. Under this overloading mechanism, the appropriate monad for each use of a name is calculated automatically during type inference.

Overloading in Gofer is accomplished by the use of *classes* (Jones, 1995c). A class for monads can be declared in Gofer by:

```
class Monad m where
   result :: a -> m a
   bind   :: m a -> (a -> m b) -> m b
```

This declaration can be read as follows: a type constructor `m` is a member of the class `Monad` if it is equipped with `result` and `bind` operations of the specified types. The fact that `m` must be a type constructor (rather than just a type) is inferred from its use in the types for the operations.

Now the type constructor `Parser` can be made into an instance of the class `Monad` using the `result` and `bind` from the previous section:

```
instance Monad Parser where
   -- result :: a -> Parser a
   result v   = \inp -> [(v,inp)]

   -- bind   :: Parser a -> (a -> Parser b) -> Parser b
   p `bind` f = \inp -> concat [f v out | (v,out) <- p inp]
```

We pause briefly here to address a couple of technical points concerning Gofer. First of all, type synonyms such as `Parser` must be supplied with all their arguments. Hence the instance declaration above is not actually valid Gofer code, since `Parser` is used in the first line without an argument. The problem is easy to solve (redefine `Parser` using `data` rather than `type`, or as a *restricted* type synonym), but for simplicity we prefer in this article just to assume that type synonyms *can* be partially applied. The second point is that the syntax of Gofer does not currently allow the types of the defined functions in instance declarations to be explicitly specified. But for clarity, as above, we include such types in comments.

Let us turn now to the following operations on parsers:

[†] Miranda is a trademark of Research Software Ltd.

```
zero :: Parser a
plus :: Parser a -> Parser a -> Parser a
```

Generalising once again from the specific case of the **Parser** type constructor, we arrive at the notion of a *monad with a zero and a plus*, which can be encapsulated using the Gofer class system in the following manner:

```
class Monad m => MonadOPlus m where
   zero :: m a
   (++) :: m a -> m a -> m a
```

That is, a type constructor **m** is a member of the class **MonadOPlus** if it is a member of the class **Monad** (that is, it is equipped with a **result** and **bind**), and if it is also equipped with **zero** and **(++)** operators of the specified types. Of course, the two extra operations must also satisfy some algebraic properties; these are discussed in (Wadler, 1992a; Wadler, 1992b). Note also that **(++)** is used above rather than **plus**, following the example of lists: we will see later on that lists form a monad for which the plus operation is just the familiar append operation **(++)**.

Now since **Parser** is already a monad, it can be made into a monad with a zero and a plus using the following definitions:

```
instance MonadOPlus Parser where
   -- zero :: Parser a
   zero      = \inp -> []

   -- (++) :: Parser a -> Parser a -> Parser a
   p ++ q    = \inp -> (p inp ++ q inp)
```

3.2 Monad comprehension syntax

So far we have seen one advantage of recognising the monadic nature of parsers: the monadic sequencing combinator **bind** handles result values better than the conventional sequencing combinator **seq**. In this section we consider another advantage of the monadic approach, namely that *monad comprehension* syntax can be used to make parsers more compact and easier to read.

As mentioned earlier, many parsers will have a structure as a sequence of **bind**s followed by single call to **result**:

```
p1 `bind` \x1 ->
p2 `bind` \x2 ->
...
pn `bind` \xn ->
result (f x1 x2 ... xn)
```

Gofer provides a special notation for defining parsers of this shape, allowing them to be expressed in the following, more appealing form:

```
[ f x1 x2 ... xn | x1 <- p1
```

```
, x2 <- p2
, ...
, xn <- pn ]
```

In fact, this notation is not specific to parsers, but can be used with any monad (Jones, 1995c). The reader might notice the similarity to the list comprehension notation supported by many functional languages. It was Wadler (1990) who first observed that the comprehension notation is not particular to lists, but makes sense for an arbitrary monad. Indeed, the algebraic properties required of the monad operations turn out to be precisely those required for the notation to make sense. To our knowledge, Gofer is the first language to implement Wadler's *monad comprehension* notation. Using this notation can make parsers much easier to read, and we will use the notation in the remainder of this article.

As our first example of using comprehension notation, we define a parser for recognising specific strings, with the string itself returned as the result:

```
string       :: String -> Parser String
string ""     = [""]
string (x:xs) = [x:xs | _ <- char x, _ <- string xs]
```

That is, if the string to be parsed is empty we just return the empty string as the result; `[""]` is just monad comprehension syntax for `result ""`. Otherwise, we parse the first character of the string using `char`, and then parse the remaining characters using a recursive call to `string`. Without the aid of comprehension notation, the above definition would read as follows:

```
string       :: String -> Parser String
string ""     = result ""
string (x:xs) = char x    `bind` \_ ->
                string xs `bind` \_ ->
                result (x:xs)
```

Note that the parser `string xs` fails if only a prefix of the given string `xs` is recognised in the input. For example, applying the parser `string "hello"` to the input `"hello there"` gives the successful result `[("hello"," there")]`. On the other hand, applying the same parser to `"helicopter"` fails with the result `[]`, even though the prefix `"hel"` of the input can be recognised.

In list comprehension notation, we are not just restricted to *generators* that bind variables to values, but can also use Boolean-valued *guards* that restrict the values of the bound variables. For example, a function `negs` that selects all the negative numbers from a list of integers can be expressed as follows:

```
negs    :: [Int] -> [Int]
negs xs = [x | x <- xs, x < 0]
```

In this case, the expression `x < 0` is a guard that restricts the variable `x` (bound by the generator `x <- xs`) to only take on values less than zero.

Wadler (1990) observed that the use of guards makes sense for an arbitrary

monad with a zero. The monad comprehension notation in Gofer supports this use of guards. For example, the `sat` combinator

```
sat  :: (Char -> Bool) -> Parser Char
sat p = item `bind` \x ->
        if p x then result x else zero
```

can be defined more succinctly using a comprehension with a guard:

```
sat  :: (Char -> Bool) -> Parser Char
sat p = [x | x <- item, p x]
```

We conclude this section by noting that there is another notation that can be used to make monadic programs easier to read: the so-called "do" notation (Jones, 1994; Jones & Launchbury, 1994). For example, using this notation the combinators `string` and `sat` can be defined as follows:

```
string       :: String -> Parser String
string ""     = do { result "" }
string (x:xs) = do { char x ; string xs ; result (x:xs) }

sat          :: (Char -> Bool) -> Parser Char
sat p         = do { x <- item ; if (p x) ; result x }
```

The `do` notation has a couple of advantages over monad comprehension notation: we are not restricted to monad expressions that end with a use of `result`; and generators of the form `_ <- e` that do not bind variables can be abbreviated by `e`. The `do` notation is supported by Gofer, but monad expressions involving parsers typically end with a use of `result` (to compute the result value from the parser), so the extra generality is not really necessary in this case. For this reason, and for simplicity, in this article we only use the comprehension notation. It would be an easy task, however, to translate our definitions into the `do` notation.

4 Combinators for repetition

Parser generators such as Lex and Yacc (Aho *et al.*, 1986) for producing parsers written in C, and Ratatosk (Mogensen, 1993) and Happy (Gill & Marlow, 1995) for producing parsers written in Haskell, typically offer a fixed set of combinators for describing grammars. In contrast, with the method of building parsers as presented in this article the set of combinators is completely extensible: parsers are first-class values, and we have the full power of a functional language at our disposal to define special combinators for special applications.

In this section we define combinators for a number of common patterns of *repetition*. These combinators are not specific to parsers, but can be used with an arbitrary monad with a zero and plus. For clarity, however, we specialise the types of the combinators to the case of parsers.

In subsequent sections we will introduce combinators for other purposes, including handling lexical issues and Gofer's offside rule.

4.1 Simple repetition

Earlier we defined a parser `word` for consuming zero or more letters from the input string. Using monad comprehension notation, the definition is:

```
word :: Parser String
word  = [x:xs | x <- letter, xs <- word] ++ [""]
```

We can easily imagine a number of other parsers that exhibit a similar structure to `word`. For example, parsers for strings of digits or strings of spaces could be defined in precisely the same way, the only difference being be that the component parser `letter` would be replaced by either `digit` or `char ' '`. To avoid defining a number of different parsers with a similar structure, we abstract on the pattern of recursion in `word` and define a general combinator, `many`, that parses sequences of items.

The combinator `many` applies a parser `p` zero or more times to an input string. The results from each application of `p` are returned in a list:

```
many  :: Parser a -> Parser [a]
many p = [x:xs | x <- p, xs <- many p] ++ [[]]
```

Different parsers can be made by supplying different arguments parsers `p`. For example, `word` can be defined just as `many letter`, and the other parsers mentioned above by `many digit` and `many (char ' ')`.

Just as the original `word` parser returns many results in general (decreasing in the number of letters consumed from the input), so does `many p`. Of course, in most cases we will only be interested in the first parse from `many p`, in which `p` is successfully applied as many times as possible. We will return to this point in the next section, when we address the efficiency of parsers.

As another application of `many`, we can define a parser for identifiers. For simplicity, we regard an identifier as a lower-case letter followed by zero or more alphanumeric characters. It would be easy to extend the definition to handle extra characters, such as underlines or backquotes.

```
ident :: Parser String
ident  = [x:xs | x <- lower, xs <- many alphanum]
```

Sometimes we will only be interested in non-empty sequences of items. For this reason we define a special combinator, `many1`, in terms of `many`:

```
many1  :: Parser a -> Parser [a]
many1 p = [x:xs | x <- p, xs <- many p]
```

For example, applying `many1 (char 'a')` to the input `"aaab"` gives the result `[("aaa","b"), ("aa","ab"), ("a","aab")]`, which is the same as for `many (char 'a')`, except that the final pair `("", "aaab")` is no longer present. Note also that `many1 p` may fail, whereas `many p` always succeeds.

Using `many1` we can define a parser for natural numbers:

```
nat :: Parser Int
nat  = [eval xs | xs <- many1 digit]
```

```
      where
        eval xs  = foldl1 op [ord x - ord '0' | x <- xs]
        m `op` n = 10*m + n
```

In turn, nat can be used to define a parser for integers:

```
int :: Parser Int
int  = [-n | _ <- char '-', n <- nat] ++ nat
```

A more sophisticated way to define int is as follows. First try and parse the negation character '-'. If this is successful then return the negation function as the result of the parse; otherwise return the identity function. The final step is then to parse a natural number, and use the function returned by attempting to parse the '-' character to modify the resulting number:

```
int :: Parser Int
int  = [f n | f <- op, n <- nat]
       where
         op = [negate | _ <- char '-'] ++ [id]
```

4.2 Repetition with separators

The many combinators parse sequences of items. Now we consider a slightly more general pattern of repetition, in which separators between the items are involved. Consider the problem of parsing a non-empty list of integers, such as [1,-42,17]. Such a parser can be defined in terms of the many combinator as follows:

```
ints :: Parser [Int]
ints  = [n:ns | _  <- char '['
              , n  <- int
              , ns <- many [x | _ <- char ',', x <- int]
              , _  <- char ']']
```

As was the case in the previous section for the word parser, we can imagine a number of other parsers with a similar structure to ints, so it is useful to abstract on the pattern of repetition and define a general purpose combinator, which we call sepby1. The combinator sepby1 is like many1 in that it recognises non-empty sequences of a given parser p, but different in that the instances of p are separated by a parser sep whose result values are ignored:

```
sepby1         :: Parser a -> Parser b -> Parser [a]
p `sepby1` sep = [x:xs | x  <- p
                       , xs <- many [y | _ <- sep, y <- p]]
```

Note that the fact that the results of the sep parser are ignored is reflected in the type of the sepby1 combinator: the sep parser gives results of type b, but this type does not occur in the type [a] of the results of the combinator.

Now ints can be defined in a more compact form:

```
ints = [ns | _  <- char '['
           , ns <- int `sepby1` char ','
           , _  <- char ']']
```

In fact we can go a little further. The bracketing of parsers by other parsers whose results are ignored — in the case above, the bracketing parsers are `char '['` and `char ']'` — is common enough to also merit its own combinator:

```
bracket :: Parser a -> Parser b -> Parser c -> Parser b
bracket open p close = [x | _ <- open, x <- p, _ <- close]
```

Now `ints` can be defined just as

```
ints = bracket (char '[')
               (int `sepby1` char ',')
               (char ']')
```

Finally, while `many1` was defined in terms of `many`, the combinator `sepby` (for possibly-empty sequences) is naturally defined in terms of `sepby1`:

```
sepby          :: Parser a -> Parser b -> Parser [a]
p `sepby` sep  = (p `sepby1` sep) ++ [[]]
```

4.3 Repetition with meaningful separators

The `sepby` combinators handle the case of parsing sequences of items separated by text that can be ignored. In this final section on repetition, we address the more general case in which the separators themselves carry meaning. The combinators defined in this section are due to Fokker (1995).

Consider the problem of parsing simple arithmetic expressions such as `1+2-(3+4)`, built up from natural numbers using addition, subtraction, and parentheses. The two arithmetic operators are assumed to associate to the left (thus, for example, `1-2-3` should be parsed as `(1-2)-3`), and have the same precedence. The standard BNF grammar for such expressions is written as follows:

expr ::= *expr addop factor* | *factor*
addop ::= + | -
factor ::= *nat* | (*expr*)

This grammar can be translated directly into a combinator parser:

```
expr   :: Parser Int
addop  :: Parser (Int -> Int -> Int)
factor :: Parser Int

expr   = [f x y | x <- expr, f <- addop, y <- factor] ++ factor

addop  = [(+) | _ <- char '+'] ++ [(-) | _ <- char '-']

factor = nat ++ bracket (char '(') expr (char ')')
```

In fact, rather than just returning some kind of parse tree, the **expr** parser above actually evaluates arithmetic expressions to their integer value: the **addop** parser returns a function as its result value, which is used to combine the result values produced by parsing the arguments to the operator.

Of course, however, there is a problem with the **expr** parser as defined above. The fact that the operators associate to the left is taken account of by **expr** being *left-recursive* (the first thing it does is make a recursive call to itself). Thus **expr** never makes any progress, and hence does not terminate.

As is well-known, this kind of non-termination for parsers can be solved by replacing left-recursion by iteration. Looking at the *expr* grammar, we see that an expression is a sequence of *factor*s, separated by *addop*s. Thus the parser for expressions can be re-defined using **many** as follows:

```
expr = [... | x   <- factor
           , fys <- many [(f,y) | f <- addop, y <- factor]]
```

This takes care of the non-termination, but it still remains to fill in the "..." part of the new definition, which computes the value of an expression.

Suppose now that the input string is `"1-2+3-4"`. Then after parsing using `expr`, the variable `x` will be `1` and `fys` will be the list `[((-),2), ((+),3), ((-),4)]`. These can be reduced to a single value `1-2+3-4 = ((1-2)+3)-4 = -2` by folding: the built-in function `foldl` is such that, for example, `foldl g a [b,c,d,e] = ((a 'g' b) 'g' c) 'g' d) 'g' e`. In the present case, we need to take `g` as the function `\x (f,y) -> f x y`, and `a` as the integer `x`:

```
expr = [foldl (\x (f,y) -> f x y) x fys
          | x   <- factor
          , fys <- many [(f,y) | f <- addop, y <- factor]]
```

Now, for example, applying `expr` to the input string `"1+2-(3+4)"` gives the result `[(-4,""), (3,"-(3+4)", (1,"+2-(3+4)")]`, as expected.

Playing the generalisation game once again, we can abstract on the pattern of repetition in **expr** and define a new combinator. The combinator, **chainl1**, parses non-empty sequences of items separated by operators that associate to the left:

```
chainl1       :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chainl1` op = [foldl (\x (f,y) -> f x y) x fys
                    | x   <- p
                    , fys <- many [(f,y) | f <- op, y <- p]]
```

Thus our parser for expressions can now be written as follows:

```
expr   = factor `chainl1` addop

addop  = [(+) | _ <- char '+'] ++ [(-) | _ <- char '-']

factor = nat ++ bracket (char '(') expr (char ')')
```

Most operator parsers will have a similar structure to **addop** above, so it is useful to abstract a combinator for building such parsers:

```
ops    :: [(Parser a, b)] -> Parser b
ops xs = foldr1 (++) [[op | _ <- p] | (p,op) <- xs]
```

The built-in function `foldr1` is such that, for example, `foldr1 g [a,b,c,d] = a 'g' (b 'g' (c 'g' d))`. It is defined for any non-empty list. In the above case then, `foldr1` places the choice operator (++) between each parser in the list. Using `ops`, our `addop` parser can now be defined by

```
addop = ops [(char '+', (+)), (char '-', (-))]
```

A possible inefficiency in the definition of the `chainl1` combinator is the construction of the intermediate list `fys`. This can be avoided by giving a direct recursive definition of `chainl1` that does not make use of `foldl` and `many`, using an accumulating parameter to construct the final result:

```
chainl1       :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chainl1` op = p `bind` rest
                 where
                    rest x = (op `bind` \f ->
                              p  `bind` \y ->
                              rest (f x y)) ++ [x]
```

This definition has a natural operational reading. The parser `p `chainl1` op` first parses a single `p`, whose result value becomes the initial accumulator for the `rest` function. Then it attempts to parse an operator and a single `p`. If successful, the accumulator and the result from `p` are combined using the function `f` returned from parsing the operator, and the resulting value becomes the new accumulator when parsing the remainder of the sequence (using a recursive call to `rest`). Otherwise, the sequence is finished, and the accumulator is returned.

As another interesting application of `chainl1`, we can redefine our earlier parser `nat` for natural numbers such that it does not construct an intermediate list of digits. In this case, the `op` parser does not do any parsing, but returns the function that combines a natural and a digit:

```
nat :: Parser Int
nat  = [ord x - ord '0' | x <- digit] `chainl1` [op]
       where
          m `op` n = 10*m + n
```

Naturally, we can also define a combinator `chainr1` that parses non-empty sequences of items separated by operators that associate to the *right*, rather than to the left. For simplicity, we only give the direct recursive definition:

```
chainr1       :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chainr1` op =
   p `bind` \x ->
      [f x y | f <- op, y <- p `chainr1` op] ++ [x]
```

That is, `p `chainr1` op` first parses a single `p`. Then it attempts to parse an operator and the rest of the sequence (using a recursive call to `chainr1`). If successful,

the pair of results from the first `p` and the rest of the sequence are combined using the function `f` returned from parsing the operator. Otherwise, the sequence is finished, and the result from `p` is returned.

As an example of using `chainr1`, we extend our parser for arithmetic expressions to handle exponentiation; this operator has higher precedence than the previous two operators, and associates to the right:

```
expr   = term   `chainl1` addop

term   = factor `chainr1` expop

factor = nat ++ bracket (char '(') expr (char ')')

addop  = ops [(char '+', (+)), (char '-', (-))]

expop  = ops [(char '^', (^))]
```

For completeness, we also define combinators `chainl` and `chainr` that have the same behaviour as `chainl1` and `chainr1`, except that they can also consume no input, in which case a given value `v` is returned as the result:

```
chainl :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainl p op v = (p `chainl1` op) ++ [v]

chainr :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainr p op v = (p `chainr1` op) ++ [v]
```

In summary then, `chainl` and `chainr` provide a simple way to build parsers for expression-like grammars. Using these combinators avoids the need for transformations to remove left-recursion in the grammar, that would otherwise result in non-termination of the parser. They also avoid the need for left-factorisation of the grammar, that would otherwise result in unnecessary backtracking; we will return to this point in the next section.

5 Efficiency of parsers

Using combinators is a simple and flexible method of building parsers. However, the power of the combinators — in particular, their ability to backtrack and return multiple results — can lead to parsers with unexpected space and time performance if one does not take care. In this section we outline some simple techniques that can be used to improve the efficiency of parsers. Readers interested in further techniques are referred to Röjemo's thesis (1995), which contains a chapter on the use of heap profiling tools in the optimisation of parser combinators.

5.1 Left factoring

Consider the simple problem of parsing and evaluating two natural numbers separated by the addition symbol '+', or by the subtraction symbol '-'. This specification can be translated directly into the following parser:

```
eval :: Parser Int
eval  = add ++ sub
        where
           add = [x+y | x <- nat, _ <- char '+', y <- nat]
           sub = [x-y | x <- nat, _ <- char '-', y <- nat]
```

This parser gives the correct results, but is inefficient. For example, when parsing the string `"123-456"` the number `123` will first be parsed by the `add` parser, that will then fail because there is no '+' symbol following the number. The correct parse will only be found by backtracking in the input string, and parsing the number `123` *again*, this time from within the `sub` parser.

Of course, the way to avoid the possibility of backtracking and repeated parsing is to *left factorise* the `eval` parser. That is, the initial use of `nat` in the component parsers `add` and `sub` should be factorised out:

```
eval = [v | x <- nat, v <- add x ++ sub x]
       where
          add x = [x+y | _ <- char '+', y <- nat]
          sub x = [x+y | _ <- char '-', y <- nat]
```

This new version of `eval` gives the same results as the original version, but requires no backtracking. Using the new `eval`, the string `"123-456"` can now be parsed in linear time. In fact we can go a little further, and right factorise the remaining use of `nat` in both `add` and `sub`. This does not improve the efficiency of `eval`, but arguably gives a cleaner parser:

```
eval = [f x y | x <- nat
              , f <- ops [(char '+', (+)), (char '-', (-))]
              , y <- nat]
```

In practice, most cases where left factorisation of a parser is necessary to improve efficiency will concern parsers for some kind of expression. In such cases, manually factorising the parser will not be required, since expression-like parsers can be built using the `chain` combinators from the previous section, which already encapsulate the necessary left factorisation.

The motto of this section is the following: backtracking is a powerful tool, but it should not be used as a substitute for care in designing parsers.

5.2 Improving laziness

Recall the definition of the repetition combinator `many`:

```
many  :: Parser a -> Parser [a]
many p = [x:xs | x <- p, xs <- many p] ++ [[]]
```

For example, applying `many (char 'a')` to the input `"aaab"` gives the result `[("aaa","b"), ("aa","ab"), ("a","aab"),("","aaab")]`. Since Gofer is lazy, we would expect the a's in the first result `"aaa"` to become available one at a time, as they are consumed from the input. This is not in fact what happens. In practice no part of the result `"aaa"` will be produced until all the a's have been consumed. In other words, **many** is not as lazy as we would expect.

But does this really matter? Yes, because it is common in functional programming to rely on laziness to avoid the creation of large intermediate structures (Hughes, 1989). As noted by Wadler (1985; 1992b), what is needed to solve the problem with **many** is a means to make explicit that the parser **many p** always succeeds. (Even if p itself always fails, **many p** will still succeed, with the empty list as the result value.) This is the purpose of the **force** combinator:

```
force  :: Parser a -> Parser a
force p = \inp -> let x = p inp in
                  (fst (head x), snd (head x)) : tail x
```

Given a parser p that always succeeds, the parser **force p** has the same behaviour as p, except that before any parsing of the input string is attempted the result of the parser is immediately forced to take on the form $(\bot, \bot) : \bot$, where $\bot$ represents a presently undefined value.

Using **force**, the **many** combinator can be re-defined as follows:

```
many  :: Parser a -> Parser [a]
many p = force ([x:xs | x <- p, xs <- many p] ++ [[]])
```

The use of **force** ensures that **many p** and all of its recursive calls return at least one result. The new definition of **many** now has the expected behaviour under lazy evaluation. For example, applying **many (char 'a')** to the partially-defined string $'a' : \bot$ gives the partially-defined result $('a' : \bot, \bot) : \bot$. In contrast, with the old version of **many**, the result for this example is the completely undefined value $\bot$.

Some readers might wonder why **force** is defined using the following selection functions, rather than by pattern matching?

```
fst :: (a,b) -> a      head :: [a] -> a
snd :: (a,b) -> b      tail :: [a] -> [a]
```

The answer is that, depending on the semantics of patterns in the particular implementation language, a definition of **force** using patterns might not have the expected behaviour under lazy evaluation.

5.3 Limiting the number of results

Consider the simple problem of parsing a natural number, or if no such number is present just returning the number 0 as the default result. A first approximation to such a parser might be as follows:

```
number :: Parser Int
number  = nat ++ [0]
```

However, this does not quite have the required behaviour. For example, applying **number** to the input **"hello"** gives the correct result **[(0,"hello")]**. On the other hand, applying **number** to **"123"** gives the result **[(123,""), (0,"123")]**, whereas we only really want the single result **[(123,"")]**.

One solution to the above problem is to make use of *deterministic* parser combinators (see section 7.5) — all parsers built using such combinators are restricted by construction to producing at most one result. A more general solution, however, is to retain the flexibility of the non-deterministic combinators, but to provide a means to make explicit that we are only interested in the first result produced by certain parsers, such as **number**. This is the purpose of the **first** combinator:

```
first  :: Parser a -> Parser a
first p = \inp -> case p inp of
                    []     -> []
                    (x:xs) -> [x]
```

Given a parser **p**, the parser **first p** has the same behaviour as **p**, except that only the first result (if any) is returned. Using **first** we can define a deterministic version **(+++)** of the standard choice combinator **(++)** for parsers:

```
(+++)  :: Parser a -> Parser a -> Parser a
p +++ q = first (p ++ q)
```

Replacing **(++)** by **(+++)** in **number** gives the desired behaviour.

As well as being used to ensure the correct behaviour of parsers, using **(+++)** can also improve their efficiency. As an example, consider a parser that accepts either of the strings **"yellow"** or **"orange"**:

```
colour :: Parser String
colour  = p1 ++ p2
          where
             p1 = string "yellow"
             p2 = string "orange"
```

Recall now the behaviour of the choice combinator **(++)**: it takes a string, applies both argument parsers to this string, and concatenates the resulting lists. Thus in the **colour** example, if **p1** is successfully applied then **p2** will still be applied to the same string, even though it is guaranteed to fail. This inefficiency can be avoided using **(+++)**, which ensures that if **p1** succeeds then **p2** is never applied:

```
colour = p1 +++ p2
         where
            p1 = string "yellow"
            p2 = string "orange"
```

More generally, if we know that a parser of the form **p ++ q** is deterministic (only ever returns at most one result value), then **p +++ q** has the same behaviour, but is more efficient: if **p** succeeds then **q** is never applied. In the remainder of this article it will mostly be the **(+++)** choice combinator that is used. For reasons of efficiency,

in the combinator libraries that accompany this article, the repetition combinators from the previous section are defined using `(+++)` rather than `(++)`.

We conclude this section by asking why `first` is defined by pattern matching, rather than by using the selection function `take :: Int -> [a] -> [a]` (where, for example, `take 3 "parsing" = "par"`):

```
first p = \inp -> take 1 (p inp)
```

The answer concerns the behaviour under lazy evaluation. To see the problem, let us unfold the use of `take` in the above definition:

```
first p = \inp -> case p inp of
                    []     -> []
                    (x:xs) -> x : take 0 xs
```

When the sub-expression `take 0 xs` is evaluated, it will yield `[]`. However, under lazy evaluation this computation will be suspended until its value is required. The effect is that the list `xs` may be retained in memory for some time, when in fact it can safely be discarded immediately. This is an example of a *space leak*. The definition of `first` using pattern matching does not suffer from this problem.

6 Handling lexical issues

Traditionally, a string to be parsed is not supplied directly to a parser, but is first passed through a lexical analysis phase (or *lexer*) that breaks the string into a sequence of tokens (Aho *et al.*, 1986). Lexical analysis is a convenient place to remove white-space (spaces, newlines, and tabs) and comments from the input string, and to distinguish between identifiers and keywords.

Since lexers are just simple parsers, they can be built using parser combinators, as discussed by Hutton (1992). However, as we shall see in this section, the need for a separate lexer can often be avoided (even for substantial grammars such as that for Gofer), with lexical issues being handled within the main parser by using some special purpose combinators.

6.1 White-space, comments, and keywords

We begin by defining a parser that consumes white-space from the beginning of a string, with a dummy value `()` returned as result:

```
spaces :: Parser ()
spaces  = [() | _ <- many1 (sat isSpace)]
          where
             isSpace x =
                (x == ' ') || (x == '\n') || (x == '\t')
```

Similarly, a single-line Gofer comment can be consumed as follows:

```
comment :: Parser ()
comment  = [() | _ <- string "--"
               , _ <- many (sat (\x -> x /= '\n'))]
```

We leave it as an exercise for the reader to define a parser for consuming multi-line Gofer comments {- ... -}, which can be nested.

After consuming white-space, there may still be a comment left to consume from the input string. Dually, after a comment there may still be white-space. Thus we are motivated to defined a special parser that repeatedly consumes white-space and comments until no more remain:

```
junk :: Parser ()
junk  = [() | _ <- many (spaces +++ comment)]
```

Note that while **spaces** and **comment** can fail, the **junk** parser always succeeds. We define two combinators in terms of **junk**: **parse** removes junk *before* applying a given parser, and **token** removes junk *after* applying a parser:

```
parse  :: Parser a -> Parser a
parse p = [v | _ <- junk, v <- p]

token  :: Parser a -> Parser a
token p = [v | v <- p, _ <- junk]
```

With the aid of these two combinators, parsers can be modified to ignore white-space and comments. Firstly, **parse** is applied once to the parser as a whole, ensuring that input to the parser begins at a significant character. And secondly, **token** is applied once to all sub-parsers that consume complete tokens, thus ensuring that the input always remains at a significant character.

Examples of parsers for complete tokens are **nat** and **int** (for natural numbers and integers), parsers of the form **string xs** (for symbols and keywords), and **ident** (for identifiers). It is useful to define special versions of these parsers — and more generally, special versions of any user-defined parsers for complete tokens — that encapsulate the necessary application of **token**:

```
natural      :: Parser Int
natural       = token nat

integer      :: Parser Int
integer       = token int

symbol       :: String -> Parser String
symbol xs     = token (string xs)

identifier   :: [String] -> Parser String
identifier ks = token [x | x <- ident, not (elem x ks)]
```

Note that **identifier** takes a list of keywords as an argument, where a keyword is a string that is not permitted as an identifier. For example, in Gofer the strings "**data**" and "**where**" (among others) are keywords. Without the keyword check, parsers defined in terms of **identifier** could produce unexpected results, or involve unnecessary backtracking to construct the correct parse of the input string.

6.2 A parser for λ-expressions

To illustrate the use of the new combinators given above, let us define a parser for simple λ-expressions extended with a "`let`" construct for local definitions. Parsed expressions will be represented in Gofer as follows:

```
data Expr = App Expr Expr           -- application
          | Lam String Expr         -- lambda abstraction
          | Let String Expr Expr    -- local definition
          | Var String              -- variable
```

Now a parser `expr :: Parser Expr` can be defined by:

```
expr     = atom `chainl1` [App]

atom     = lam +++ local +++ var +++ paren

lam      = [Lam x e | _ <- symbol "\\"
                    , x <- variable
                    , _ <- symbol "->"
                    , e <- expr]

local    = [Let x e e' | _  <- symbol "let"
                       , x  <- variable
                       , _  <- symbol "="
                       , e  <- expr
                       , _  <- symbol "in"
                       , e' <- expr]

var      = [Var x | x <- variable]

paren    = bracket (symbol "(") expr (symbol ")")

variable = identifier ["let","in"]
```

Note how the `expr` parser handles white-space and comments by using the `symbol` parser in place of `string` and `char`. Similarly, the keywords "`let`" and "`in`" are handled by using `identifier` to define the parser for variables. Finally, note how applications `(f e1 e2 ... en)` are parsed in the form `(((f e1) e2) ... )` by using the `chainl1` combinator.

7 Factorising the parser monad

Up to this point in the article, combinator parsers have been our only example of the notion of a monad. In this section we define a number of other monads related to the parser monad, leading up to a modular reformulation of the parser monad in terms of two simpler monads (Jones, 1995a). The immediate benefit is that, as

we shall see, the basic parser combinators no longer need to be defined explicitly. Rather, they arise automatically as a special case of lifting monad operations from a base monad m to a certain other monad parameterised over m. This also means that, if we change the nature of parsers by modifying the base monad (for example, limiting parsers to producing at most one result), new combinators for the modified monad of parsers are also defined automatically.

7.1 The exception monad

Before starting to define other monads, it is useful to first focus briefly on the intuition behind the use of monads in functional programming (Wadler, 1992a).

The basic idea behind monads is to distinguish the ***values*** that a computation can produce from the *computation* itself. More specifically, given a monad `m` and a type `a`, we can think of `m a` as the type of computations that yield results of type `a`, with the nature of the computation captured by the type constructor `m`. The combinators `result` and `bind` (with `zero` and `(++)` if appropriate) provide a means to structure the building of such computations:

```
result :: m a
bind   :: m a -> (a -> m b) -> m b
zero   :: m a
(++)   :: m a -> m a -> m a
```

From a computational point of view, `result` converts values into computations that yield those values; `bind` chains two computations together in sequence, with results of the first computation being made available for use in the second; `zero` is the trivial computation that does nothing; and finally, `(++)` is some kind of choice operation for computations.

Consider, for example, the type constructor `Maybe`:

```
data Maybe a = Just a | Nothing
```

We can think of a value of type `Maybe a` as a computation that either succeeds with a value of type `a`, or fails, producing no value. Thus, the type constructor `Maybe` captures computations that have the possibility to fail.

Defining the monad combinators for a given type constructor is usually just a matter of making the "obvious definitions" suggested by the types of the combinators. For example, the type constructor `Maybe` can be made into a monad with a zero and plus using the following definitions:

```
instance Monad Maybe where
   -- result          :: a -> Maybe a
   result x            = Just x

   -- bind            :: Maybe a -> (a -> Maybe b) -> Maybe b
   (Just x) `bind` f = f x
   Nothing  `bind` f = Nothing
```

```
instance MonadOPlus Maybe where
   -- zero          :: Maybe a
   zero              = Nothing

   -- (++)          :: Maybe a -> Maybe a -> Maybe a
   Just x  ++ y     = Just x
   Nothing ++ y     = y
```

That is, `result` converts a value into a computation that succeeds with this value; `bind` is a sequencing operator, with a successful result from the first computation being available for use in the second computation; `zero` is the computation that fails; and finally, `(++)` is a (deterministic) choice operator that returns the first computation if it succeeds, and the second otherwise.

Since failure can be viewed as a simple kind of exception, `Maybe` is sometimes called the *exception* monad in the literature (Spivey, 1990).

7.2 The non-determinism monad

A natural generalisation of `Maybe` is the list type constructor `[]`. While a value of type `Maybe a` can be thought of as a computation that either succeeds with a single result of type `a` or fails, a value of type `[a]` can be thought of as a computation that has the possibility to succeed with any number of results of type `a`, including zero (which represents failure). Thus the list type constructor `[]` can be used to capture *non-deterministic* computations.

Now `[]` can be made into a monad with a zero and plus:

```
instance Monad [] where
   -- result        :: a -> [a]
   result x         = [x]

   -- bind          :: [a] -> (a -> [b]) -> [b]
   []     `bind` f  = []
   (x:xs) `bind` f  = f x ++ (xs `bind` f)

instance MonadOPlus [] where
   -- zero          :: [a]
   zero             = []

   -- (++)          :: [a] -> [a] -> [a]
   []     ++ ys     = ys
   (x:xs) ++ ys     = x : (xs ++ ys)
```

That is, `result` converts a value into a computation that succeeds with this single value; `bind` is a sequencing operator for non-deterministic computations; `zero` always fails; and finally, `(++)` is a (non-deterministic) choice operator that appends the results of the two argument computations.

7.3 The state-transformer monad

Consider the (binary) type constructor `State`:

```
type State s a = s -> (a,s)
```

Values of type `State s a` can be interpreted as follows: they are computations that take an initial state of type `s`, and yield a value of type `a` together with a new state of type `s`. Thus, the type constructor `State s` obtained by applying `State` to a single type `s` captures computations that involve state of type `s`. We will refer to values of type `State s a` as *stateful computations*.

Now `State s` can be made into a monad:

```
instance Monad (State s) where
   -- result  :: a -> State s a
   result v    = \s -> (v,s)

   -- bind    :: State s a -> (a -> State s b) -> State s b
   st `bind` f = \s -> let (v,s') = st s in f v s'
```

That is, `result` converts a value into a stateful computation that returns that value without modifying the internal state, and `bind` composes two stateful computations in sequence, with the result value from the first being supplied as input to the second. Thinking pictorially in terms of boxes and wires is a useful aid to becoming familiar with these two operations (Jones & Launchbury, 1994).

The *state-transformer* monad `State s` does not have a zero and a plus. However, as we shall see in the next section, the *parameterised* state-transformer monad over a given based monad `m` *does* have a zero and a plus, provided that `m` does.

To allow us to access and modify the internal state, a few extra operations on the monad `State s` are introduced. The first operation, `update`, modifies the state by applying a given function, and returns the old state as the result value of the computation. The remaining two operations are defined in terms of `update`: `set` replaces the state with a new state, and returns the old state as the result; `fetch` returns the state without modifying it.

```
update  :: (s -> s) -> State s s
set     :: s -> State s s
fetch   :: State s s

update f = \s -> (s, f s)
set s    = update (\_ -> s)
fetch    = update id
```

In fact `State s` is not the only monad for which it makes sense to define these operations. For this reason we encapsulate the extra operations in a class, so that the same names can be used for the operations of different monads:

```
class Monad m => StateMonad m s where
   update :: (s -> s) -> m s
```

```
set    :: s -> m s
fetch  :: m s

set s   = update (\_ -> s)
fetch   = update id
```

This declaration can be read as follows: a type constructor `m` and a type `s` are together a member of the class `StateMonad` if `m` is a member of the class `Monad`, and if `m` is also equipped with `update`, `set`, and `fetch` operations of the specified types. Moreover, the fact that `set` and `fetch` can be defined in terms of `update` is also reflected in the declaration, by means of default definitions.

Now because `State s` is already a monad, it can be made into a state monad using the `update` operation as defined earlier:

```
instance StateMonad (State s) s where
   -- update :: (s -> s) -> State s s
   update f   = \s -> (s, f s)
```

7.4 The parameterised state-transformer monad

Recall now our type of combinator parsers:

```
type Parser a = String -> [(a,String)]
```

We see now that parsers combine two kinds of computation: non-deterministic computations (the result of a parser is a list), and stateful computations (the state is the string being parsed). Abstracting from the specific case of returning a list of results, the `Parser` type gives rise to a generalised version of the `State` type constructor that applies a given type constructor `m` to the result of the computation:

```
type StateM m s a = s -> m (a,s)
```

Now `StateM m s` can be made into a monad with a zero and a plus, by inheriting the monad operations from the base monad `m`:

```
instance Monad m => Monad (StateM m s) where
   -- result   :: a -> StateM m s a
   result v     = \s -> result (v,s)

   -- bind     :: StateM m s a ->
   --                (a -> StateM m s b) -> StateM m s b
   stm 'bind' f = \s -> stm s 'bind' \(v,s') -> f v s'

instance MonadOPlus m => MonadOPlus (StateM m s) where
   -- zero     :: StateM m s a
   zero         = \s -> zero

   -- (++)     :: StateM m s a -> StateM m s a -> StateM m s a
   stm ++ stm'  = \s -> stm s ++ stm' s
```

That is, `result` converts a value into a computation that returns this value without modifying the internal state; `bind` chains two computations together; `zero` is the computation that fails regardless of the input state; and finally, `(++)` is a choice operation that passes the same input state through to both of the argument computations, and combines their results.

In the previous section we defined the extra operations `update`, `set` and `fetch` for the monad `State s`. Of course, these operations can also be defined for the *parameterised state-transformer monad* `StateM m s`. As previously, we only need to define `update`, the remaining two operations being defined automatically via default definitions:

```
instance Monad m => StateMonad (StateM m s) s where
   -- update :: Monad m => (s -> s) -> StateM m s s
   update f   = \s -> result (s, f s)
```

7.5 The parser monad revisited

Recall once again our type of combinator parsers:

```
type Parser a = String -> [(a,String)]
```

This type can now be re-expressed using the parameterised state-transformer monad `StateM m s` by taking `[]` for `m`, and `String` for `s`:

```
type Parser a = StateM [] String a
```

But why view the `Parser` type in this way? The answer is that all the basic parser combinators no longer need to be defined explicitly (except one, the parser `item` for single characters), but rather arise as an instance of the general case of extending monad operations from a type constructor `m` to the type constructor `StateM m s`. More specifically, since `[]` forms a monad with a zero and a plus, so does `State [] String`, and hence Gofer automatically provides the following combinators:

```
result :: a -> Parser a
bind   :: Parser a -> (a -> Parser b) -> Parser b
zero   :: Parser a
(++)   :: Parser a -> Parser a -> Parser a
```

Moreover, defining the parser monad in this modular way in terms of `StateM` means that, if we change the type of parsers, then new combinators for the modified type are also defined automatically. For example, consider replacing

```
type Parser a = StateM [] String a
```

by a new definition in which the list type constructor `[]` (which captures non-deterministic computations that can return many results) is replaced by the `Maybe` type constructor (which captures deterministic computations that either fail, returning no result, or succeed with a single result):

```
data Maybe a  = Just a | Nothing

type Parser a = StateM Maybe String a
```

Since **Maybe** forms a monad with a zero and a plus, so does the re-defined **Parser** type constructor, and hence Gofer automatically provides **result**, **bind**, **zero**, and (++) combinators for deterministic parsers. In earlier approaches that do not exploit the monadic nature of parsers (Wadler, 1985; Hutton, 1992; Fokker, 1995), the basic combinators would have to be re-defined by hand.

The only basic parsing primitive that does not arise from the monadic structure of the **Parser** type is the parser **item** for consuming single characters:

```
item :: Parser Char
item  = \inp -> case inp of
                  []     -> []
                  (x:xs) -> [(x,xs)]
```

However, **item** can now be re-defined in monadic style. We first fetch the current state (the input string); if the string is empty then the **item** parser fails, otherwise the first character is consumed (by applying the **tail** function to the state), and returned as the result value of the parser:

```
item  = [x | (x:_) <- update tail]
```

The advantage of the monadic definition of **item** is that it does not depend upon the internal details of the **Parser** type. Thus, for example, it works equally well for both the non-deterministic and deterministic versions of **Parser**.

8 Handling the offside rule

Earlier (section 6) we showed that the need for a lexer to handle white-space, comments, and keywords can be avoided by using special combinators within the main parser. Another task usually performed by a lexer is handling the Gofer *offside rule*. This rule allows the grouping of definitions in a program to be indicated using indentation, and is usually implemented by the lexer inserting extra tokens (concerning indentation) into its output stream.

In this section we show that Gofer's offside rule can be handled in a simple and natural manner without a separate lexer, by once again using special combinators. Our approach was inspired by the monadic view of parsers, and is a development of an idea described earlier by Hutton (1992).

8.1 The offside rule

Consider the following simple Gofer program:

```
a = b + c
    where
      b = 10
```

```
        c = 15 - 5
d = a * 2
```

It is clear from the use of indentation that a and d are intended to be global definitions, with b and c local definitions to a. Indeed, the above program can be viewed as a shorthand for the following program, in which the grouping of definitions is made explicit using special brackets and separators:

```
{ a = b + c
      where
         { b = 10
         ; c = 15 - 5 }
; d = a * 2 }
```

How the grouping of Gofer definitions follows from their indentation is formally specified by the *offside rule*. The essence of the rule is as follows: consecutive definitions that begin in the same column c are deemed to be part of the same group. To make parsing easier, it is further required that the remainder of the text of each definition (excluding white-space and comments, of course) in a group must occur in a column strictly greater than c. In terms of the offside rule then, definitions a and d in the example program above are formally grouped together (and similarly for b and c) because they start in the same column as one another.

8.2 Modifying the type of parsers

To implement the offside rule, we will have to maintain some extra information during parsing. First of all, since column numbers play a crucial role in the offside rule, parsers will need to know the column number of the first character in their input string. In fact, it turns out that parsers will also require the current line number. Thus our present type of combinator parsers,

```
type Parser a = StateM [] String a
```

is revised to the following type, in which the internal state of a parser now contains a (line,column) position in addition to a string:

```
type Parser a = StateM [] Pstring a

type Pstring  = (Pos,String)

type Pos      = (Int,Int)
```

In addition, parsers will need to know the starting position of the current definition being parsed — if the offside rule is not in effect, this *definition position* can be set with a negative column number. Thus our type of parsers is revised once more, to take the current definition position as an extra argument:

```
type Parser a = Pos -> StateM [] Pstring a
```

Another option would have been to maintain the definition position in the parser state, along with the current position and the string to be parsed. However, definition positions can be nested, and supplying the position as an extra argument to parsers — as opposed to within the parser state — is more natural from the point of view of implementing nesting of positions.

Is the revised `Parser` type still a monad? Abstracting from the details, the body of the `Parser` type definition is of the form `s -> m a` (in our case `s` is `Pos`, `m` is the monad `StateM [] Pstring`, and `a` is the parameter type `a`.) We recognise this as being similar to the type `s -> m (a,s)` of parameterised state-transformers, the difference being that the type `s` of states no longer occurs in the type of the result: in other words, the state can be read, but not modified. Thus we can think of `s -> m a` as the type of *parameterised state-readers*. The monadic nature of this type is the topic of the next section.

8.3 The parameterised state-reader monad

Consider the type constructor `ReaderM`, defined as follows:

```
type ReaderM m s a = s -> m a
```

In a similar way to `StateM m s`, `ReaderM m s` can be made into a monad with a zero and a plus, by inheriting the monad operations from the base monad `m`:

```
instance Monad m => Monad (ReaderM m s) where
   -- result   :: a -> ReaderM m s a
   result v     = \s -> result v

   -- bind     :: ReaderM m s a ->
   --                (a -> ReaderM m s b) -> ReaderM m s b
   srm `bind` f = \s -> srm s `bind` \v -> f v s

instance MonadOPlus m => MonadOPlus (ReaderM m s) where
   -- zero     :: ReaderM m s a
   zero         = \s -> zero

   -- (++)     :: ReaderM m s a ->
   --                ReaderM m s a -> ReaderM m s a
   srm ++ srm'  = \s -> srm s ++ srm' s
```

That is, `result` converts a value into a computation that returns this value without consulting the state; `bind` chains two computations together, with the same state being passed to both computations (contrast with the `bind` operation for `StateM`, in which the second computation receives the new state produced by the first computation); `zero` is the computation that fails; and finally, `(++)` is a choice operation that passes the same state to both of the argument computations.

To allow us to access and set the state, a couple of extra operations on the *parameterised state-reader* monad `ReaderM m s` are introduced. As for `StateM`, we

encapsulate the extra operations in a class. The operation **env** returns the state as the result of the computation, while **setenv** replaces the current state for a given computation with a new state:

```
class Monad m => ReaderMonad m s where
   env    :: m s
   setenv :: s -> m a -> m a

instance Monad m => ReaderMonad (ReaderM m s) s where
   -- env      :: Monad m => ReaderM m s s
   env          = \s -> result s

   -- setenv   :: Monad m => s ->
   --                 ReaderM m s a -> ReaderM m s a
   setenv s srm = \_ -> srm s
```

The name **env** comes from the fact that one can think of the state supplied to a state-reader as being a kind of *env*ironment. Indeed, in the literature state-reader monads are sometimes called *environment* monads.

8.4 The new parser combinators

Using the `ReaderM` type constructor, our revised type of parsers

```
type Parser a = Pos -> StateM [] Pstring a
```

can now be expressed as follows:

```
type Parser a = ReaderM (StateM [] Pstring) Pos a
```

Now since `[]` forms a monad with a zero and a plus, so does `StateM [] Pstring`. and hence so does `ReaderM (StateM [] Pstring) Pos`. Thus Gofer automatically provides **result**, **bind**, **zero**, and **(++)** operations for parsers that can handle the offside rule. Since the type of parsers is now defined in terms of `ReaderM` at the top level, the extra operations **env** and **setenv** are also provided for parsers. Moreover, the extra operation **update** (and the derived operations **set** and **fetch**) from the underlying state monad can be lifted to the new type of parsers — or more generally, to any parameterised state-reader monad — by ignoring the environment:

```
instance StateMonad m a => StateMonad (ReaderM m s) a where
   -- update :: StateMonad m a => (a -> a) -> ReaderM m s a
   update f    = \_ -> update f
```

Now that the internal state of parsers has been modified (from **String** to **Pstring**), the parser **item** for consuming single characters from the input must also be modified. The new definition for **item** is similar to the old,

```
item :: Parser Char
item  = [x | (x:_) <- update tail]
```

except that the `item` parser now fails if the position of the character to be consumed is not *onside* with respect to current definition position:

```
item :: Parser Char
item  = [x | (pos,x:_) <- update newstate
           , defpos    <- env
           , onside pos defpos]
```

A position is onside if its column number is strictly greater than the current definition column. However, the first character of a new definition begins in the same column as the definition column, so this is handled as a special case:

```
onside              :: Pos -> Pos -> Bool
onside (l,c) (dl,dc) = (c > dc) || (l == dl)
```

The remaining auxiliary function, `newstate`, consumes the first character from the input string, and updates the current position accordingly (for example, if a newline character was consumed, the current line number is incremented, and the current column number is set back to zero):

```
newstate :: Pstring -> Pstring
newstate ((l,c),x:xs)
   = (newpos,xs)
     where
        newpos = case x of
                    '\n' -> (l+1,0)
                    '\t' -> (l,((c `div` 8)+1)*8)
                    _    -> (l,c+1)
```

One aspect of the offside rule still remains to be addressed: for the purposes of this rule, white-space and comments are not significant, and should always be successfully consumed even if they contain characters that are not onside. This can be handled by temporarily setting the definition position to $(0,-1)$ within the `junk` parser for white-space and comments:

```
junk :: Parser ()
junk  = [() | _ <- setenv (0,-1) (many (spaces +++ comment))]
```

All that remains now is to define a combinator that parses a sequence of definitions subject to the Gofer offside rule:

```
many1_offside  :: Parser a -> Parser [a]
many1_offside p = [vs | (pos,_) <- fetch
                      , vs      <- setenv pos (many1 (off p))]
```

That is, `many1_offside p` behaves just as `many1 (off p)`, except that within this parser the definition position is set to the current position. (There is no need to skip white-space and comments before setting the position, since this will already have been effected by proper use of the lexical combinators `token` and `parse`.) The auxiliary combinator `off` takes care of setting the definition position locally for

each new definition in the sequence, where a new definition begins if the column position equals the definition column position:

```
off  :: Parser a -> Parser a
off p = [v | (dl,dc)   <- env
           , ((l,c),_) <- fetch
           , c == dc
           , v         <- setenv (l,dc) p]
```

For completeness, we also define a combinator `many_offside` that has the same behaviour as the combinator `many1_offside`, except that it can also parse an empty sequence of definitions:

```
many_offside :: Parser a -> Parser [a]
many_offside p = many1_offside p +++ [[]]
```

To illustrate the use of the new combinators defined above, let us modify our parser for λ-expressions (section 6.2) so that the "`let`" construct permits non-empty sequences of local definitions subject to the offside rule. The datatype `Expr` of expressions is first modified so that the `Let` constructor has type `[(String,Expr)] -> Expr` instead of `String -> Expr -> Expr`:

```
data Expr = ...
          | Let [(String,Expr)] Expr
          | ...
```

The only part of the parser that needs to be modified is the parser `local` for local definitions, which now accepts sequences:

```
local = [Let ds e | _  <- symbol "let"
                  , ds <- many1_offside defn
                  , _  <- symbol "in"
                  , e  <- expr]

defn  = [(x,e) | x <- identifier
               , _ <- symbol "="
               , e <- expr]
```

We conclude this section by noting that the use of the offside rule when laying out sequences of Gofer definitions is not mandatory. As shown in our initial example, one also has the option to include explicit layout information in the form of parentheses "{" and "}" around the sequence, with definitions separated by semi-colons ";". We leave it as an exercise to the reader to use `many_offside` to define a combinator that implements this convention.

In summary then, to permit combinator parsers to handle the Gofer offside rule, we changed the type of parsers to include some positional information, modified the `item` and `junk` combinators accordingly, and defined two new combinators: `many1_offside` and `many_offside`. All other necessary redefining of combinators is done automatically by the Gofer type system.

9 Acknowledgements

The first author was employed by the University of Utrecht during part of the writing of this article, for which funding is gratefully acknowledged.

Special thanks are due to Luc Duponcheel for many improvements to the implementation of the combinator libraries in Gofer (particularly concerning the use of type classes and restricted type synonyms), and to Mark P. Jones for detailed comments on the final draft of this article.

10 Appendix: a parser for data definitions

To illustrate the monadic parser combinators developed in this article in a real-life setting, we consider the problem of parsing a sequence of Gofer datatype definitions. An example of such a sequence is as follows:

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

data Tree a b = Leaf a
              | Node (Tree a b, b, Tree a b)
```

Within the parser, datatypes will be represented as follows:

```
type Data = (String,              -- type name
            [String],             -- parameters
            [(String,[Type])])    -- constructors and arguments
```

The representation `Type` for types will be treated shortly. A parser `datadecls :: Parser [Data]` for a sequence of datatypes can now be defined by

```
datadecls   = many_offside datadecl

datadecl    = [(x,xs,b) | _  <- symbol "data"
                        , x  <- constructor
                        , xs <- many variable
                        , _  <- symbol "="
                        , b  <- condecl `sepby1` symbol "|"]

constructor = token [(x:xs) | x  <- upper
                            , xs <- many alphanum]

variable    = identifier ["data"]

condecl     = [(x,ts) | x  <- constructor
                      , ts <- many type2]
```

There are a couple of points worth noting about this parser. Firstly, all lexical issues (white-space and comments, the offside rule, and keywords) are handled by combinators. And secondly, since `constructor` is a parser for a complete token, the `token` combinator is applied within its definition.

Within the parser, types will be represented as follows:

```
data Type   = Arrow Type Type   -- function
            | Apply Type Type   -- application
            | Var String        -- variable
            | Con String        -- constructor
            | Tuple [Type]      -- tuple
            | List Type         -- list
```

A parser `type0 :: Parser Type` for types can now be defined by

```
type0       = type1 `chainr1` [Arrow | _ <- symbol "->"]
type1       = type2 `chainl1` [Apply]
type2       = var +++ con +++ list +++ tuple

var         = [Var x | x <- variable]

con         = [Con x | x <- constructor]

list        = [List x | x <- bracket
                              (symbol "[")
                              type0
                              (symbol "]")]

tuple       = [f ts | ts <- bracket
                              (symbol "(")
                              (type0 `sepby` symbol ",")
                              (symbol ")")]
              where f [t] = t
                    f ts  = Tuple ts
```

Note how `chainr1` and `chainl1` are used to handle parsing of function-types and application. Note also that (as in Gofer) building a singleton tuple `(t)` of a type `t` is not possible, since `(t)` is treated as a parenthesised expression.

References

Aho, A., Sethi, R., & Ullman, J. (1986). *Compilers — principles, techniques and tools.* Addison-Wesley.

Burge, W.H. (1975). *Recursive programming techniques.* Addison-Wesley.

Fokker, Jeroen. 1995 (May). Functional parsers. *Lecture notes of the Baastad Spring school on functional programming.*

Gill, Andy, & Marlow, Simon. 1995 (Jan.). *Happy: the parser generator for Haskell.* University of Glasgow.

Hughes, John. (1989). Why functional programming matters. *The computer journal,* **32**(2), 98–107.

Hutton, Graham. (1992). Higher-order functions for parsing. *Journal of functional programming,* **2**(3), 323–343.

Jones, Mark P. (1994). *Gofer 2.30a release notes.* Unpublished manuscript.

Jones, Mark P. (1995a). Functional programming beyond the Hindley/Milner type system. *Proc. lecture notes of the Baastad spring school on functional programming.*

Jones, Mark P. (1995b). *The Gofer distribution.* Available from the University of Nottingham: `http://www.cs.nott.ac.uk/Department/Staff/mpj/`.

Jones, Mark P. (1995c). A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of functional programming,* **5**(1), 1–35.

Jones, Simon Peyton, & Launchbury, John. (1994). *State in Haskell.* University of Glasgow.

Landin, Peter. (1966). The next 700 programming languages. *Communications of the ACM,* **9**(3).

Mogensen, Torben. (1993). *Ratatosk: a parser generator and scanner generator for Gofer.* University of Copenhagen (DIKU).

Moggi, Eugenio. (1989). Computation lambda-calculus and monads. *Proc. IEEE symposium on logic in computer science.* A extended version of the paper is available as a technical report from the University of Edinburgh.

Röjemo, Niklas. (1995). *Garbage collection and memory efficiency in lazy functional languages.* Ph.D. thesis, Chalmers University of Technology.

Spivey, Mike. (1990). A functional theory of exceptions. *Science of computer programming,* **14**, 25–42.

Wadler, Philip. (1985). How to replace failure by a list of successes. *Proc. conference on functional programming and computer architecture.* Springer–Verlag.

Wadler, Philip. (1990). Comprehending monads. *Proc. ACM conference on Lisp and functional programming.*

Wadler, Philip. (1992a). The essence of functional programming. *Proc. principles of programming languages.*

Wadler, Philip. (1992b). Monads for functional programming. Broy, Manfred (ed), *Proc. Marktoberdorf Summer school on program design calculi.* Springer–Verlag.

Due to copyright restrictions, the following article is not included in this collection, but can be found in:

The Design of a Pretty-printing Library by John Hughes
Lecture Notes in Computer Science 925
Johan Jeuring, Erik Meijer (Eds.)
Springer Verlag Berlin Heidelberg 1995

Microprocessor Specification in Hawk

John Matthews, John Launchbury, Byron Cook
Oregon Graduate Institute

Abstract. Modern microprocessors require an immense investment of time and effort to create and verify, from the high-level architectural design downwards. We are exploring ways to increase the productivity of design engineers by creating a domain-specific language for specifying and simulating processor architectures. We believe that the structuring principles used in modern functional programming languages, such as static typing, parametric polymorphism, first-class functions, and lazy evaluation provide a good formalism for such a domain-specific language, and have made initial progress by creating a library on top of the functional language Haskell. We have specified the integer subset of a pipelined DLX microprocessor, including bypass logic, load-hazard resolution, and speculative branch execution. Two key abstractions of this library are the *signal* abstract data type (ADT), which models the simulation history of a wire, and the *transaction* ADT, which models the state of an entire instruction as it travels through the microprocessor. We are currently using the same techniques to model the architecture of modern superscalar microprocessors.

Introduction

Modern microprocessor technologies have substantially increased processor performance. For example, *pipelining* allows a processor to overlap the execution of several instructions at once. With *superscalar* execution, multiple instructions are read per clock cycle. *Out-of-order execution*, where some instructions that logically come after a given instruction may be executed before the given instruction, can also greatly increase processor speed [Jon91]. All of these technologies dramatically increase design complexity. In fact, creating and verifying these designs is a significant proportion of the total microprocessor development lifecycle. As the number of possible gates in future microprocessors increases exponentially, so too does design complexity.

At OGI, we have developed the *Hawk* library for building executable specifications of microprocessors, concentrating on the level of micro-architecture. The Hawk library constitutes the initial phase of a project that we hope will lead towards an independent language. In the meantime we have in essence embedded our language into Haskell, a strongly-typed functional language with lazy (demand-driven) evaluation, first-class functions, and parametric polymorphism [HPF96] [Pet97].

The library makes essential use of these features. As an example, we have used Hawk to specify and simulate the integer portion of the DLX [HP95] microprocessor. The DLX is a complete microprocessor and is a widely used model among researchers. Several DLX simulators exist, as well as a version of the Gnu C compiler that generates DLX assembly instructions [DLX97]. The processor

The authors are supported by Intel and Air Force Material Command (Contract F19628-93-C-0069)

John Matthews is supported through an NSF fellowship

includes the most common instructions found in commercial RISC processors. Our specification, including data and control hazard resolution, is only two pages of Hawk code. A non-pipelined version of the processor was specified in half of a page.

In this report, we introduce the concepts behind the Hawk library. Rather than attempting a patient explanation of the whole of the DLX with all of its inherent complexity, we have chosen to exhibit the techniques on a considerably simplified model. A corresponding annotated specification of the DLX itself can be found in [Hawk97].

The Hawk Library

We start with a simple example that introduces several functions used in later examples. Consider the resettable counter circuit of Figure 1.

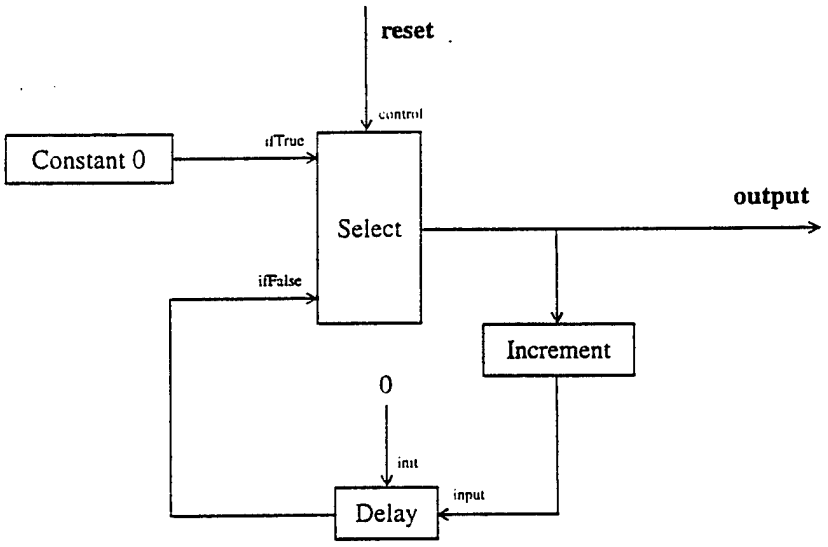


FIGURE 1. Resettable Counter. A simple circuit that counts the number of clock cycles between reset signals.

The *reset* wire is Boolean valued, while the other wires are integer valued. Of course, in silicon, integer-valued wires are represented by a vector of Boolean wires, but as a design abstraction, a Hawk user may choose to use a single wire. The circuit counts (and outputs) the number of clock cycles since *reset* was last asserted.

Signals

Notice that there is no explicit clock in the diagram. Rather, each wire in the diagram carries a *signal* (integer or boolean valued) which is an implicitly clocked value. The output of a circuit only changes between clock cycles. We build signals using an abstract type constructor called `Signal`. As a men-

tal model we could think of a value of type `Signal a` as a function from integers to values of type `a`[1].

```
type Signal a = (Int -> a)
```

The integers denote the current time, measured as the number of clock cycles since the start of the simulation. Circuits and components of circuits are represented as functions from signals to signals. This view of signals is used extensively in the hardware verification community [Mel88] [WC94].

In the resettable counter example above, the *constant 0* circuit outputs zero on every clock cycle. The *select* component chooses between its inputs on each clock cycle depending on the value of *reset*. If *reset* is asserted on a given cycle (has value *true*), then the output is equal to *select*'s top input, in this case zero. If *reset* is not asserted, then its output is the value of its bottom input. In either case, *select*'s output is the output of the entire circuit, as well as the input to the *increment* component, which simply adds 1 to its input. The output of *increment* is fed into the *delay* component. A delay component outputs whatever was on its input in the previous clock cycle: it "delays" its input by one cycle. However, on the first clock cycle of the simulation there is no previous input, so on the first cycle *delay* outputs whatever is on its *init* input, which is zero in this circuit.

Components

The components used in the resettable counter are trivial examples of the sorts of things provided by the Hawk library, but let's look at a specification of each component in turn.

The simplest component is `constant`:

```
constant :: a -> Signal a
constant val = (λ time . val)
```

The `constant` function takes an input of any type `a`, and returns an output of type `Signal a`, that is, a function from time to a value of type `a`. (Function definition and application in Haskell are denoted by simply placing the function arguments after the function symbol, separated by spaces). The λ symbol in the body of `constant` constructs a function with a single parameter, here called `time`. The return value of a λ-function is the value of the expression that follows the "." symbol. In this case the λ-function ignores its `time` argument and always returns `val`. Thus for every clock cycle, (`constant x`) always has the same value `x`.

The next component is `select`:

```
select :: Signal Bool -> Signal a -> Signal a -> Signal a
select boolSig xSig ySig =
    λ time . if (boolSig time) then
                xSig time
             else (ySig time)
```

The first line declares `select` to be a function. In a Haskell type declaration, anything to the left of an arrow is a function argument. Thus, the `select` function takes one Boolean input signal and two polymorphic input signals, that is, two functions from time to `a`, and returns the λ-function representing the output signal. The function being returned from `select` applies `boolSig` to its `time`

1. We actually implement signals using lazy lists, so that type `Signal a = List a`. This implementation choice will be explained later in the paper.

argument. If (`boolSig time`) equals `True`, then we apply `xSig` to `time` and return the result, otherwise we return the result of applying `ySig` to `time`.

The `increment` component is also quite simple:

```
increment :: Signal Int -> Signal Int
increment xSig = λ time . (xSig time) + 1
```

Given the `xSig` input signal, we return a function (built with a λ) that takes its `time` parameter, applies `xSig` to it, adds one, and returns the result.

The `delay` component is more interesting:

```
delay :: a -> Signal a -> Signal a
delay initVal xSig = λ time . if (time == 0) then
                                initVal
                              else (xSig (time - 1))
```

This function takes an initial value of type `a`, and an input signal of type `Signal a`, and it returns a value of type `Signal a` (the input arguments are in reverse order from the diagram). If `time` is equal to zero, we just return `initVal`, otherwise we return whatever value `xSig` had at clock cycle (`time - 1`). This function can thus propagate values from one clock cycle to the next. Note that the delay function is polymorphic, and can be used to delay signals of any type.

Using the components

Once we have defined primitive signal components like the ones above, we never refer to time values explicitly. This can be seen in the definition of the resettable counter itself:

```
resetCounter :: Signal Bool -> Signal Int
resetCounter reset = output
  where
    output = select reset (constant 0)
                          (delay 0 (increment output))
```

The `resetCounter` definition takes `reset` as a Boolean signal, and returns an integer signal. The `reset` signal is passed into `select`. On every clock cycle where `reset` returns `True`, `select` outputs `0`, otherwise it outputs the result of the `delay` function. On the first clock cycle `delay` outputs `0`, and thereafter outputs the result of whatever (`increment output`) was on the previous clock cycle. The output of the whole circuit is the output of the `select` function, here called `output`. Notice that `output` is used twice in this function: once as the input to `increment`, and once as the result of the entire function. This corresponds to the fact that the `output` wire in Figure 1 is split and used in two places. Whenever a wire is duplicated in this fashion, we must use a `where` statement in Hawk to name the wire.

Recursive definitions

There is something else curious about the `output` variable. It is being used recursively in the same place it is being defined! Most languages only allow such recursion for functions with explicit arguments. In Haskell, one can also define recursive data-structures and functions with implicit arguments, such as the one above. If we didn't have this ability, we would have had to define resetCounter as follows:

```
resetCounter reset = output
  where
    output time = (select reset (constant 0)
```

```
(delay 0 (increment output))) time
```

Every time we have a cycle in a circuit, we have to create a local recursive function, passing an explicit time parameter. This breaks the abstraction of the `Signal` ADT. In fact, in the real implementation of signals, we don't use functions at all. We use infinite lists instead. Each element of the list corresponds to a value at a particular clock cycle; the first list element corresponds to the first clock cycle, the second element to the second clock cycle, and so on. By storing signals as lazy lists, we compute a signal value at a given clock cycle only once, no matter how many times it is subsequently accessed.

Haskell allows recursive definitions of abstract data structures because it is a lazy language, that is, it only computes a part of a data structure when some client code demands its value. It is lazy evaluation that allows Haskell to simulate infinite data structures, such as infinite lists.

A Simple Microprocessor

As we noted in the introduction, the DLX architecture is too complex to explain in fine detail in an introductory report. Thus for pedagogical purposes we show how to use similar techniques to specify a simple microprocessor called SHAM (Simple HAwk Microprocessor). We begin with the simplest possible SHAM architecture (unpipelined), and then add features: pipelining, and a memory-cache.

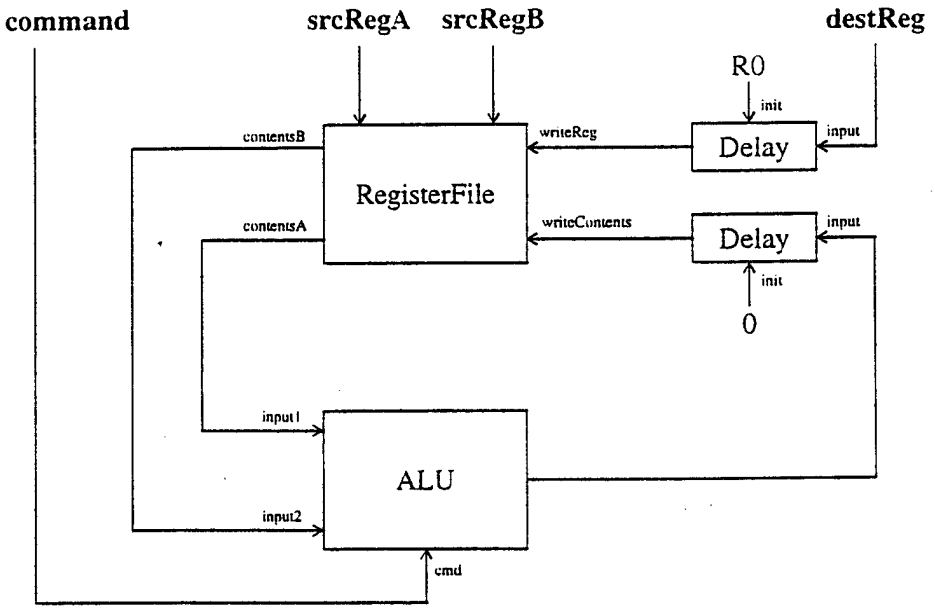


FIGURE 2. Unpipelined version of SHAM.

The unpipelined SHAM diagram is shown in Figure 2. The microprocessor consists of an ALU and a register file. The ALU recognizes three operations: `ADD`, `SUB`, and `INC`. The `ADD` and `SUB` operations add and subtract, respectively, the contents of the two ALU inputs. The `INC` operation causes

the ALU to increment its first input by one and output the result. The register file contains eight integer registers, numbered R0 through R7. Register R0 is hardwired to the value zero, so writes to R0 have no effect. The register file has one write-port and two read-ports. The write-port is a pair of wires; the register to update, called *writeReg*, and the value being written, called *writeContents*. The input to each read-port is a wire carrying a register name. The contents of the named read-port registers are output every cycle along the wires *contentsA* and *contentsB*. If a register is written to and read from during the same clock cycle, the newly written value is reflected in the read-port's output. This is consistent with the behavior of most modern microprocessor register files.

SHAM instructions are provided externally; in our drive for simplicity there is no notion of a program counter. Each instruction consists of an ALU operation, the destination register name, and the two source register names. For each instruction the contents of the two source registers are loaded into the ALU's inputs, and the ALU's result is written back into the destination register.

Unpipelined SHAM Specification

Let us assume we have already specified the register file and ALU, with the signatures below:

```
data Reg = R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7

regFile :: (Signal Reg, Signal Int) ->        (write port inputs)
           (Signal Reg, Signal Reg) ->        (read port inputs)
           (Signal Int, Signal Int)           (read port outputs)

data Cmd = ADD | SUB | INC

alu :: Signal Cmd -> Signal Int -> Signal Int -> Signal Int
```

The `alu` specification takes a command signal and two input signals, and returns a result signal. Given these signatures and the previous definition of `delay`, it is easy in Hawk to specify an unpipelined version of SHAM:

```
sham_1 ::(Signal Cmd,Signal Reg,Signal Reg,Signal Reg) ->
         (Signal Reg,Signal Int)
sham_1 (cmd,destReg,srcRegA,srcRegB) = (destReg',aluOutput')
  where
    (aluInputA,aluInputB) = regFile (destReg',aluOutput')
                                    (srcRegA,srcRegB)
    aluOutput = alu cmd aluInputA aluInputB
    aluOutput' = delay 0 aluOutput
    destReg' = delay R0 destReg
```

The definition of `sham_1` takes a tuple of signals representing the stream of instructions, and returns a pair of signals representing the sequence of register assignments generated by the instructions. The first three lines in the body of `sham_1` read the source register values from the register file and perform the ALU operation. The next two lines delay the destination register name and ALU output, in effect returning the values of the previous clock cycle. The delayed signals become the write-port for the register file. It is necessary to delay the write-port since modifications to the register file logically take effect for the next instruction, not the current one.

Pipelining

Suppose we wanted to increase SHAM's performance by doubling the clock frequency. We will assume that, while `sham_1` could perform both the register file and ALU operations within one clock cycle, with the increased frequency it will take two clock cycles to perform both functions seri-

ally. We use pipelining to increase the overall performance. While the ALU is working on instruction *n*, the register file will be writing the result of instruction *n - 1* back into the appropriate register, and simultaneously reading the source registers of instruction *n + 1*.

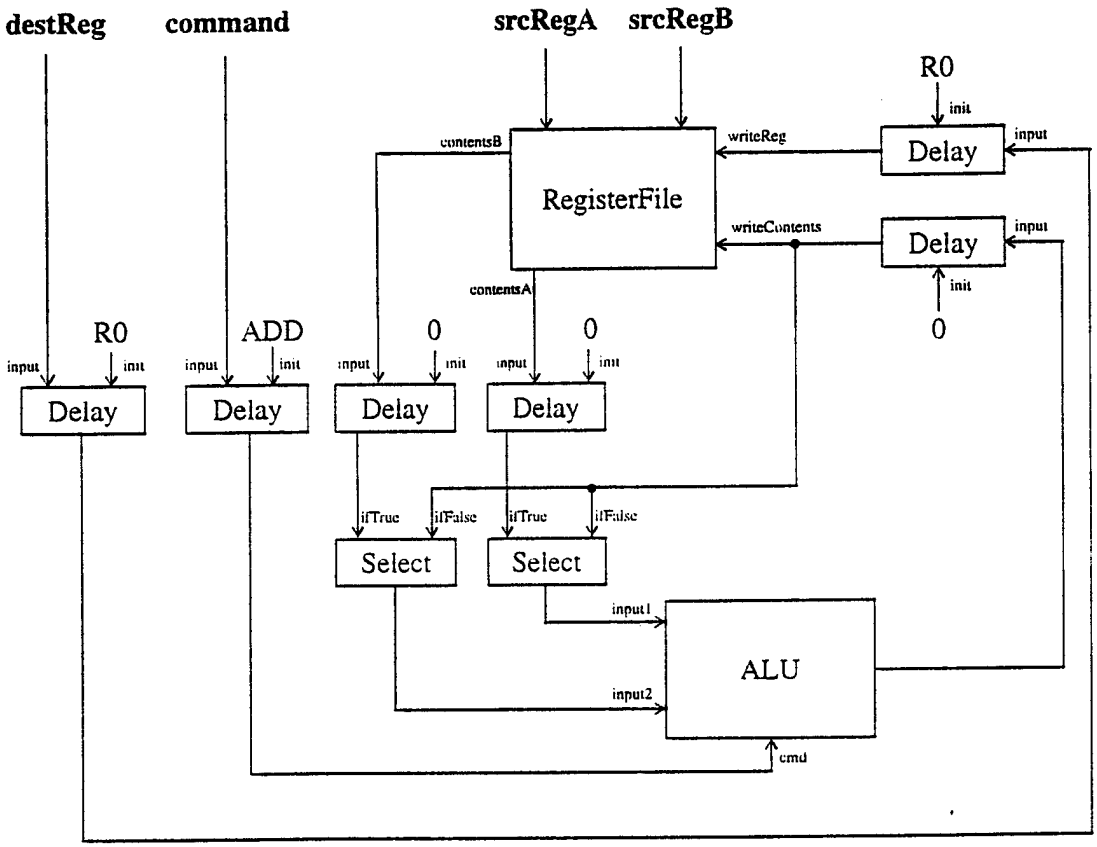


FIGURE 3. Pipelined version of SHAM. Since the register file and the ALU each now take one clock cycle to complete, we have to introduce *pipeline register* delay circuits. The pipeline registers in turn require us to add *Select* circuits to act as bypasses. The logic controlling the *Select* circuits is not shown.

But now consider the following sequence of instructions:

```
R2 ← R1 ADD R3
R4 ← R2 SUB R5
```

When the ADD instruction is in the ALU stage, the SUB instruction is in the register-fetch stage. But one of the registers that is being fetched (R2), has not been written back into the register file yet, because the ALU is still calculating the result. The SUB instruction will read an out-of-date value for R2. This is an example of a *data hazard*, where naive pipelining can produce a result different from

the unpipelined version of a microprocessor. To resolve this hazard, we will first add *bypass logic* to the pipeline, then later abstract away from this added inconvenience.

Figure 3 contains the diagram of a pipelined version of SHAM with bypass logic. By the time the source operands to the SUB instruction (R2 and R5) are ready to be input into the ALU, the up-to-date value for R2 is stored in the delay circuit between the ALU and the register file's write-port. The bypass logic uses this stored value of R2 as the input to the ALU, rather than the out-of-date value read from the register file. The bypass logic examines the incoming instructions to determine when this is necessary. Figure 4 contains the Hawk specification.

```
sham_2 :: (Signal Cmd,Signal Reg,Signal Reg,Signal Reg) ->
          (Signal Reg,Signal Int)
sham_2 (cmd,destReg,srcRegA,srcRegB) = (destReg'',aluOut')
  where
    (registerA,registerB) = regFile (destReg'',aluOut')
                                    (srcRegA,srcRegB)
    registerA' = delay 0 registerA
    registerB' = delay 0 registerB
    destReg' = delay R0 destReg
    cmd' = delay ADD cmd

    aluInputA = select inputValidA registerA' aluOut'
    aluInputB = select inputValidB registerB' aluOut'

    aluOut = alu cmd' aluInputA aluInputB

    aluOut' = delay 0 aluOut
    destReg'' = delay R0 destReg'

    --- Control logic ---

    inputValidA = delay True (noHazard srcRegA)
    inputValidB = delay True (noHazard srcRegB)

    noHazard :: Signal Reg -> Signal Bool
    noHazard srcReg =
    sigOr(sigEqual destReg' (constant R0))
    (sigNotEqual destReg' srcReg)
```

FIGURE 4. A Hawk specification of the pipelined SHAM

The first two lines of the code read the contents of the source registers from the register file. The next four lines delay the source register contents, the ALU command, and the destination register name by one cycle. The two `select` commands decide whether the delayed values should be bypassed. The decision is made by the Boolean signals `inputValidA` and `inputValidB`, which are defined in the control logic section. The next line performs the ALU operation. The last two lines in the data-flow section delay the ALU result and the destination register. The delayed result, called `aluOut'`, is written back into the register file in the register named by `destReg''`, as indicated in the first two lines of the section.

The control logic section determines when to bypass the ALU inputs. The signals `inputValidA` and `inputValidB` are set to `True` whenever the corresponding ALU input is up-to-date. The definition of these signals uses the function `noHazard`, which tests whether the previous instruction's destination register name matches a source register name of the current instruction. If they do, then the function returns `False`. The exception to this is when the destination register is `R0`. In this case the ALU input is always up-to-date, so `noHazard` returns `True`.

Transactions

The definition of `sham_2` highlights a difficulty of many such specifications. Although the data flow section is relatively easy to understand, the control logic section is far from satisfactory. In fact, it takes nearly as many lines of Hawk code to specify the control logic as it does to specify the data flow, and mistakes in the control logic may not be easy to spot. We need a more intuitive way of defining control logic sections in microprocessors.

We use a notion of *transactions* within Hawk to specify the state of an entire instruction as it travels through the microprocessor (similar in spirit to Aagaard and Leeser [AL94]). A transaction holds an instruction's source operand values, the ALU command, and the destination operand value. Transactions also record the register names associated with the source and destination operands:

```
data Transaction = Trans DestOperand Cmd [SrcOperand]

type DestOperand = Operand
type SrcOperand = Operand
type Operand = (Reg,Value)

data Value = Unknown | Val Int
```

An operand is a pair containing a register and its (possibly unknown) value.

For example, the instruction (`R3 ← R2 ADD R1`), when it has completed, would be encoded as shown below (assume that register `R2` holds the value `3`, and `R1` holds `4`):

```
Trans (R3,Val 7) ADD [(R2,Val 3),(R1,Val 4)]
```

This expression states that register `R3` should be assigned the value `7` as a result of adding the contents of register `R2` and `R1`.

Not all of the register values in a transaction are known in the early stages of the pipeline. When a register name does not have an associated value yet, it is assigned the value `Unknown`. For example, if the above instruction had not reached the ALU stage yet, then the corresponding transaction would be:

```
Trans (R3,Unknown) ADD [(R2,(Val 3)),(R1,Val 4))]
```

Figure 5 shows how a transaction's values are filled in as it travels down the pipeline

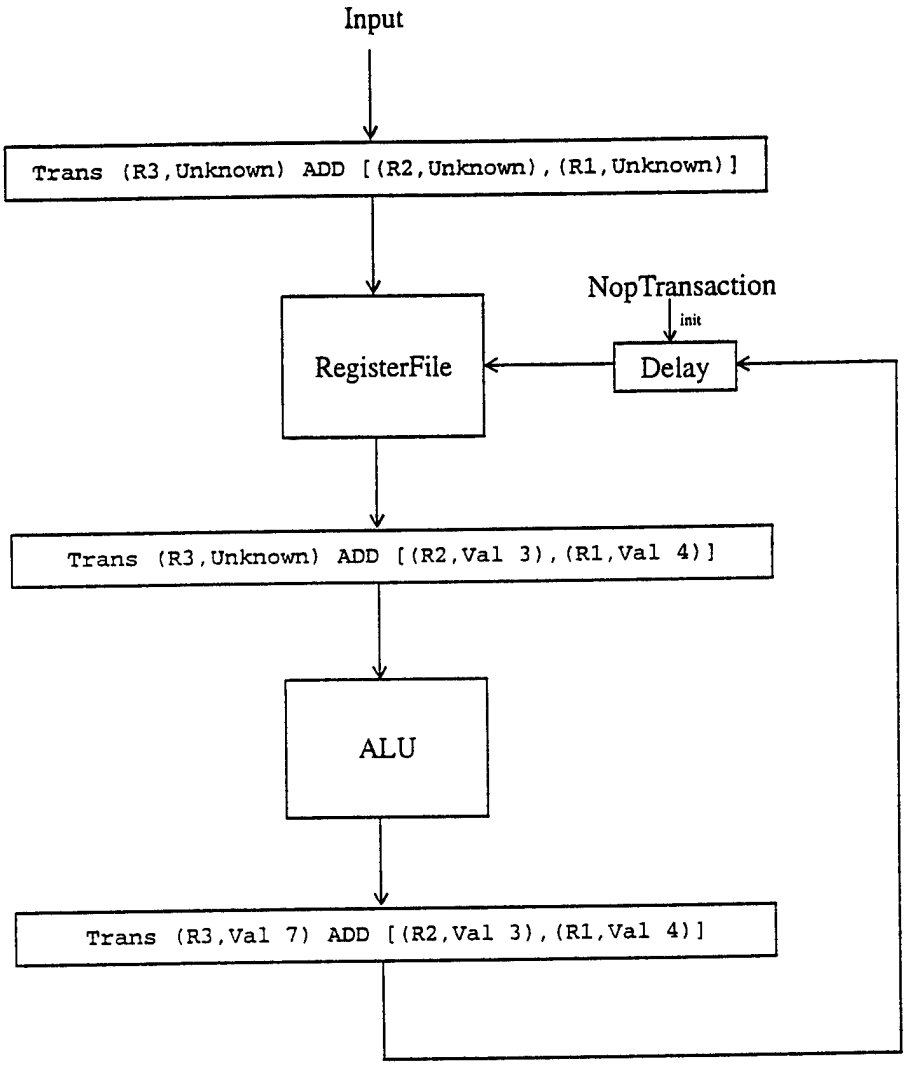


FIGURE 5. A transaction as it flows down the pipeline.

Transaction structure

In general, the `Transaction` datatype contains three subfields. The first field holds the destination register name and its current state. The *state* of a register indicates the current value for the register at a given stage of the pipeline. Possible state values are `Unknown`, or `(Val k)`. The second field is the instruction's ALU operation, in this case the ADD command. The third field holds a list of source operand register names and their corresponding states. In this example, it holds the names and states for the source operands `R2` and `R1`.

The instruction (`R3` ← `R2 ADD R1`), before it enters the SHAM pipeline, is encoded as the transaction:

```
Trans (R3,Unknown) ADD [(R2,Unknown),(R1,Unknown)]
```

At this point, none of the register values are known.

Changes to handle transactions

We change the `regFile` and `alu` functions so that they take and return transactions:

```
regFile :: Signal Transaction ->
           Signal Transaction ->
           Signal Transaction

alu :: Signal Transaction ->
       Signal Transaction
```

Because the register file needs to both write new values to the CPU registers and read values from them, the `regFile` function takes a *write-transaction* and a *read-transaction* as inputs. The function examines the destination register field of the write-transaction and updates the corresponding register in the register file. It outputs the read-transaction, modified so that all of the source register fields contain current values from the register file. For example, suppose `regFile` is applied to the completed write-transaction:

```
Trans (R1,Val 4) INC [(R1,Val 3)]
```

and uses as the read transaction:

```
Trans (R3,Unknown) ADD [(R2,Unknown),(R1,Unknown)]
```

Further, assume that register R1 is assigned 20 and R2 is assigned 3 before `regFile`'s application. Then `regFile` will update R1 to contain 4 from the write-transaction, and will output a new transaction that is identical to the read-transaction, except that all of the source registers have been assigned current values from the register file:

```
Trans (R3,Unknown) ADD [(R2,Val 3),(R1,Val 4)]
```

The revised `alu` function takes a transaction whose source operands have values, performs the appropriate operation, and outputs a modified transaction whose destination field has been filled in. Thus if the ADD transaction above were given to `alu`, it would return:

```
Trans (R3,Val 7) ADD [(R2,Val 3),(R1,Val 4)]
```

Unpipelined SHAM

Using transactions, the unpipelined version of SHAM is even easier to specify than it was before.

```
sham_1_Trans :: Signal Transaction -> Signal Transaction
sham_1_Trans instr = aluOutput'
  where
    aluInput = regFile aluOutput' instr
    aluOutput = alu aluInput
    aluOutput' = delay nop aluOutput

nop = Trans (R0,Val 0) ADD [(R0,Val 0),(R0,Val 0)]
```

But the real benefit of transactions comes from specifying more complex micro-architectures, as we shall see next.

SHAM_2 with Transactions

Transactions are designed to contain the necessary information for concisely specifying control logic. The control logic needs to determine when an instruction's source operand is dependent on

another instruction's destination operand. To calculate the dependency, the source and destination register names must be available. The transaction carries these names for each instruction. Because of this additional information, bypass logic is easily modeled with following combinator:

```
bypass :: Signal Transaction -> Signal Transaction ->
          Signal Transaction
```

The bypass function usually just outputs its first argument. Sometimes, however, the second argument's destination operand name matches one or more of the first argument's source operand names. In this case, the source operand's state values are updated to match the destination operand state value. The updated version of the first argument is then returned.

So if at clock cycle *n* the first argument to bypass is:

```
Trans (R4,Unknown) ADD [(R3,Val 12),(R2,Val 4)]
```

and the second argument at cycle *n* is:

```
Trans (R3,Val 20) SUB [(R8,Val 2),(R11,Val 10)]
```

then because R3 in the second transaction's destination field matches R3 in the first transaction's source field, the output of bypass will be an updated version of the first transaction:

```
Trans (R4,Unknown) ADD [(R3,Val 20),(R2,Val 4)]
```

One special case to bypass's functionality is when a source register is R0. Since R0 is a constant register, it does not get updated. The pipelined version of SHAM with bypass logic is now straightforward. Notice that no explicit control logic is needed, as all the decisions are taken locally in the bypass operations.

```
SHAM_2_Trans :: Signal Transaction -> Signal Transaction
SHAM_2_Trans instr = aluOutput'
  where
    preppedInstr = regFile aluOutput' instr
    preppedInstr' = delay nopTrans preppedInstr
    aluInput = bypass preppedInstr' aluOutput'
    aluOutput = alu aluInput
    aluOutput' = delay nopTrans aluOutput
```

The first line takes instr and fills in its source operand fields from the register file. The filled-in transaction is delayed by one cycle in the second line. In the third line bypass is invoked to ensure that all of the source operands are up-to-date. Finally the transaction result is computed by alu and delayed one cycle so that the destination operand can be written back to the register file.

Hazards

There are some microprocessor hazards that cannot be handled through bypassing. For example, suppose we extended the SHAM architecture to process load and store instructions:

```
R3 ← MEM[R2]
MEM[R5] ← R2
```

The first instruction above is a load instruction; it loads the contents of the address pointed to by R2 into R3. The second instruction is a store; it stores the contents of R2 into the address pointed to by R5. A block diagram of the extended SHAM architecture is shown in Figure 5. There is now a load/

store pipeline stage after the ALU stage. However, this introduces a new problem. Suppose SHAM executes the following two instructions in sequence:

```
R2 ← MEM[R1]
R4 ← R2 ADD R3
```

These two instructions have a data hazard, just as before, but we can not use bypassing to resolve it. Bypassing depends on having a value to bypass at the *beginning* of a clock cycle, but R2's value won't be known until the end of the cycle, after the memory contents have been retrieved from the memory cache. To resolve this hazard, we have to *stall* the pipeline at the register-fetch stage. When the first instruction has reached the end of the ALU stage, the second instruction will have reached the end of the register-fetch stage. At this point the pipeline registers between the register-fetch stage and the ALU stage are overridden; on the next clock cycle they instead output the equivalent of a no-op instruction. The register-fetch stage itself re-reads the second instruction on the next clock cycle. In effect, the pipeline stall inserts a no-op instruction between the two instructions involved in the hazard:

```
R2 ← MEM[R1]
NOP
R4 ← R2 ADD R3
```

Now when the ADD instruction is about to be processed by the ALU, the load instruction has already completed the memory stage. R2's value is held in the pipeline registers after the memory stage, so bypass logic can be used to bring the ALU's input up-to-date. In order to stall correctly, we have to re-read the second instruction. Thus stalling reduces the performance of the pipeline.

Hawk Specification of Extended SHAM

In this section we will give more evidence of the simplifying power of transactions by specifying the extended SHAM architecture. The load/store extension significantly complicates the control logic for the SHAM architecture. We shall see that even though this version of SHAM is still quite simple, the Hawk specification without transactions is much more verbose than the specification with transactions.

To start, we need to define some additional Hawk circuits. The first circuit, `defaultDelay`, augments the normal `delay` circuit so that when a stall hazard is detected, the augmented circuit will output a default value on the next clock cycle, rather than its current input value:

```
defaultDelay :: Signal Bool -> a -> Signal a -> Signal a
defaultDelay emitDefault default input =
delay default (select emitDefault (constant default) input)
```

The `defaultDelay` circuit uses `delay` to store values between clock cycles. The value it stores for the next clock cycle is `default` if `emitDefault` is equal to `True` on the current cycle, otherwise it stores `input`. On the first cycle of the simulation `defaultDelay` always returns `default`.

The `isLoadCmd` circuit returns `True` whenever its argument signal is equal to `LOAD`:

```
isLoadCmd :: Signal Cmd -> Signal Bool
isLoadCmd cmdFunc = (λ time . (cmdFunc time) == LOAD)
```

We also need a function that acts as a Boolean inverter on each clock cycle:

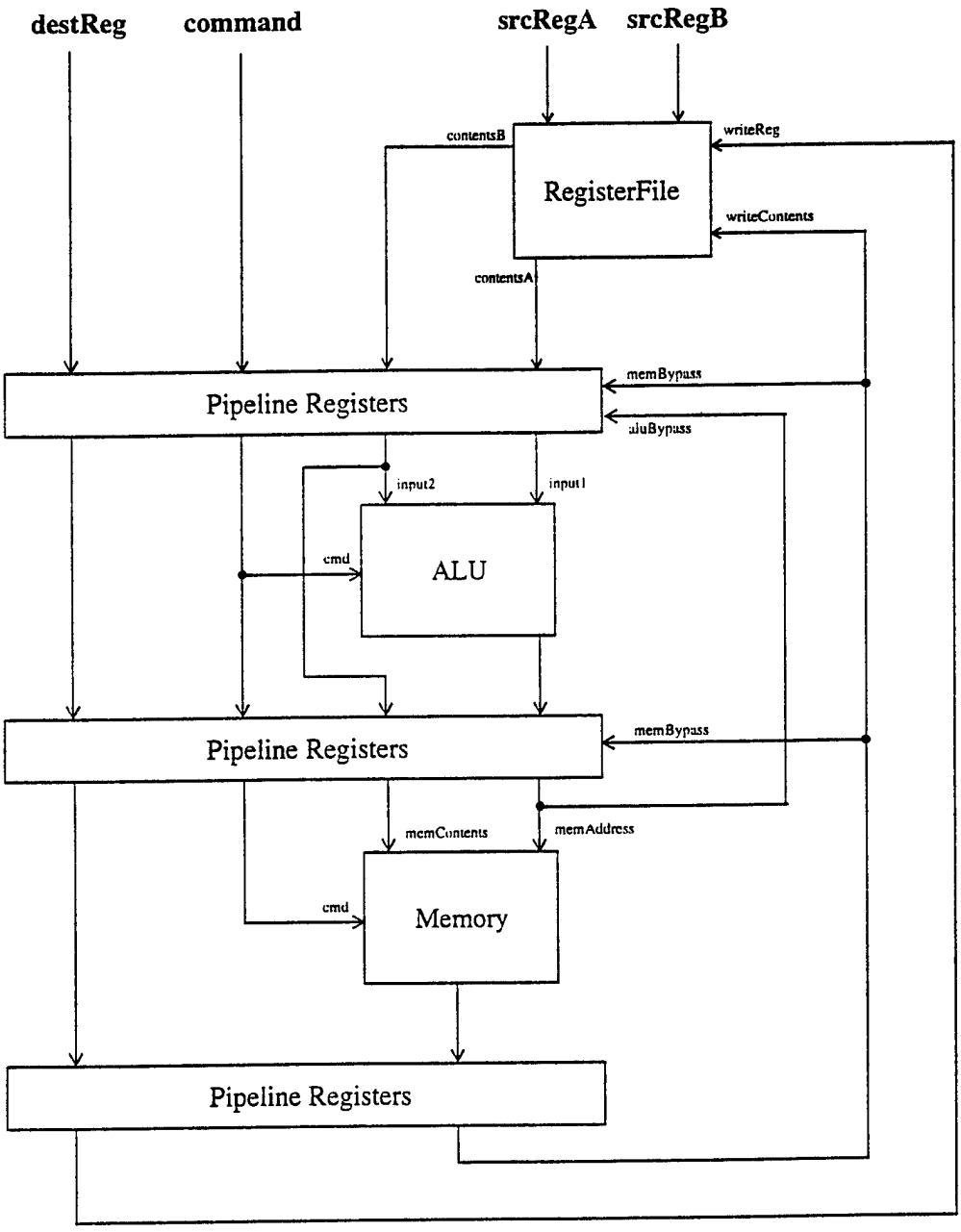


FIGURE 6. Block diagram of extended SHAM pipeline. Each *Pipeline Register* circuit is made up of multiple *Delay* and *Select* circuits. The *Select* circuits are used for bypassing, ensuring that the source operands are up-to-date.

```
sigNot :: Signal Bool -> Signal Bool
sigNot boolFunc = (λ time . not (boolFunc time))
```

Although we previously passed SHAM instructions as parameters, we now need to call a function, `instrCache`, to explicitly retrieve them:

```
instrCache :: Signal Bool -> (Signal Cmd,Signal Reg,Signal Reg,Signal
Reg)
```

Since the pipeline can stall, we need a way to ask for the same instruction two cycles in a row. The `instrCache` function takes a Boolean signal and returns the components of the instruction as separate signals. Whenever the argument signal is `True`, then on the next cycle `instrCache` returns the same instruction as it did for the current clock cycle. Otherwise, it returns the next instruction as normal.

We need to add the commands LOAD and STORE to the Cmd type:

```
data Cmd = ADD | SUB | INC | LOAD | STORE
```

We also need a circuit that actually performs the loads and stores:

```
mem ::Signal Cmd -> Signal Int -> Signal Int -> Signal Int
```

The mem circuit takes a Cmd signal and two `Int` signals, and returns an `Int` signal. On those clock cycles where the Cmd signal is anything but LOAD or STORE, the mem function simply returns the current value of its first `Int` signal. On a LOAD command, mem uses the first `Int` signal as the address to load from, and returns the contents at that memory location. The second argument is ignored. On a STORE command, mem again uses the first `Int` signal as the memory address, and uses the second `Int` signal as the value to store at that memory location. The mem circuit always returns 0 on stores, since the return value is never used.

Finally, we have to extend the definition of `alu` (although not its signature) to simply return its first `Int` signal argument on all clock cycles where its Cmd signal argument is either LOAD or STORE.

Without Transactions

Given the above circuits, we can now define the Hawk specification of the extended SHAM pipeline without transactions. Don't read this specification: just observe how complex it is.

```
SHAM_3 :: (Signal Reg,Signal Int)
SHAM_3 = (destReg'',aluOut')
  where

-- register-fetch stage --
(cmd,destReg,srcRegA,srcRegB) = instrCache loadHazard
(registerA,registerB) = regFile(destReg''',memOut')
(srcRegA,srcRegB)

-- register-fetch stage pipeline registers --
registerA' = defaultDelay loadHazard 0 registerA
registerB' = defaultDelay loadHazard 0 registerB
destReg' = defaultDelay loadHazard R0 destReg
cmd' = defaultDelay loadHazard ADD cmd
```

```
-- ALU stage bypassing --
preInputA = select hazardTwoA' memOut' registerA'
aluInputA = select hazardOneA' aluOut' preInputA
preInputB = select hazardTwoB' memOut' registerB'
aluInputB = select hazardOneB' aluOut' preInputB

-- ALU stage --
aluOut = alu cmd' aluInputA aluInputB

-- ALU stage pipeline registers --
aluInputB' = delay 0 aluInputB
aluOut' = delay 0 aluOut
destReg'' = delay R0 destReg'
cmd'' = delay ADD cmd'

-- memory stage bypassing --
memAddress = select hazardOneA'' memOut' aluOut'
memContents = select hazardOneB'' memOut' aluInputB'

-- memory stage --
memOut = mem cmd'' memAddress memContents

-- memory stage pipeline registers --
memOut' = delay 0 memOut
destReg''' = delay R0 destReg''

--- Control logic ---

-- hazard detection logic --
preHazardOneA = hazard destReg' srcRegA
preHazardOneB = hazard destReg' srcRegB
loadHazard = sigAnd(isLoadCmd cmd')
(sigOr preHazardOneA preHazardOneB)
noLoadHazard = sigNot loadHazard
hazardOneA = sigAnd noLoadHazard preHazardOneA
hazardTwoA = sigAnd noLoadHazard (hazard destReg'' srcRegA)
hazardOneB = sigAnd noLoadHazard preHazardOneB
hazardTwoB = sigAnd noLoadHazard (hazard destReg'' srcRegB)

-- bypass commands for ALU stage inputs --
hazardTwoA' = delay False hazardTwoA
hazardOneA' = delay False hazardOneA
hazardTwoB' = delay False hazardTwoB
hazardOneB' = delay False hazardOneB

-- bypass commands for memory stage inputs --
hazardOneA'' = delay False hazardOneA'
hazardOneB'' = delay False hazardOneB'

hazard ::Signal Reg -> Signal Reg -> Signal Bool
hazard dstReg srcReg =
sigAnd(sigNotEqual dstReg (constant R0))
(sigEqual dstReg srcReg)
```

Even though we have only added one additional pipeline stage, and still have no notion of a program counter or conditional instructions, we can see that the specification is becoming unmanagable. We need to use abstractions like transactions to keep the specifications appropriately compact. In other cases, other abstractions are appropriate. The hardware specification language needs to be powerful enough to enable the designer to invent them at will.

With Transactions

We will see how transactions greatly simplify the specification of extended SHAM. We first have to modify some of the additional Hawk functions to handle transactions:

```
instrCache :: Signal Bool -> Signal Transaction

mem :: Signal Transaction -> Signal Transaction

isLoadTrans :: Signal Transaction -> Signal Bool
```

We also define a new Hawk function, `transHazard`, that returns `True` whenever its two transaction arguments would cause a hazard, if the first transaction preceeded the second transaction in a pipeline:

```
transHazard :: Signal Transaction -> Signal Transaction ->
               Signal Bool
```

The extended Hawk specification using transactions is given below:

```
SHAM_3_Trans :: Signal Transaction
SHAM_3_Trans = memOutput'
  where

-- register-fetch stage --
instr = instrCache loadHazard
preppedInstr = regFile memOutput' instr

-- register-fetch stage pipeline register --
preppedInstr' = defaultDelay loadHazard nopTrans preppedInstr

-- ALU stage bypassing --
aluInput = bypass (bypass preppedIntr' memOutput') aluOutput'

-- ALU stage --
aluOutput = alu aluInput

-- ALU stage pipeline register --
aluOutput' = delay nopTrans aluOutput

-- memory stage bypassing --
memInput = bypass aluOutput' memOutput'

-- memory stage --
memOutput = mem memInput

-- memory stage pipeline register --
memOutput' = delay nopTrans memOutput

----- Control logic -----
```

```
-- hazard detection logic --
loadHazard = sigAnd (isLoadTrans preppedInstr')
(transHazard preppedInstr' preppedInstr)
```

The register-fetch stage retrieves the instruction and fills in its source operands from the register file. The register-fetch pipeline register delays the transaction by one clock cycle, although if there is a load hazard, the register instead outputs a nop-instruction on the next cycle. The ALU stage first updates the source operands of the stored transaction with the results of the two preceding transactions (memOutput' and aluOutput') by invoking bypass twice. It then performs the corresponding ALU operation, if any, on the transaction and stores it in the ALU-stage pipeline register. The memory stage again updates the stored transaction with the immediately preceding transaction, performs any required mem operation, and stores the transaction. The stored transaction is written back to the register file on the next clock cycle. The control logic section determines whether a load hazard exists for the current transaction, that is, whether the immediately preceding transaction was a load instruction that is in hazard with the current transaction.

As we can see, the body of the specification has been reduced from 42 to 13 uncommented source code lines. The overall specification is also much more intuitive. In particular, the control logic section is now only a small minority of the overall specification. We feel the transaction ADT is close to the level of abstraction design engineers use informally when reasoning about microprocessor architectures.

Modelling the DLX

Using techniques comparable to those described in this report we have modeled several DLX architectures:

- An unpipelined version, where each instruction executes in one cycle.
- A pipelined version where all branching instructions cause the pipeline to stall for two cycles.
- A more complex pipelined version where branches cause a one-cycle pipeline stall.
- A pipelined version with branch prediction and speculative execution. Branches are predicted using a one-level branch target buffer. Whenever the guess is correct, the the branch instruction incurs no pipeline stalls. If the guess is incorrect, the pipeline stalls for two cycles.

The microarchitectural specification for the unpipelined DLX is written in a quarter page of uncommented source code; the most complicated pipelined version takes up just over half a page.

Executing the model

We used the Gnu C compiler that generates DLX assembly to test our specifications on several programs. These test cases include a program that calculates the greatest common divisor of two integers, and a recursive procedure that solves the towers of Hanoi puzzle.

We have not made detailed simulation performance measurements yet. Although we plan to test Hawk on several benchmark programs, we do not expect to break simulation-speed records. Hawk is built on top of a lazy functional language, which imposes some performance costs. Transactions also perform some run-time tests that are "compiled-away" in a lower-level pipeline specification. While it would be nice to get high performance, Hawk is primarily a specification language, and only secondarily a simulation tool.

Related Work

There are several research areas that bear a relation on this work, some more closely than others. In particular, modeling specific application domains with Haskell, especially those that depend on time, modeling hardware in various programming languages, and verifying microprocessors with theorem provers. We will pick an example or two from each of these categories. Of course, some of the examples lie in more than one category.

Modeling Specific Domains with Haskell

Haskell compared favorably in an experiment comparing several prototyping languages [HJ94]. The application domain involved modeling the *Geometric Region Server* module, which tracks the regions surrounding ships and planes in a military theatre. The module is required to answer such questions as when an enemy plane will enter a friendly ship's weapons range, or whether a plane has entered a commercial airspace corridor. Experts in each of several languages including Haskell, C++, Awk, and Griffin wrote a prototype program based on the same requirements document. The Haskell solution was considered the most concise and understandable of all the submitted entries. The authors claim their major success factors were: their heavy use of higher-order functions, Haskell's simple syntax, and the availability of powerful list-manipulating primitives in the standard Haskell library.

Haskell has been used to specify several time-varying domains. For instance, RBMH [EH97] is a Haskell library that models interactive multimedia animations. The authors provide ADTs for time-varying behaviors, events, and interactions between behaviors and events. Unlike Hawk, RBMH's model of time is continuous. Also, an RBMH function can examine the values of future events, while Hawk signals only depend on current and past signal values. This non-monotonicity of time in RBMH requires a more sophisticated *time-interval* analysis than is required for Hawk.

Haskell has also been used to directly model hardware circuits at the gate level. O'Donnell [OD95] has developed a Haskell library called Hydra that models gates at several levels of abstraction, ranging from implementations of gates using CMOS and NMOS pass-transistors, up to abstract gate representations using lazy lists to denote time-varying values. Hydra has been used to teach advanced undergraduate courses on computer design, where students use Hydra to eventually design and test a simple microprocessor. Hydra is similar to Hawk in many ways, including the use of higher-order functions and lazy lists to model signals. However, Hydra does not allow users to define *composite* signal types, such as signals of integers or signals of transactions. In Hydra, these composite types have to be built up as tuples or lists of Boolean signals. While this limitation does not cause problems in an introductory computer architecture course, composite signal types significantly reduce specification complexity for more realistic microprocessor specifications.

Modeling Hardware in Programming Languages

There are many other languages for specifying hardware circuits at varying levels of abstraction. The most widely used such languages are Verilog and VHDL. Both of these languages are more general than Hawk in that they can model asynchronous as well as synchronous circuits. However, Verilog and VHDL are large languages with complex semantics, which makes circuit verification more difficult. Also, neither of these languages support polymorphic circuits, nor higher-order circuit combinators, as well as Hawk.

The Ruby language, created by Jones and Sheeran [JS90], is a specification and simulation language based on relations, rather than functions. Ruby is more general than Hawk in that relations can describe more circuits than functions can. On the other hand, existing Ruby simulators require Ruby

relations to be *causal*, i.e. to be implementable as functions. Thus Hawk is equal in expressive power to executable Ruby programs. In addition, much of Ruby's emphasis is on circuit layout. There are combinators to specify where circuits are located in relation to each other and to external wires. Hawk's emphasis is on circuit correctness, so we do not need to address layout issues.

The Voss system [Seg93] lies at a still higher level of abstraction. Voss is a specialized theorem prover with a lazy functional meta-language and support for symbolic Boolean expressions. For example, the expressions $(X \wedge Y)$ and $((X \vee Y) \Rightarrow Z)$ are symbolic Boolean terms. Voss can manipulate such terms directly through an efficient encoding called Ordered Binary Decision Diagrams (OBDDs). Although OBDDs allow for efficient verification of Boolean-valued circuits, it is not yet clear how to generalize them to arbitrary datatypes, which currently reduces their usefulness in Hawk.

Two other languages that are strongly related are HML [LL95] and MHDL. HML is a hardware modelling language based on the functional language ML. It also has higher-order functions and polymorphic types, allowing many of the same abstraction techniques that are used in Hawk, with similar safety guarantees. On the other hand, HML is not lazy, so does not easily allow the recursive circuit specifications that turned out to be key in specifying micro-architectures. The goal of HML is also rather different from Hawk, concentrating on circuits that can be immediately realized by translation to VHDL.

MHDL is a hardware description language for describing analog microwave circuits, and includes an interface to VHDL [Bar95]. Though it tackles a very different part of the hardware design spectrum, like Hawk, MHDL is essentially an extended version of Haskell. The MHDL extensions have to do with physical units on numbers, and universal variables to track frequency and time etc.

Verifying Microprocessos with Theorem Provers

One goal of hardware specification languages is to formally verify circuit correctness. There is much active research related to Hawk on using general-purpose theorem provers to verify microprocessors. Typically the specification language is some form of logic, often either higher-order or temporal. For example, Burch and Dill [BD94] were able to verify the control logic of a simplified version of the DLX. Starting with an untyped functional specification language, they compiled a specification of the DLX's behavior, as well as a description of its architecture, into a pair of state-transition functions using a form of quantifier-free predicate logic, with uninterpreted functions and a notion of equality between them. They were able to verify that the architectural description implements the behavior specification, using a restricted form of theorem proving called model checking.

Windley and Coe [WC94] have used the HOL theorem prover [GM93] to specify a simple pipelined microprocessor called UINTA, which has control and data hazards. UINTA has a five-stage pipeline, with data forwarding, load stall detection, and two branch-delay slots. The authors modeled the architecture as a series of four higher-order logic specifications, each specification in the series being more abstract than the previous one. For each adjacent pair of specifications, they verified that the lower-level specification implemented the functionality of the higher-level specification. They then verified that the top-most specification implemented the functionality of the microprocessor's instruction set. Through a transitivity argument, they were then able to prove that the lowest-level specification, which corresponded to the actual microprocessor circuits, implemented the instruction set functionality.

Eventually we would like to apply verification techniques such as those presented above to verify Hawk designs. We feel that Haskell's strong typing and executability will be helpful in this regard.

For instance, the parametricity theorem is a technique for verifying certain strong properties of polymorphic functions in Haskell, based only on the function's type. Furthermore, because Hawk is executable we can apply symbolic simulation techniques to prove program equivalences. Verification efforts that rely on non-executable or untyped specification languages cannot take advantage of these methods.

Conclusion and Future Directions

We have just completed the specification of a superscalar version of DLX, with speculative and out-of-order instruction execution. The use of transactions has scaled well to this architecture; it turns out that superscalar components like *reservation stations* and *reorder buffers* are naturally expressed as queues of transactions.

Beyond this, we intend to push in a number of directions.

- We hope to use Hawk formally to verify the correctness of microprocessors through the mechanical theorem prover Isabelle [Pau94]. Isabelle is well-suited for Hawk; it has built-in support for manipulating higher-order functions and polymorphic types. It also has well-developed rewriting tactics. Thus simplification strategies like partial evaluation and deforestation [GLP93] can be directly implemented.

  We believe that Hawk is well-suited for formal verification. Often hardware-verification libraries use relations, rather than functions, to specify circuits. Functions have the advantage of being amenable to *equational reasoning*, where terms can be simplified through rewriting without changing their meaning. We hope eventually to take a pipelined specification and through equational rewriting reduce it to an unpipelined version (with of course the extra delay and stall circuits that the pipeline makes necessary).

  We also expect that transactions will aid the verification process. Transactions make explicit much of the pipeline state needed to prove correctness. In lower-level specifications this data has to be inferred from the pipeline context.
- We are working on a visualization tool which will enable the microprocessor designer to inspect values passing along internal wires, at any level in the design hierarchy. We have a probe function with type

  ```
  probe :: String -> Signal a -> Signal a
  ```

  that acts as the identity function on the wire (so has no effect on the simulation) but passes the values of the wire to the outside world, as they are demanded by the simulation. The string parameter allows each probe to be named.
- We have made initial progress on formally extracting stand-alone control logic from the transaction-based models of pipelines. This is potentially a very different approach to proving correctness of control logic from the methods usually repoerted in the literature.
- We intend to stabilize a definition for the Hawk library. We have discovered that as we specify more and more architectures, the same themes keep recurring, and we intend to fix on these patterns.

Haskell as a host language

We have achieved such compact specifications by relying on the powerful abstractions made possible by Haskell. Lazy evaluation and first-class functions allow us to abstract time-varying signals into a single value, which in turn can be defined by recursive definitions. We can abstract away from vectors of wires representing binary numbers or entire instructions by relying on Haskell's type system

to represent these vectors by a single wire. We rely on polymorphism to reuse circuit definitions across multiple signal types.

Once we have stabilized the definition for the Hawk library, we will in effect have defined yet another embedded domain-specific language in Haskell, this one for specifying microprocessors. Haskell has proved to be a very flexible medium for this. Some other examples include animation specification, CGI programming, parser and pretty printer specification, and COM agent scripting.

Acknowledgements

We wish to thank Simon Peyton Jones, Carl Seger, Boris Agapiev, Dick Kieburtz, and Elias Sinderson for their valuable contributions to this research.

References

[AL94] Mark Aagaard and Miriam Leeser. Reasoning about Pipelines with Structural Hazards. In *Proceedings of the Second International Conference on Theorem Provers in Circuit Design.* Bad Herrenalb, Germany, September 1994, pp. 13--32

[Bar95] David Barton, Advanced Modeling Features of MHDL. *Proceedings of International Conference on Electronic Hardware Description Languages (ICEHDL'95)*, Society for Computer Simulation, Jan 95.

[BD94] Jerry Birch and David Dill. Automatic Verification of Pipelined Microprocessor Control. In *Proceedings of the 6th International Conference of Computer Aided Verification.* Stanford, California, June 1994, pp. 68--80

[DLX97] DLX software resources available at:
http://www.mkp.com/books_catalog/ca/hp2e_res.htm#dlx

[EH97] Conal Elliott and Paul Hudak. Functional Reactive Animation. To appear in *Proceedings of the International Conference on Functional Programming 97.* Amsterdam, The Netherlands, June 1997

[GLP93] Andrew Gill, John Launchbury, and Simon Peyton Jones. A Short-cut to Deforestation. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture 93.* Copenhagen, Denmark, 1993

[GM93] Mike Gordon and Tom Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, Cambridge, Great Britain, 1993

[Hawk97] Hawk library and example specifications available at: http://www.cse.ogi.edu/PacSoft/hawk/

[HJ94] Paul Hudak and Mark Jones. Haskell vs. Ada vs. C++ vs. Awk vs. ...
An Experiment in Software Prototyping Productivity. Yale Technical Report YALEU/DCS/RR-1049, October 1994. Available from
ftp://nebula.systemsz.cs.yale.edu/pub/yale-fp/papers/NSWC/

[HP95] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Francisco, 2nd edition, 1995

[HPF96] Paul Hudak, John Peterson, and Joseph Fasel. *A Gentle Introduction to Haskell.* Available from www.haskell.org, December 1996

[Jon91] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, N.J. 1991

[JS90] Geraint Jones and Mary Sheeran. Circuit Design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland, New York, 1990, pp. 13--70.

[LL95] Y. Li and M. Leeser. HML: An Innovative Hardware Design Language and Its Translation to VHDL, *Proceedings CHDL'95*

[Mel88] Thomas Melham. Abstraction Mechanisms for Hardware Verification. In G. Birtwhistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, 1988

[OD95] John O'Donnell. From Transistors to Computer Architecture: Teaching Functional Circuit Specification in Hydra. In *Symposium on Functional Programming Languages in Education*. Springer-Verlag LNCS 1022, 1995, pp. 195--214

[Pau94] Lawrence Paulson, with contributions by Tobias Nipkow. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994

[Pet97] John Peterson, et al. *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language, Version 1.4*. Available at www.haskell.org, April 1997

[Seg93] Carl-Johan Seger. Voss -- A Formal Hardware Verification System, User's Guide. Technical Report 93-45, University of British Columbia, Vancouver, B.C., 1993

[WC94] Philip Windley and Michael Coe. A Correctness Model for Pipelined Microprocessors. In *Proceedings of the Second International Conference on Theorem Provers in Circuit Design*. Bad Herrenalb, Germany, September 1994, pp. 33--51

Due to copyright restrictions, the following article is not included in this collection, but can be found on the web at:

Composing Reactive Animations by Conal Elliott
http://www.research.microsoft.com/research/graphics/elliott/fran/tutorial.htm

Haskore Music Tutorial

Paul Hudak
Yale University
Department of Computer Science
New Haven, CT 06520
hudak@cs.yale.edu

February 14, 1997

1 Introduction

Haskore is a collection of Haskell modules designed for expressing musical structures in the high-level, declarative style of *functional programming*. In Haskore, musical objects consist of primitive notions such as notes and rests, operations to transform musical objects such as transpose and tempo-scaling, and operations to combine musical objects to form more complex ones, such as concurrent and sequential composition. From these simple roots, much richer musical ideas can easily be developed.

Haskore is a means for describing *music*—in particular Western Music—rather than *sound*. It is not a vehicle for synthesizing sound produced by musical instruments, for example, although it does capture the way certain (real or imagined) instruments permit control of dynamics and articulation.

Haskore also defines a notion of *literal performance* through which *observationally equivalent* musical objects can be determined. From this basis many useful properties can be proved, such as commutative, associative, and distributive properties of various operators. An *algebra of music* thus surfaces.

In fact a key aspect of Haskore is that objects represent both *abstract musical ideas* and their *concrete implementations*. This means that when we prove some property about an object, that property is true about the music in the abstract *and* about its implementation. Similarly, transformations that preserve musical meaning also preserve the behavior of their implementations. For this reason Haskell is often called an *executable specification language*; i.e. programs serve the role of mathematical specifications that are directly executable.

Building on the results of the functional programming community's Haskell effort has several important advantages: First, and most obvious, we can avoid the difficulties involved in new programming language design, and at the same time take advantage of the many years of effort that went into the design of Haskell. Second, the resulting system is both *extensible* (the user is free to add new features in substantive, creative ways) and *modifiable* (if the user doesn't like our approach to a particular musical idea, she is free to change it).

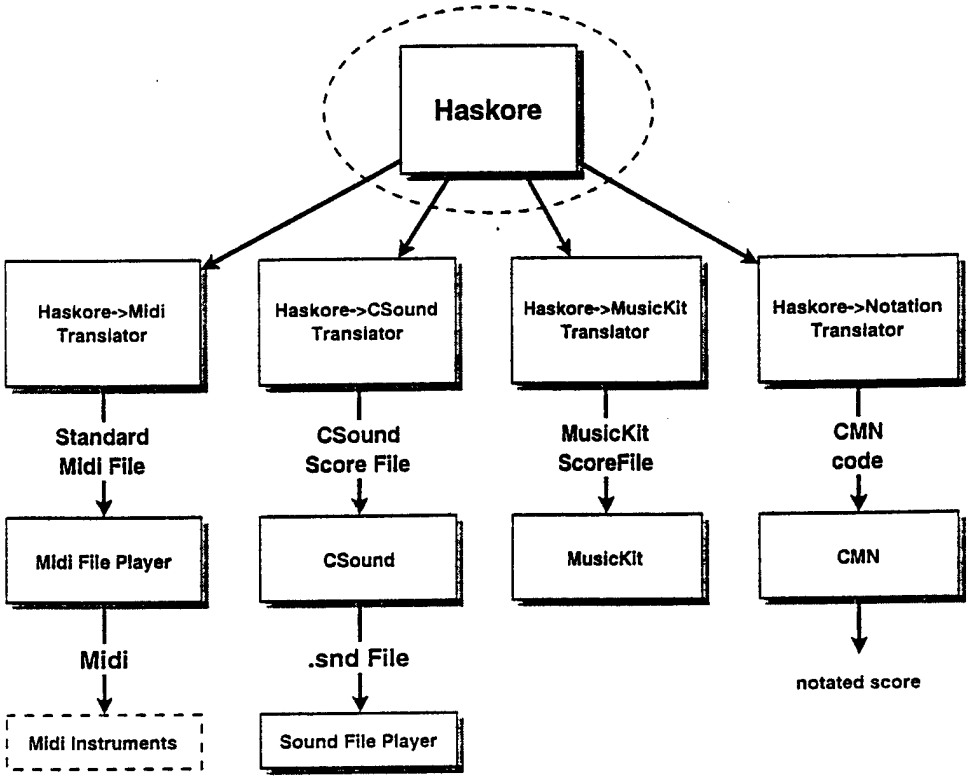


Figure 1: Overall System Diagram

In the remainder of this paper I assume that the reader is familar with the basics of functional programming and Haskell in particular. If not, I encourage reading at least *A Gentle Introduction to Haskell* [HF92] before proceeding. I also assume some familiarity with *equational reasoning*; an excellent introductory text on this is [BW88].

2 The Architecture of Haskore

Figure 1 shows the overall structure of Haskore. Note the degree of independence of high level structures from the "music platform"—it is desirable for Haskore compositions to run equally well as conventional midi-files [IMA90], NeXT MusicKit score files [JB91], and csound score files [Ver86], and to print Haskore compositions in traditional notation using the CMN (Common Music Notation) subsystem. This independence is accomplished by having abstract notions of musical object, player, instrument, and performance that are eventually mapped down to a particular music platform. In this paper I will provide only the details of the mapping to Midi, since it is likely to be the most popular platform for users. In any case, of most interest is the box labeled "Haskore" in the diagram.

At the module level, Haskore is organized as follows:

```
> module Haskore (module Haskore, module Basics, module Performance,
>                 module HaskToMidi)          -- module Players
>         where
>
> import Basics           -- described in Section 3
> import Performance      -- described in Section 4
> -- import Players       -- described in Section 5
> import HaskToMidi       -- described in Section 6
```

As I present various musical ideas in Haskell, I urge the reader to question, at every step, the decisions that I make. There is no supreme theory of music that dictates my decisions, and what I present is actually one of several versions that we have developed (this version is much richer than the one described in [HMGW96]; it is the "Haskore in practice" version alluded to in Section 6 of that paper). I believe this version is suitable for many practical purposes, but the reader may wish to modify it to better satisfy her intuitions and/or application.

This document was written in the *literate programming style*, and thus the LaTeX manuscript file from which it was generated is an *executable Haskell program*. It can be compiled under LaTeX in two ways: a basic mode provides all of the functionality that most users will need, and an extended mode in which various pieces of lower-level code are provided and documented as well (see file header for details). This version was compiled in basic mode. The document can be retrieved via WWW from `ftp://nebula.systemsz.cs.yale.edu/pub/yale-fp/papers/haskore` (consult the README file for details). It is also delivered with the standard joint Nottingham/Yale Hugs release.

The code conforms to Haskell 1.4, although it does not adequately use any of the newer features in Haskell, since most of it was written when Haskell 1.2 was the latest release. Parts of the code should clearly be rewritten to take advantage of some Haskell 1.4 features, in particular "named fields" in datatype declarations. Haskore has been tested under the February, 1997 Nottingham/Yale release of Hugs 1.4, which unfortunately does not yet support mutually recursive modules. For this reason all references to the module `Players` in this document are commented out, which in effect makes it part of module `Performance` (with which it is mutually recursive).

3 The Basics

```
> module Basics where
> infixr 5 :+:, :=:
```

Perhaps the most basic musical idea is that of a *pitch*, which consists of a *pitch class* (i.e. one of 12 semi-tones) and an *octave*:

```
> type Pitch      = (PitchClass, Octave)
> data PitchClass = Cf | C | Cs | Df | D | Ds | Ef | E | Es | Ff | F | Fs
>                 | Gf | G | Gs | Af | A | As | Bf | B | Bs
>      deriving (Eq,Ord,Ix,Show)
> type Octave     = Int
```

So a `Pitch` is a pair consisting of a pitch class and an octave. Octaves are just integers, but we define a datatype for pitch classes, since distinguishing enharmonics (such as G# and Ab) may be important (especially for notation!). By convention, A440 = `(A,4)`.

Musical objects are captured by the `Music` datatype:[1]

```
> data Music = Note Pitch Dur [NoteAttribute]   -- a note \ atomic
>            | Rest Dur                         -- a rest /    objects
>            | Music :+: Music                  -- sequential composition
>            | Music :=: Music                  -- parallel composition
>            | Tempo  Int Int Music             -- scale the tempo
>            | Trans  Int Music                 -- transposition
>            | Instr  IName Music               -- instrument label
>            | Player PName Music               -- player label
>            | Phrase [PhraseAttribute] Music   -- phrase attributes
>     deriving Show
>
> type Dur   = Float                            -- in whole notes
> type IName = String
> type PName = String
```

Here a `Note` is its pitch paired with its duration (in number of whole notes), along with a list of `NoteAttributes` (defined later). A `Rest` also has a duration, but of course no pitch or other attributes.

[1] I prefer to call these "musical objects" rather than "musical values" because the latter may be confused with musical aesthetics.

From these two atomic constructors we can build more complex musical objects using the other constructors, as follows:

- `m1 :+: m2` is the sequential composition of `m1` and `m2`; i.e. `m1` and `m2` are played in sequence.
- `m1 :=: m2` is the parallel composition of `m1` and `m2`; i.e. `m1` and `m2` are played simultaneously.
- `Tempo a b m` scales the rate at which `m` is played (i.e. its tempo) by a factor of `a/b`.
- `Trans i m` transposes `m` by interval `i` (in semitones).
- `Instr iname m` declares that `m` is to be performed using instrument `iname`.
- `Player pname m` declares that `m` is to be performed by player `pname`.
- `Phrase pas m` declares that `m` is to be played using the phrase attributes (described later) in the list `pas`.

It is convenient to represent these ideas in Haskell as a recursive datatype because we wish to not only construct musical objects, but also take them apart, analyze their structure, print them in a structure-preserving way, interpret them for performance purposes, etc.

3.1 Convenient Auxiliary Functions

For convenience we first create a few names for familiar notes, durations, and rests, as shown in Figure 2. Treating pitches as integers is also useful in many settings, so we define some functions for converting between `Pitch` values and `AbsPitch` values (integers). These also are shown in Figure 2, along with a definition of `trans`, which transposes pitches (analogous to `Trans`, which transposes values of type `Music`).

Exercise 1 *Show that* `abspitch . pitch = id`, *and, up to enharmonic equivalences,* `pitch . abspitch = id`.

Exercise 2 *Show that* `trans i (trans j p) = trans (i+j) p`.

3.2 Some Simple Examples

With this modest beginning, we can already express quite a few musical relationships simply and effectively. For example, two common ideas in music are the construction of notes in a horizontal fashion (a *line* or *melody*), and in a vertical fashion (a *chord*):

```
> line, chord :: [Music] -> Music
> line  = foldr (:+:) (Rest 0)
> chord = foldr (:=:) (Rest 0)
```

```
> cf,c,cs,df,d,ds,ef,e,es,ff,f,fs,gf,g,gs,af,a,as,bf,b,bs ::
>    Octave -> Dur -> [NoteAttribute] -> Music
>
> cf o = Note (Cf,o);  c o = Note (C,o);  cs o = Note (Cs,o)
> df o = Note (Df,o);  d o = Note (D,o);  ds o = Note (Ds,o)
> ef o = Note (Ef,o);  e o = Note (E,o);  es o = Note (Es,o)
> ff o = Note (Ff,o);  f o = Note (F,o);  fs o = Note (Fs,o)
> gf o = Note (Gf,o);  g o = Note (G,o);  gs o = Note (Gs,o)
> af o = Note (Af,o);  a o = Note (A,o);  as o = Note (As,o)
> bf o = Note (Bf,o);  b o = Note (B,o);  bs o = Note (Bs,o)
>
> wn,  hn,  qn,  en,  sn,  tn  :: Dur
> wnr, hnr, qnr, enr, snr, tnr :: Music
>
> wn = 1          ; wnr = Rest wn      -- whole note rest
> hn = 1/2        ; hnr = Rest hn      -- half note rest
> qn = 1/4        ; qnr = Rest qn      -- quarter note rest
> en = 1/8        ; enr = Rest en      -- eight note rest
> sn = 1/16       ; snr = Rest sn      -- sixteenth note rest
> tn = 1/32       ; tnr = Rest tn      -- thirty-second note rest
>
> pitchClass :: PitchClass -> Int
>
> pitchClass pc = case pc of
>      Cf -> -1;  C -> 0;  Cs -> 1    -- or should Cf be 11?
>      Df -> 1;   D -> 2;  Ds -> 3
>      Ef -> 3;   E -> 4;  Es -> 5
>      Ff -> 4;   F -> 5;  Fs -> 6
>      Gf -> 6;   G -> 7;  Gs -> 8
>      Af -> 8;   A -> 9;  As -> 10
>      Bf -> 10;  B -> 11; Bs -> 12   -- or should Bs be 0?
>
> type AbsPitch = Int
>
> absPitch :: Pitch -> AbsPitch
> absPitch (pc,oct) = 12*oct + pitchClass pc
>
> pitch    :: AbsPitch -> Pitch
> pitch    ap       = ( [C,Cs,D,Ds,E,F,Fs,G,Gs,A,As,B] !! mod ap 12,
>                       quot ap 12)
>
> trans    :: Int -> Pitch -> Pitch
> trans i p = pitch (absPitch p + i)
```

Figure 2: Convenient note names and pitch conversion functions.

From the notes in the C major triad in register 4, I can now construct a C major arpeggio and chord as well:

```
> cMaj = map (\f->f 4 qn []) [c, e, g]  -- octave 4, quarter notes
>
> cMajArp = line  cMaj
> cMajChd = chord cMaj
```

Suppose now we wish to describe a melody `m` accompanied by an identical voice a perfect 5th higher. In Haskore we simply write "`m :=: Trans 7 m`." Similarly, a canon-like structure involving `m` can be expressed as "`m :=: delay d m`," where:

```
> delay :: Dur -> Music -> Music
> delay d m = Rest d :+: m
```

Of course, Haskell's non-strict semantics also allows us to define infinite musical objects. For example, a musical object may be repeated *ad nauseum* using this simple function:

```
> repeatM :: Music -> Music
> repeatM m = m :+: repeatM m
```

Thus an infinite ostinato can be expressed in this way, and then used in different contexts that extract only the portion that's actually needed.

The notions of inversion, retrograde, retrograde inversion, etc. used in 12-tone theory are also easily captured in Haskore. First let's define a transformation from a line created by `line` to a list:

```
> lineToList :: Music -> [Music]
> lineToList n@(Rest 0) = []
> lineToList (n :+: ns) = n : lineToList ns
>
> retro, invert, retroInvert, invertRetro :: Music -> Music
> retro    = line . reverse . lineToList
> invert m = line (map inv l)
>   where l@(Note r _ _: _)  = lineToList m
>         inv (Note p d nas) = Note (pitch (2*(absPitch r) - absPitch p)) d nas
>         inv (Rest d)       = Rest d
> retroInvert = retro  . invert
> invertRetro = invert . retro
```

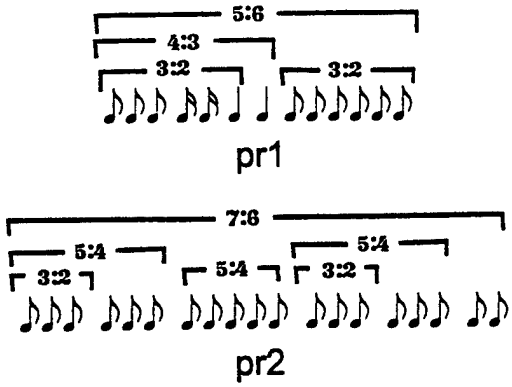


Figure 3: Nested Polyrhythms

Exercise 3 *Show that* "`retro . retro`," "`invert . invert`," *and* "`retroInvert . invertRetro`" *are the identity on values created by* `line`.

For some rhythmical ideas, consider first a simple *triplet* of eighth notes; it can be expressed as "`Tempo 3 2 m`," where `m` is a line of 3 eighth notes. So in fact `Tempo` can be used to create quite complex rhythmical patterns. For example, consider the "nested polyrhythms" shown in Figure 3. They can be expressed quite naturally in Haskore as follows (note the use of the *where* clause in `pr2` to capture recurring phrases):

```
> pr1, pr2 :: Pitch -> Music
> pr1 p = Tempo 5 6 (Tempo 4 3 (mkLn 1 p qn :+:
>                               Tempo 3 2 (mkLn 3 p en :+:
>                                          mkLn 2 p sn :+:
>                                          mkLn 1 p qn    ) :+:
>                               mkLn 1 p qn) :+:
>                   Tempo 3 2 (mkLn 6 p en))
>
> pr2 p = Tempo 7 6 (m1 :+:
>                    Tempo 5 4 (mkLn 5 p en) :+:
>                    m1 :+:
>                    mkLn 2 p en)
>         where m1 = Tempo 5 4 (Tempo 3 2 m2 :+: m2)
>               m2 = mkLn 3 p en
>
> mkLn n p d = line (take n (repeat (Note p d □)))
```

To play polyrhythms `pr1` and `pr2` in parallel using middle C and middle G, respectively, we would do the following (middle C is in the 5th octave):

```
> pr12 :: Music
> pr12 = pr1 (C,5) :=: pr2 (G,5)
```

As a final example in this section, we can can compute the duration in beats of a musical object, a notion we will need in Section 4, as follows:

```
> dur :: Music -> Dur
>
> dur (Note _ d _)  = d
> dur (Rest d)      = d
> dur (m1 :+: m2)   = dur m1   +   dur m2
> dur (m1 :=: m2)   = dur m1 `max` dur m2
> dur (Tempo a b m) = dur m * float b / float a
> dur (Trans  _  m) = dur m
> dur (Instr  _  m) = dur m
> dur (Player _  m) = dur m
> dur (Phrase _  m) = dur m
>
> float = fromInteger . toInteger :: Int -> Float
```

Using dur we can define a function revM that reverses any Music value (and is thus considerably more useful than retro defined earlier). Note the tricky treatment of (:=:).

```
> revM :: Music -> Music
> revM n@(Note _ _ _)  = n
> revM r@(Rest _)      = r
> revM (Tempo i1 i2 m) = Tempo i1 i2 (revM m)
> revM (Trans i  m)    = Trans i     (revM m)
> revM (Instr i  m)    = Instr i     (revM m)
> revM (Phrase pas m)  = Phrase pas  (revM m)
> revM (m1 :+: m2)     = revM m2 :+: revM m1
> revM (m1 :=: m2)     = let d1 = dur m1
>                            d2 = dur m2
>                        in if d1>d2 then revM m1 :=:
>                                         (Rest (d1-d2) :+: revM m2)
>                                   else (Rest (d2-d1) :+: revM m1) :=:
>                                         revM m2
```

Exercise 4 *Find a simple piece of music written by your favorite composer, and transcribe it into Haskore. In doing so, look for repeating patterns, transposed phrases, etc. and reflect this in your code, thus revealing deeper structural aspects of the music than that found in common practice notation.*

Appendix C shows the first 28 bars of Chick Corea's "Children's Song No. 6" encoded in Haskore.

3.3 Phrasing and Articulation

Recall that the `Note` constructor contained a field of `NoteAttributes`. These are values that are attached to notes for the purpose of notation or musical interpretation. Likewise, the `Phrase` constructor permits one to annotate an entire musical object with `PhraseAttributes`. These two attribute datatypes cover a wide range of attributions found in common practice notation, and are shown in Figure 4. Beware that use of them requires the use of a player that knows how to interpret them! Players will be described in more detail in Section 5.

Note that some of the attributes are parameterized with a numeric value. This is used by a player to control the degree to which an articulation is to be applied. For example, we would expect `Legato 1.2` to create more of a legato feel than `Legato 1.1`. The following constants represent default values for some of the parameterized attributes:

```
> legato, staccato  :: Articulation
> accent, bigAccent :: Dynamic
>
> legato    = Legato 1.1
> staccato  = Staccato 0.5
> accent    = Accent 1.2
> bigAccent = Accent 1.5
```

To understand exactly how a player interprets an attribute requires knowing how players are defined. Haskore defines only a few simple players, so in fact many of the attributes in Figure 4 are to allow the user to give appropriate interpretations of them by her particular player. But before looking at the structure of players we will need to look at the notion of a *performance* (these two ideas are tightly linked, which is why the `Players` and `Performance` modules are mutually recursive).

```
> data NoteAttribute = Volume Float        -- by convention: 0=min, 100=max
>                    | Fingering Int
>                    | Dynamics String
>      deriving Show
>
> data PhraseAttribute = Dyn Dynamic
>                      | Art Articulation
>                      | Orn Ornament
>      deriving Show
>
> data Dynamic = Accent Float | Crescendo Float | Diminuendo Float
>              | PPP | PP | P | MP | SF | MF | NF | FF | FFF | Loudness Float
>              | Ritardando Float | Accelerando Float
>      deriving Show
>
> data Articulation = Staccato Float | Legato Float | Slurred Float
>                   | Tenuto | Marcato | Pedal | Fermata | FermataDown | Breath
>                   | DownBow | UpBow | Harmonic | Pizzicato | LeftPizz
>                   | BartokPizz | Swell | Wedge | Thumb | Stopped
>      deriving Show
>
> data Ornament = Trill | Mordent | InvMordent | DoubleMordent
>               | Turn | TrilledTurn | ShortTrill
>               | Arpeggio | ArpeggioUp | ArpeggioDown
>               | Instruction String | Head NoteHead
>      deriving Show
>
> data NoteHead = DiamondHead | SquareHead | XHead | TriangleHead
>               | TremoloHead | SlashHead | ArtHarmonic | NoHead
>      deriving Show
```

Figure 4: Note and Phrase Attributes.

4 Interpretation and Performance

```
> module Performance (module Performance, module Basics) -- module Players
>        where
>
> import Basics
> -- import Players
```

Now that we have defined the structure of musical objects, let us turn to the issue of *performance*, which we define as a temporally ordered sequence of musical *events*:

```
> type Performance = [Event]
>
> data Event = Event Time IName AbsPitch DurT Volume
>      deriving (Eq,Ord,Show)
>
> type Time      = Float
> type DurT      = Float
> type Volume    = Float
```

An event is the lowest of our music representations not yet committed to Midi, csound, or the MusicKit. An event `Event s i p d v` captures the fact that at start time `s`, instrument `i` sounds pitch `p` with volume `v` for a duration `d` (where now duration is measured in seconds, rather than beats).

To generate a complete performance of, i.e. give an interpretation to, a musical object, we must know the time to begin the performance, and the proper volume, key and tempo. We must also know what *players* to use; that is, we need a mapping from the `PNames` in an abstract musical object to the actual players to be used. (We don't yet need a mapping from abstract `INames` to instruments, since this is handled in the translation from a performance into, say, Midi, such as defined in Section 6.)

We can thus model a performer as a function `perform` which maps all of this information and a musical object into a performance:

```
> perform :: PMap -> Context -> Music -> Performance
>
> type PMap    = PName -> Player
> type Context = (Time,Player,IName,DurT,Key,Volume)
> type Key     = AbsPitch
```

```
perform pmap c@(t,pl,i,dt,k,v) m =
  case m of
    Note p d nas -> playNote pl c p d nas
    Rest d       -> □
    m1 :+: m2    -> perform pmap c m1 ++
                    perform pmap (setTime  c (t+(dur m1)*dt)) m2
    m1 :=: m2    -> merge (perform pmap c m1) (perform pmap c m2)
    Tempo  a b m -> perform pmap (setTempo  c (dt * float b / float a)) m
    Trans  p   m -> perform pmap (setTrans  c     (k+p)) m
    Instr  nm  m -> perform pmap (setInstr  c       nm ) m
    Player nm  m -> perform pmap (setPlayer c (pmap nm)) m
    Phrase pas m -> interpPhrase pl pmap c pas m
```

Some things to note:

1. The `Context` is the running "state" of the performance, and gets updated in several different ways. For example, the interpretation of the `Tempo` constructor involves scaling `dt` appropriately and updating the `DurT` field of the context. Figure 5 defines a convenient group of selectors and mutators for contexts and events.

2. Interpretation of notes and phrases is player dependent. Ultimately a single note is played by the `playNote` function, which takes the player as an argument. Similarly, phrase interpretation is also player dependent, reflected in the use of `interpPhrase`. Precisely how these two functions work is described in Section 5.

3. The `DurT` component of the context is the duration, in seconds, of one whole note. To make it easier to compute, we can define a "metronome" function that, given a standard metronome marking (in beats per minute) and the note type associated with one beat (quarter note, eighth note, etc.) generates the duration of one whole note:

```
> metro :: Float -> Dur -> DurT
> metro setting dur = 60 / (setting*dur)
```

   Thus, for example, `metro 96 qn` creates a tempo of 96 quarter notes per minute.

4. In the treatment of `(:+:)`, note that the sub-sequences are appended together, with the start time of the second argument delayed by the duration of the first. The function `dur` (defined in Section 3.2) is used to compute this duration. Note that this results in a quadratic time complexity for `perform`. A more efficient solution is to have `perform` compute the duration directly, returning it as part of its result. This version of `perform` is shown in Figure 6.

5. In contrast, the sub-sequences derived from the arguments to `(:=:)` are merged into a time-ordered stream. The definition of `merge` is given below.

```
setTime, setInstr, setTempo, setTrans, and setVolume
have type:  Context -> X -> Context, where X is obvious.

> setTime   (t,pl,i,dt,k,v) t'  = (t',pl,i,dt,k,v)
> setPlayer (t,pl,i,dt,k,v) pl' = (t,pl',i,dt,k,v)
> setInstr  (t,pl,i,dt,k,v) i'  = (t,pl,i',dt,k,v)
> setTempo  (t,pl,i,dt,k,v) dt' = (t,pl,i,dt',k,v)
> setTrans  (t,pl,i,dt,k,v) k'  = (t,pl,i,dt,k',v)
> setVolume (t,pl,i,dt,k,v) v'  = (t,pl,i,dt,k,v')

getEventTime, getEventInst, getEventPitch, getEventDur, and getEventVol
have type:  Event -> X, where X is obvious

> getEventTime  (Event t _ _ _ _) = t
> getEventInst  (Event _ i _ _ _) = i
> getEventPitch (Event _ _ p _ _) = p
> getEventDur   (Event _ _ _ d _) = d
> getEventVol   (Event _ _ _ _ v) = v

setEventTime, setEventInst, setEventPitch, setEventDur, and setEventVol
have type:  Event -> X -> Event, where X is obvious.

> setEventTime  (Event t i p d v) t' = Event t' i  p  d  v
> setEventInst  (Event t i p d v) i' = Event t  i' p  d  v
> setEventPitch (Event t i p d v) p' = Event t  i  p' d  v
> setEventDur   (Event t i p d v) d' = Event t  i  p  d' v
> setEventVol   (Event t i p d v) v' = Event t  i  p  d  v'
```

Figure 5: Selectors and mutators for contexts and events.

```
> perform pmap c m = fst (perf pmap c m)
>
> perf :: PMap -> Context -> Music -> (Performance, DurT)
> perf pmap c@(t,pl,i,dt,k,v) m =
>   case m of
>     Note p d nas -> (playNote pl c p d nas, d*dt)
>     Rest d       -> ([], d*dt)
>     m1 :+: m2    -> let (pf1,d1) = perf pmap c m1
>                         (pf2,d2) = perf pmap (setTime c (t+d1)) m2
>                     in (pf1++pf2, d1+d2)
>     m1 :=: m2    -> let (pf1,d1) = perf pmap c m1
>                         (pf2,d2) = perf pmap c m2
>                     in (merge pf1 pf2, max d1 d2)
>     Tempo  a b m -> perf pmap (setTempo  c (dt * float b / float a)) m
>     Trans  p m   -> perf pmap (setTrans  c     (k+p)) m
>     Instr  nm m  -> perf pmap (setInstr  c      nm ) m
>     Player nm m  -> perf pmap (setPlayer c (pmap nm)) m
>     Phrase pas m -> interpPhrase pl pmap c pas m
```

Figure 6: The "real" perform function.

```
> merge :: Performance -> Performance -> Performance

  merge a@(e1:es1) b@(e2:es2) =
        if e1 < e2 then e1 : merge es1 b
                   else e2 : merge a es2
  merge [] es2 = es2
  merge es1 [] = es1
```

Note that merge compares entire events rather than just start times. This is to ensure that it is commutative, a desirable condition for some of the proofs used in Section 8. Here is a more efficient version that will work just as well in practice:

```
> merge a@(e1@(Event t1 _ _ _ _) : es1) b@(e2@(Event t2 _ _ _ _) : es2) =
>       if t1 < t2 then e1 : merge es1 b
>                  else e2 : merge a es2
> merge [] es2 = es2
> merge es1 [] = es1
```

5 Players

```
module Players (module Players, module Music, module Performance)
       where

import Music
import Performance
```

In the last section we saw how a performance involved the notion of a *player*. The reason for this is the same as for real players and their instruments: many of the note and phrase attributes (see Section 3.3) are player and instrument dependent. For example, how should "legato" be interpreted in a performance? Or "diminuendo?" Different players interpret things in different ways, of course, but even more fundamental is the fact that a pianist, for example, realizes legato in a way fundamentally different from the way a violinist does, because of differences in their instruments. Similarly, diminuendo on a piano and a harpsichord are different concepts.

With a slight stretch of the imagination, we can even consider a "notator" of a score as a kind of player: exactly how the music is rendered on the written page may be a personal, stylized process. For example, how many, and which staves should be used to notate a particular instrument?

In any case, to handle these issues, Haskore has a notion of a *player* which "knows" about differences with respect to performance and notation. A Haskore player is a 4-tuple consisting of a name and 3 functions: one for interpreting notes, one for phrases, and one for producing a properly notated score.

```
> data Player     = MkPlayer PName NoteFun PhraseFun NotateFun
>
> type NoteFun    = Context -> Pitch -> Dur -> [NoteAttribute] -> Performance
> type PhraseFun = PMap -> Context -> [PhraseAttribute] -> Music -> (Performance,Dur)
> type NotateFun = ()
```

The last line above is temporary for this executable version of Haskore, since notation only works on systems supporting CMN. The real definition should read:

```
type NotateFun = [Glyph] -> Staff
```

Note that both `NoteFun` and `PhraseFun` return a `Performance` (imported from module `Perform`), whereas `NotateFun` returns a `Staff` (imported from module `Notation`).

For convenience we define:

```
> pName          ::  Player -> PName
> pName          (MkPlayer nm _ _ _) = nm
>
> playNote       ::  Player -> NoteFun
> playNote       (MkPlayer _ nf _ _) = nf
>
> interpPhrase   ::  Player -> PhraseFun
> interpPhrase   (MkPlayer _ _ pf _) = pf
>
> notatePlayer   ::  Player -> NotateFun
> notatePlayer   (MkPlayer _ _ _ nf) = nf
```

5.1 Examples of Player Construction

A "default player" called `defPlayer` (not to be confused with "deaf player"!) is defined for use when none other is specified in the score; it also functions as a base from which other players can be derived. `defPlayer` responds only to the `Volume` note attribute and to the `Accent`, `Staccato`, and `Legato` phrase attributes. It is defined in Figure 7. Before reading this code, recall how players are invoked by the `perform` function defined in the last section; in particular, note the calls to `playNote` and `interpPhase` defined above. Then note:

1. `defPlayNote` is the only function (even in the definition of `perform`) that actually generates an event. It also modifies that event based on an interpretation of each note attribute by the function `defHasHandler`.

2. `defNasHandler` only recognizes the `Volume` attribute, which it uses to set the event volume accordingly.

3. `defInterpPhrase` calls (mutually recursively) `perform` to interpret a phrase, and then modifies the result based on an interpretation of each phrase attribute by the function `defPasHandler`.

4. `defPasHandler` only recognizes the `Accent`, `Staccato`, and `Legato` phrase attributes. For each of these it uses the numeric argument as a "scaling" factor of the volume (for `Accent`) and duration (for `Staccato` and `Lagato`). Thus `(Phrase [Legato 1.1] m)` effectively increases the duration of each note in `m` by 10% (without changing the tempo).

It should be clear that much of the code in Figure 7 can be re-used in defining a new player. For example, to define a player `weird` that interprets note attributes just like `defPlayer` but behaves differently with respect to phrase attributes, we could write:

```
weird :: Player
weird  = MkPlayer "Weirdo" (defPlayNote     defNasHandler)
                           (defInterpPhrase myPasHandler )
                           (defNotatePlayer ()           )
```

```
> defPlayer :: Player
> defPlayer  = MkPlayer "Default" (defPlayNote     defNasHandler)
>                                 (defInterpPhrase defPasHandler)
>                                 (defNotatePlayer ()           )
>
> defPlayNote :: (Context->NoteAttribute->Event->Event) -> NoteFun
> defPlayNote nasHandler c@(t,pl,i,dt,k,v) p d nas =
>        [ foldr (nasHandler c)
>                (Event t i (absPitch p + k) (d*dt) v)
>                nas ]
>
> defNasHandler :: Context-> NoteAttribute -> Event -> Event
> defNasHandler (_,_,_,_,_,v) (Volume v') ev = setEventVol ev (v*v'/100.0)
> defNasHandler _             _           ev = ev
>
> defInterpPhrase :: (PhraseAttribute->Performance->Performance) -> PhraseFun
> defInterpPhrase pasHandler pmap c@(t,pl,i,dt,k,v) pas m =
>        let (pf,dur) = perf pmap c m
>        in  (foldr pasHandler pf pas, dur)
>
> defPasHandler :: PhraseAttribute -> Performance -> Performance
> defPasHandler (Dyn (Accent x))     pf =
>        map (\e -> setEventVol e (x * getEventVol e)) pf
> defPasHandler (Art (Staccato x))   pf =
>        map (\e -> setEventDur e (x * getEventDur e)) pf
> defPasHandler (Art (Legato   x))   pf =
>        map (\e -> setEventDur e (x * getEventDur e)) pf
> defPasHandler  _                   pf = pf
>
> defNotatePlayer   :: () -> NotateFun
> defNotatePlayer _  =  ()
```

Figure 7: Definition of default Player defPlayer.

and then supply a suitable definition of `myPasHandler`. That definition could also re-use code, in the following sense: suppose we wish to add an interpretation for `Crescendo`, but otherwise have `myPasHandler` behave just like `defPasHandler`.

```
myPasHandler :: PhraseAttribute -> Performance -> Performance
myPasHandler (Dyn (Crescendo x)) pf = ...
myPasHandler  pa                 pf = defPasHandler pa pf
```

Exercise 5 *Fill in the ... in the definition of* `myPasHandler` *according to the following strategy: Assume* $0 < x < 1$. *Gradually scale the volume of each event by a factor of* 1.0 *through* $1.0 + x$, *using linear interpolation.*

Exercise 6 *Choose some of the other phrase attributes and provide interpretations of them, such as* `Diminuendo`, `Slurred`, `Trill`, *etc.*

In a system that supports it, the default notation handler sets up a staff with a treble clef for the player and appends any glyphs to the end of the staff:

```
defNotatePlayer gs = Staff "Default" 1.0 5 (Clef Treble : gs)
```

Figure 8 defines a relatively sophisticated player called `fancyPlayer` that knows all that `defPlayer` knows, and much more. Note that `Slurred` is different from `Legato` in that it doesn't extend the duration of the *last* note(s). The behavior of (`Ritardando` x) can be explained as follows. We'd like to "stretch" the time of each event by a factor from 0 to x, linearly interpolated based on how far along the musical phrase the event occurs. I.e., given a start time t_0 for the first event in the phrase, total phrase duration D, and event time t, the new event time t' is given by:

$$t' = (1 + \frac{t - t_0}{D}x)(t - t_0) + t_0$$

Further, if d is the duration of the event, then the end of the event $t + d$ gets stretched to a new time t'_d given by:

$$t'_d = (1 + \frac{t + d - t_0}{D}x)(t + d - t_0) + t_0$$

The difference $t'_d - t'$ gives us the new, stretched duration d', which after simplification is:

$$d' = (1 + \frac{2(t - t_0) + d}{D}x)d$$

`Accelerando` behaves in exactly the same way, except that it shortens event times rather than lengthening them. And, a similar but simpler strategy explains the behaviors of `Crescendo` and `Diminuendo`.

```
> fancyPlayer :: Player
> fancyPlayer  = MkPlayer "Fancy" (defPlayNote     defNasHandler )
>                                 fancyInterpPhrase
>                                 (defNotatePlayer ()            )
>
> fancyInterpPhrase :: PhraseFun
> fancyInterpPhrase pmap c                    []      m = perf pmap c m
> fancyInterpPhrase pmap c@(t,pl,i,dt,k,v) (pa:pas) m =
>  let pfd@(pf,dur) = fancyInterpPhrase pmap c pas m
>      loud x       = fancyInterpPhrase pmap c (Dyn (Loudness x) : pas) m
>      stretch x = let t0 = getEventTime (head pf)
>                      r  = x/dur
>                      upd (Event t i p d v) = let dt = t-t0
>                                                  t' = (1+dt*r)*dt + t0
>                                                  d' = (1+(2*dt+d)*r)*d
>                                              in Event t' i p d' v
>                  in (map upd pf, (1+x)*dur)
>      inflate x = let t0 = getEventTime (head pf)
>                      r  = x/dur
>                      upd (Event t i p d v) = let dt = t-t0
>                                              in Event t i p d ((1+dt*r)*v)
>                  in (map upd pf, dur)
>  in case pa of
>     Dyn (Accent x)      -> (map (\e-> setEventVol e (x * getEventVol e)) pf, dur)
>     Dyn PPP -> loud  40  ;  Dyn  PP -> loud  50  ;  Dyn   P -> loud  60
>     Dyn  MP -> loud  70  ;  Dyn  SF -> loud  80  ;  Dyn  MF -> loud  90
>     Dyn  NF -> loud 100  ;  Dyn  FF -> loud 110  ;  Dyn FFF -> loud 120
>     Dyn (Loudness x)    -> fancyInterpPhrase pmap (t,pl,i,dt,k,v*x/100) pas m
>     Dyn (Crescendo x)   -> inflate   x
>     Dyn (Diminuendo x)  -> inflate (-x)
>     Dyn (Ritardando  x) -> stretch   x
>     Dyn (Accelerando x) -> stretch (-x)
>     Art (Staccato x)    -> (map (\e-> setEventDur e (x * getEventDur e)) pf, dur)
>     Art (Legato   x)    -> (map (\e-> setEventDur e (x * getEventDur e)) pf, dur)
>     Art (Slurred  x)    ->
>         let lastStartTime = foldr (\e t -> max (getEventTime e) t) 0 pf
>             setDur e      = if getEventTime e < lastStartTime
>                             then setEventDur e (x * getEventDur e)
>                             else e
>         in (map setDur pf, dur)
>     Art _               -> pfd  -- Remaining articulations:
>                 --   Tenuto | Marcato | Pedal | Fermata  | FermataDown
>                 -- | Breath | DownBow | UpBow | Harmonic | Pizzicato
>                 -- | LeftPizz | BartokPizz | Swell | Wedge | Thumb | Stopped
>     Orn _               -> pfd  -- Remaining ornamenations:
>                 --   Trill | Mordent | InvMordent | DoubleMordent | Turn
>                 -- | TrilledTurn | ShortTrill | Arpeggio | ArpeggioUp
>                 -- | ArpeggioDown | Instruction String | Head NoteHead
>     -- Design Bug: To do these right we need to keep the KEY SIGNATURE
>     -- around so that we can determine, for example, what the trill note is.
>     -- Alternatively, provide an argument to Trill to carry this info.
```


Figure 8: Definition of Player fancyPlayer.

6 Midi

Midi ("musical instrument digital interface") is a standard protocol adopted by most, if not all, manufacturers of electronic instruments. At its core is a protocol for communicating *musical events* (note on, note off, key press, etc.) as well as so-called *meta events* (select synthesizer patch, change volume, etc.). Beyond the logical protocol, the Midi standard also specifies electrical signal characteristics and cabling details. In addition, it specifies what is known as a *standard Midi file* which any Midi-compatible software package should be able to recognize.

Over the years musicians and manufacturers decided that they also wanted a standard way to refer to *common* or *general* instruments such as "acoustic grand piano," "electric piano," "violin," and "acoustic bass," as well as more exotic ones such as "chorus aahs," "voice oohs," "bird tweet," and "helicopter." A simple standard known as *General Midi* was developed to fill this role. It is nothing more than an agreed-upon list of instrument names along with a *program patch number* for each, a parameter in the Midi standard that is used to select a Midi instrument's sound.

Most "sound-blaster"-like sound cards on conventional PC's know about Midi, as well as General Midi. However, the sound generated by such modules, and the sound produced from the typically-scrawny speakers on most PC's, is often poor. It is best to use an outboard keyboard or tone generator, which are attached to a computer via a Midi interface and cables. It is possible to connect several Midi instruments to the same computer, with each assigned a different *channel*. Modern keyboards and tone generators are quite amazing little beasts. Not only is the sound quite good (when played on a good stereo system), but they are also usually *multi-timbral*, which means they are able to generate many different sounds simultaneously, as well as *polyphonic*, meaning that simultaneous instantiations of the same sound are possible.

Note: If you decide to use the General midi features of your sound-card, you need to know about another set of conventions known as "Basic Midi" which is not discussed here. The most important aspect of Basic Midi is that Channel 10 is dedicated to *percussion*. A future release of Haskore should make these distinctions more concrete.

Haskore provides a way to specify a Midi channel number and General Midi instrument selection for each `IName` in a Haskore composition. It also provides a means to generate a Standard Midi File, which can then be played using any conventional Midi software. In this section the top-level code needed by the user to invoke this functionality will be described; the extended document contains all of the gory details.

```
> module HaskToMidi (module HaskToMidi, module GeneralMidi, module MidiFile)
>        where
>
> import Basics
> import Performance
> import MidiFile
> import GeneralMidi
> import List(partition)
> import Char(toLower,toUpper)
```

Instead of converting a Haskore `Performance` directly into a Midi file, Haskore first converts it into a datatype that *represents* a Midi file, which is then written to a file in a separate pass. This separation of concerns makes the structure of the Midi file clearer, makes debugging easier, and provides a natural path for extending Haskore's functionality with direct Midi capability (in fact there is a version of Haskore that does this under Windows '95, but it is not described here).

A `UserPatchMap` is a user-supplied table for mapping instrument names (`IName`'s) to Midi channels and General Midi patch names. The patch names are by default General Midi names, although the user can also provide a `PatchMap` for mapping Patch Names to unconventional Midi Program Change numbers.

```
> type UserPatchMap = [(IName,GenMidiName,MidiChannel)]
```

See Appendix A for an example of a useful user patch map.

Given a `UserPatchMap`, a performance is converted to a datatype representing a Standard Midi File using the `performToMidi` function.

```
> performToMidi :: Performance -> UserPatchMap -> MidiFile
> performToMidi pf pMap =
>         MidiFile mfType (Ticks division)
>           (map (performToMEvs pMap) (splitByInst pf))
```

A table of General Midi assignments called `genMidiMap` is imported from `GeneralMidi` in Appendix E. The Midi file datatype itself and functions for writing it to files are imported from the module `MidiFile`, briefly described below. The remaining details are omitted in the basic version of this document.

```
> module MidiFile where
>
> import Monads(Output, run0, out0)
> import MonadUtils(zeroOrMore, oneOrMore)
> import Utils(unlinesS, rightS, concatS)
> import IOExtensions (readBinaryFile, writeBinaryFile)
```

`OutputMidiFile` is the main function for writing `MidiFile` values to an actual file: its first argument is the filename:

```
> outputMidiFile :: String -> MidiFile -> IO ()
> outputMidiFile fn mf = writeBinaryFile fn (midiFileToString mf)
```

Exercise 7 *Take as many examples as you like from the previous sections, create one or more* `UserPatchMaps`*, write the examples to a file, and play them using a conventional Midi player.*

Appendix A defines some functions which should make the above exercise easier. Appendices B, C, and D contain more extensive examples.

¡title¿ Haskore Tutorial: Chords¡/title¿

7 Representing Chords

I have described how to represent chords as values of type `Music`. However, sometimes it is convenient to treat chords more abstractly. Rather than think of a chord in terms of its actual notes, it is useful to think of it in terms of its chord "quality," coupled with the key it is played in and the particular voicing used. For example, we can describe a chord as being a "major triad in root position, with root middle C." Several approaches have been put forth for representing this information, and we cannot cover all of them here. Rather, I will describe two basic representations, leaving other alternatives to the skill and imagination of the reader.[2]

First, one could use a *pitch* representation, where each note is represented as its distance from some fixed pitch. `0` is the obvious fixed pitch to use, and thus, for example, `[0,4,7]` represents a major triad in root position. The first zero is in some sense redundant, of course, but it serves to remind us that the chord is in "normal form." For example, when forming and transforming chords, we may end up with a representation such as `[2,6,9]`, which is not normalized; its normal form is in fact `[0,4,7]`. Thus we define:

> A chord is in *pitch normal form* if the first pitch is zero. and the subsequent pitches are monotonically increasing.

One could also represent a chord *intervalically*; i.e. as a sequence of intervals. A major triad in root position, for example, would be represented as `[4,3,-7]`, where the last interval "returns" us to the "origin." Like the `0` in the pitch representation, the last interval is redundant, but allows us to define another sense of normal form:

> A chord is in *interval normal form* if the intervals are all greater than zero, except for the last which must be equal to the negation of the sum of the others.

In either case, we can define a chord type as:

```
> type Chord = [AbsPitch]
```

We might ask whether there is some advantage, computationally, of using one of these representations over the other. However, there is an invertible linear transformation between them, as defined by the following functions, and thus there is in fact little advantage of one over the other:

```
> pitToInt :: Chord -> Chord
> pitToInt ch = aux ch
```

[2]For example, Forte prescribes normal forms for chords in an atonal setting [For73].

```
>    where aux (n1:n2:ns) = (n2-n1) : aux (n2:ns)
>          aux [n]        = [head ch - n]
>
> intToPit :: Chord -> Chord
> intToPit ch = 0 : aux 0 ch
>    where aux p [n]    = []
>          aux p (n:ns) = n' : aux n' ns  where n' = p+n
```

Exercise 8 *Show that* `pitToInt` *and* `intToPit` *are* inverses *in the following sense: for any chord* `ch1` *in pitch normal form, and* `ch2` *in interval normal form, each of length at least two:*

```
intToPit (pitToInt ch1) = ch1
pitToInt (intToPit ch2) = ch2
```

Another operation we may wish to perform is a test for *equality* on chords, which can be done at many levels: based only on chord quality, taking inversion into account, absolute equality, etc. Since the above normal forms guarantee a unique representation, equality of chords with respect to chord quality and inversion is simple: it is just the standard (overloaded) equality operator on lists. On the other hand, to measure equality based on chord quality alone, we need to account for the notion of an *inversion.*

Using the pitch representation, the inversion of a chord can be defined as follows:

```
> pitInvert (p1:p2:ps) = 0 : map (subtract p2) ps ++ [12-p2]
```

Although we could also directly define a function to invert a chord given in interval representation, we will simply define it in terms of functions already defined:

```
> intInvert = pitToInt . pitInvert . intToPit
```

We can now determine whether a chord in normal form has the same quality (but possibly different inversion) as another chord in normal form, as follows: simply test whether one chord is equal either to the other chord or to one of its inversions. Since there is only a finite number of inversions, this is well defined. In Haskell:

```
> samePitChord ch1 ch2 =
>  let invs = take (length ch1) (iterate pitInvert ch1)
>  in  or (map (==ch2) invs)
>
```

```
> sameIntChord ch1 ch2 =
>  let invs = take (length ch1) (iterate intInvert ch1)
>  in  or (map (==ch2) invs)
```

For example, `samePitChord [0,4,7] [0,5,9]` returns `True` (since `[0,5,9]` is the pitch normal form for the second inversion of `[0,4,7]`).

8 Equivalence of Literal Performances

A *literal performance* is one in which no aesthetic interpretation is given to a musical object. The function `perform` in fact yields a literal performance; aesthetic nuances must be expressed explicitly using note and phrase attributes.

There are many musical objects whose literal performances we expect to be *equivalent.* For example, the following two musical objects are certainly not equal as data structures, but we would expect their literal performances to be identical:

```
(m1 :+: m2) :+: (m3 :+: m4)
  m1 :+: m2 :+: m3 :+: m4
```

Thus we define a notion of equivalence:

Definition: Two musical objects `m1` and `m2` are *equivalent*, written `m1` ≡ `m2`, if and only if:

```
(∀imap,c)  perform imap c m1 = perform imap c m2
```

where "=" is equality on values (which in Haskell is defined by the underlying equational logic).

One of the most useful things we can do with this notion of equivalence is establish the validity of certain *transformations* on musical objects. A transformation is *valid* if the result of the transformation is equivalent (in the sense defined above) to the original musical object; i.e. it is "meaning preserving."

The most basic of these transformation we treat as *axioms* in an *algebra of music.* For example:

Axiom 1 *For any* `r1`, `r2`, `r3`, `r4`, *and* `m`*:*

```
Tempo r1 r2 (Tempo r3 r4 m)  ≡  Tempo (r1*r3) (r2*r4) m
```

To prove this axiom, we use conventional equational reasoning (for clarity we omit `imap` and simplify the context to just `dt`):

Proof:

```
perform dt (Tempo r1 r2 (Tempo r3 r4 m))
= perform (r2*dt/r1) (Tempo r3 r4 m)        -- unfolding perform
= perform (r4*(r2*dt/r1)/r3) m              -- unfolding perform
= perform ((r2*r4)*dt/(r1*r3)) m            -- simple arithmetic
= perform dt (Tempo (r1*r3) (r2*r4) m)      -- folding perform
```

Here is another useful transformation and its validity proof (for clarity in the proof we omit `imap` and simplify the context to just `(t,dt)`):

Axiom 2 *For any* `r1`, `r2`, `m1`, *and* `m2`*:*

```
Tempo r1 r2 (m1 :+: m2)  ≡  Tempo r1 r2 m1 :+: Tempo r1 r2 m2
```

In other words, *tempo scaling distributes over sequential composition.*

Proof:

```
perform (t,dt) (Tempo r1 r2 (m1 :+: m2))
= perform (t,r2*dt/r1) (m1 :+: m2)                          -- unfolding perform
= perform (t,r2*dt/r1) m1 ++ perform (t',r2*dt/r1) m2       -- unfolding perform
= perform (t,dt) (Tempo r1 r2 m1) ++
          perform (t',dt) (Tempo r1 r2 m2)                  -- folding perform
= perform (t,dt) (Tempo r1 r2 m1) ++
          perform (t'',dt) (Tempo r1 r2 m2)                 -- folding dur
= perform (t,dt) (Tempo r1 r2 m1 :+: Tempo r1 r2 m2)        -- folding perform
where t'  = t + (dur m1)*r2*dt/r1
      t'' = t + (dur (Tempo r1 r2 m1))*dt
```

An even simpler axiom is given by:

Axiom 3 *For any* `r` *and* `m`*:*

```
Tempo r r m  ≡  m
```

In other words, *unit tempo scaling is the identity.*

Proof:

```
perform (t,dt) (Tempo r r m)
= perform (t,r*dt/r) m                          -- unfolding perform
= perform (t,dt) m                              -- simple arithmetic
```

Note that the above proofs, being used to establish axioms, all involve the definition of `perform`. In contrast, we can also establish *theorems* whose proofs involve only the axioms. For example, Axioms 1, 2, and 3 are all needed to prove the following:

Theorem 1 *For any* `r1`, `r2`, `m1`, *and* `m2`*:*

```
Tempo r1 r2 m1 :+: m2  ≡  Tempo r1 r2 (m1 :+: Tempo r2 r1 m2)
```

Proof:

```
Tempo r1 r2 (m1 :+: Tempo r2 r1 m2)
= Tempo r1 r2 m1 :+: Tempo r1 r2 (Tempo r2 r1 m2)     -- by Axiom 1
= Tempo r1 r2 m1 :+: Tempo (r1*r2) (r2*r1) m2         -- by Axiom 2
= Tempo r1 r2 m1 :+: Tempo (r1*r2) (r1*r2) m2         -- simple arithmetic
= Tempo r1 r2 m1 :+: m2                               -- by Axiom 3
```

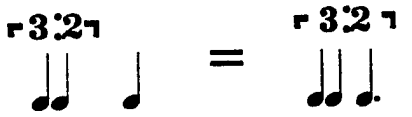


Figure 9: Equivalent Phrases

For example, this fact justifies the equivalence of the two phrases shown in Figure 9.

Many other interesting transformations of Haskore musical objects can be stated and proved correct using equational reasoning. We leave as an exercise for the reader the proof of the following axioms (which include the above axioms as special cases).

Axiom 4 `Tempo` *is* multiplicative *and* `Transpose` *is* additive. *That is, for any* `r1`, `r2`, `r3`, `r4`, `p`, *and* `m`*:*

```
Tempo r1 r2 (Tempo r3 r4 m)  ≡  Tempo (r1*r3) (r2*r4) m
      Trans p1 (Trans p2 m)  ≡  Trans (p1+p2) m
```

Axiom 5 *Function composition is* commutative *with respect to both tempo scaling and transposition. That is, for any* `r1`, `r2`, `r3`, `r4`, `p1` *and* `p2`*:*

```
Tempo r1 r2 . Tempo r3 r4  ≡  Tempo r3 r4 . Tempo r1 r2
     Trans p1 . Trans p2   ≡  Trans p2 . Trans p1
   Tempo r1 r2 . Trans p1  ≡  Trans p1 . Tempo r1 r2
```

Axiom 6 *Tempo scaling and transposition are* distributive *over both sequential and parallel composition. That is, for any* `r1`, `r2`, `p`, `m1`, *and* `m2`*:*

```
Tempo r1 r2 (m1 :+: m2)  ≡  Tempo r1 r2 m1 :+: Tempo r1 r2 m2
Tempo r1 r2 (m1 :=: m2)  ≡  Tempo r1 r2 m1 :=: Tempo r1 r2 m2
     Trans p (m1 :+: m2)  ≡  Trans p m1 :+: Trans p m2
     Trans p (m1 :=: m2)  ≡  Trans p m1 :=: Trans p m2
```

Axiom 7 *Sequential and parallel composition are* associative. *That is, for any* `m0`, `m1`, *and* `m2`*:*

```
m0 :+: (m1 :+: m2)  ≡  (m0 :+: m1) :+: m2
m0 :=: (m1 :=: m2)  ≡  (m0 :=: m1) :=: m2
```

Axiom 8 *Parallel composition is* commutative. *That is, for any* `m0` *and* `m1`*:*

```
m0 :=: m1  ≡  m1 :=: m0
```

Axiom 9 `Rest 0` *is a* unit *for* `Tempo` *and* `Trans`*, and a* zero *for sequential and parallel composition. That is, for any* `r1`*,* `r2`*,* `p`*, and* `m`*:*

```
  Tempo r1 r2 (Rest 0)  ≡  Rest 0
    Trans p (Rest 0)  ≡  Rest 0
m :+: Rest 0  ≡  m  ≡  Rest 0 :+: m
m :=: Rest 0  ≡  m  ≡  Rest 0 :=: m
```

Exercise 9 *Establish the validity of each of the above axioms.*

¡title¿Haskore Tutorial: Related and Future Research¡/title¿

9 Related and Future Research

Many proposals have been put forth for programming languages targeted for computer music composition [Dan89, Sch83, Col84, AK92, DFV92, HS92, CR84, OFLB94], so many in fact that it would be difficult to describe them all here. None of them (perhaps surprisingly) are based on a *pure* functional language, with one exception: the recent work done by Orlarey et al. at GRAME [OFLB94], which uses a pure lambda calculus approach to music description, and bears a strong resemblance to our effort (but unfortunately has not been implemented). There are some other related approaches based on variants of Lisp, most notably Dannenberg's *Fugue* language [DFV92], in which operators similar to ours can be found but where the emphasis is more on instrument synthesis rather than note-oriented composition. Fugue also highlights the utility of lazy evaluation in certain contexts, but extra effort is needed to make this work in Lisp, whereas in a non-strict language such as Haskell it essentially comes "for free." Other efforts based on Lisp utilize Lisp primarily as a convenient vehicle for "embedded language design," and the applicative nature of Lisp is not exploited well (for example, in Common Music the user will find a large number of macros which are difficult if not impossible to use in a functional style).

We are not aware of any computer music language that has been shown to exhibit the kinds of algebraic properties that we have demonstrated for Haskore. Indeed, none of the languages that we have investigated make a useful distinction between music and performance, a property that we find especially attractive about the Haskore design. On the other hand, Balaban describes an abstract notion (apparently not yet a programming language) of "music structure," and provides various operators that look similar to ours [Bal92]. In addition, she describes an operation called *flatten* that resembles our literal interpretation `perform`. It would be interesting to translate her ideas into Haskell; the match would likely be good.

Perhaps surprisingly, the work that we find most closely related to ours is not about music at all: it is Henderson's *functional geometry*, a functional language approach to generating computer graphics [Hen82]. There we find a structure that is in spirit very similar to ours: most importantly, a clear distinction between object *description* and *interpretation* (which in this paper we have been calling musical objects and their performance). A similar structure can be found in Arya's *functional animation* work [Ary94].

There are many interesting avenues to pursue with this research. On the theoretical side, we need a deeper investigation of the algebraic structure of music, and would like to express certain modern theories of music in Haskore. The possibility of expressing other scale types instead of the thus far unstated assumption of standard equal temperament scales is another area of investigation. On the practical side, the potential of a graphical interface to Haskore is appealing. We are also interested in extending the methodology to sound synthesis. Our primary goal currently, however, is to continue using Haskore as a vehicle for interesting algorithmic composition (for example, see [HB95]).

A Convenient Functions for Getting Started With Haskore

```
> module TestHaskore where
> import Haskore
> import System( system )
>
> ------------------------------------------------------------------------
> -- Given a PMap, Context, UserPatchMap, and file name, we can
> -- write a Music value into a midi file:
> ------------------------------------------------------------------------
> mToMF :: PMap -> Context -> UserPatchMap -> String -> Music -> IO ()
> mToMF pmap c upm fn m =
>       let pf = perform pmap c m
>           mf = performToMidi pf upm
>       in outputMidiFile fn mf
>
> ------------------------------------------------------------------------
> -- Convenient default values and test routines
> ------------------------------------------------------------------------
> -- a default UserPatchMap
> -- Note: the PC sound card I'm using is limited to 9 instruments
> defUpm :: UserPatchMap
> defUpm = [("piano","Acoustic Grand Piano",1),
>           ("vibes","Vibraphone",2),
>           ("bass","Acoustic Bass",3),
>           ("flute","Flute",4),
>           ("sax","Tenor Sax",5),
>           ("guitar","Acoustic Guitar (steel)",6),
>           ("violin","Viola",7),
>           ("violins","String Ensemble 1",8),
>           ("drums","Synth Drum",9)]
>
> -- a default PMap that makes everything into a fancyPlayer
> defPMap :: String -> Player
> defPMap pname =
>   MkPlayer pname nf pf sf
>   where MkPlayer _ nf pf sf = fancyPlayer
>
> -- a default Context
> defCon :: Context
> defCon = (0, fancyPlayer, "piano", metro 120 qn, 0, 100)
>
> -- Using the defaults above, from a Music object, we can:
> -- a) generate a performance
> testPerf  :: Music -> Performance
> testPerf m = perform defPMap defCon m
> testPerfDur  :: Music -> (Performance, Dur)
> testPerfDur m = perf defPMap defCon m
>
> -- b) generate a midifile datatype
```

```
> testMidi :: Music -> MidiFile
> testMidi m = performToMidi (testPerf m) defUpm
>
> -- c) generate a midifile
> test     :: Music -> IO ()
> test     m = outputMidiFile "test.mid" (testMidi m)
>
> -- d) generate and play a midifile on Windows, Linux or NeXT
> testWin95, testNT, testLinux, testNext :: Music -> IO ()
> testWin95 m = test m                          >>
>               system "mplayer test.mid" >>
>               return ()
> testNT    m = test m                          >>
>               system "mplay32 test.mid" >>
>               return ()
> testLinux m = test m                               >>
>               system "playmidi -rf test.mid" >>
>               return ()
> testNext  m = test m                        >>
>               system "open test.midi" >>
>               return ()
```

Alternatively, just run "test m" manually, and then invoke the midi player on your system using "play", defined below for NT:

```
> play = system "mplay32 test.mid" >>
>        return ()

> ------------------------------------------------------------------------
> -- Some General Midi test functions (use with caution)
> ------------------------------------------------------------------------
> -- a General Midi user patch map; i.e. one that maps GM instrument names
> -- to themselves, using a channel that is the patch number modulo 16.
> -- This is for use ONLY in the code that follows, o/w channel duplication
> -- is possible, which will screw things up in general.
> gmUpm :: UserPatchMap
> gmUpm = map (\(gmn,n) -> (gmn, gmn, mod n 16 + 1)) genMidiMap
>
> -- Something to play each "instrument group" of 8 GM instruments;
> -- this function will play a C major arpeggio on each instrument.
> gmTest :: Int -> IO()
> gmTest i =  let gMM = take 8 (drop (i*8) genMidiMap)
>                 mu  = line (map simple gMM)
>                 simple (inm,_) = Instr inm cMajArp
>             in  mToMF defPMap defCon gmUpm "test.mid" mu
```

B Examples of Haskore in Action

```
> module HaskoreExamples (module Haskore, module TestHaskore,
>                         module ChildSong6, module SelfSim)
>        where
>
> import Haskore
> import TestHaskore
> import ChildSong6
> import SelfSim

Simple examples of Haskore in action.  Note that this module also
imports modules ChildSong6 and SelfSim.

-------------------------------------------------------------------------

From the tutorial, try things such as pr12, cMajArp, cMajChd, etc. and
try applying inversions, retrogrades, etc. on the same examples.  Also
try "childSong6" imported from module ChildSong.  For example:

> t0 = test (Instr "piano" childSong6)

-------------------------------------------------------------------------

C Major scale for use in examples below:

> cMajScale = Tempo 2 1
>             (line [c 4 en [], d 4 en [], e 4 en [], f 4 en [],
>                    g 4 en [], a 4 en [], b 4 en [], c 5 en []])
>
> cms = cMajScale

Test of various articulations and dynamics:

> t1 = test (Instr "piano"
>        (Phrase [Art (Staccato 0.1)] cms :+:
>         cms                             :+:
>         Phrase [Art (Legato   1.1)] cms    ))
>
> t2 = test (Instr "vibes"
>        (Phrase [Dyn (Diminuendo 0.75)] cms :+:
>         Phrase [Dyn (Crescendo 4.0), Dyn (Loudness 25)] cms))
>
> t3 = test (Instr "flute"
>        (Phrase [Dyn (Accelerando 0.3)] cms :+:
>         Phrase [Dyn (Ritardando  0.6)] cms    ))

-------------------------------------------------------------------------

A function to recursively apply transformations f (to elements in a
```

sequence) and g (to accumulated phrases):

```
> rep :: (Music -> Music) -> (Music -> Music) -> Int -> Music -> Music
> rep f g 0 m = Rest 0
> rep f g n m = m :=: g (rep f g (n-1) (f m))
```

An example using "rep" three times, recursively, to create a "cascade"
of sounds.

```
> run       = rep (Trans 5) (delay tn) 8 (c 4 tn □)
> cascade   = rep (Trans 4) (delay en) 8 run
> cascades  = rep  id       (delay sn) 2 cascade
> t4' x     = test (Instr "piano" x)
> t4        = test (Instr "piano"
>               (cascades :+: revM cascades))
```

What happens if we simply reverse the f and g arguments?

```
> run'      = rep (delay tn) (Trans 5) 4 (c 4 tn □)
> cascade'  = rep (delay en) (Trans 4) 6 run'
> cascades' = rep (delay sn)  id       2 cascade'
> t5        = test (Instr "piano" cascades')
```

Example from the SelfSim module.

```
> t10s   = test (rep (delay durss) (Trans 4) 2 ss)
```

Example from the ChildSong6 module.

```
> cs6 = test childSong6
```

Midi percussion test. Plays all "notes" in a range. (Requires adding
an instrument for percussion to the UserPatchMap.)

```
> drums a b = Instr "drums"
>                 (line (map (\p-> Note (pitch p) sn □) [a..b]))
> t11 a b = test (drums a b)
```

C Partial Encoding of Chick Corea's "Children's Song No. 6"

```
> module ChildSong6 where
> import Haskore
>
> -- Preliminaries: define some dotted durations
> dhn, dqn, den, dsn, dtn :: Float
> dhn = 3/4;  dqn = 3/8;  den = 3/16;  dsn = 3/32;  dtn = 3/64
>
> -- note updaters for mappings
> fd d n = n d v
> vol  n = n   v
> v      = [Volume 80]
> lmap f l = line (map f l)
>
> -- repeat something n times
> times  1    m = m
> times (n+1) m = m :+: (times n m)
>
> -- Baseline:
> b1 = lmap (fd dqn) [b  3, fs 4, g  4, fs 4]
> b2 = lmap (fd dqn) [b  3, es 4, fs 4, es 4]
> b3 = lmap (fd dqn) [as 3, fs 4, g  4, fs 4]
>
> bassLine = times 3 b1 :+: times 2 b2 :+: times 4 b3 :+: times 5 b1
>
> -- Main Voice:
> v1  = v1a :+: v1b
> v1a = lmap (fd en) [a 5, e 5, d 5, fs 5, cs 5, b 4, e 5, b 4]
> v1b = lmap vol    [cs 5 tn, d 5 (qn-tn), cs 5 en, b 4 en]
>
> v2  = v2a :+: v2b :+: v2c :+: v2d :+: v2e :+: v2f
> v2a = lmap vol [cs 5 (dhn+dhn), d 5 dhn,
>                 f 5 hn, gs 5 qn, fs 5 (hn+en), g 5 en]
> v2b = lmap (fd en) [fs 5, e 5, cs 5, as 4] :+: a 4 dqn v :+:
>       lmap (fd en) [as 4, cs 5, fs 5, e 5, fs 5, g 5, as 5]
> v2c = lmap vol [cs 6 (hn+en), d 6 en, cs 6 en, e 5 en] :+: enr :+:
>       lmap vol [as 5 en, a 5 en, g 5 en, d 5 qn, c 5 en, cs 5 en]
> v2d = lmap (fd en) [fs 5, cs 5, e 5, cs 5, a 4, as 4, d 5, e 5, fs 5] :+:
>       lmap vol [fs 5 tn, e 5 (qn-tn), d 5 en, e 5 tn, d 5 (qn-tn),
>                 cs 5 en, d 5 tn, cs 5 (qn-tn), b 4 (en+hn)]
> v2e = lmap vol [cs 5 en, b 4 en, fs 5 en, a 5 en, b 5 (hn+qn), a 5 en,
>                 fs 5 en, e 5 qn, d 5 en, fs 5 en, e 5 hn, d 5 hn, fs 5 qn]
> v2f = Tempo 3 2 (lmap vol [cs 5 en, d 5 en, cs 5 en]) :+: b 4 (3*dhn+hn) v
>
> mainVoice = times 3 v1 :+: v2
>
> -- Putting it all together:
> childSong6 = Instr "piano" (Tempo 3 1 (Phrase [Dyn SF] bassLine :=: mainVoice))
```

D Example of Simple Self-Similar (Fractal) Music

```
> module SelfSim where
>
> import Haskore
> import TestHaskore

An example of self-similar, or fractal, music.

> data Cluster = Cl SNote [Cluster]  -- this is called a Rose tree
> type Pat     = [SNote]
> type SNote   = [(AbsPitch,Dur)]    -- i.e. a chord
>
> sim :: Pat -> [Cluster]
> sim pat = map mkCluster pat
>     where mkCluster notes = Cl notes (map (mkCluster . addmult notes) pat)
>
> addmult pds iss = zipWith addmult' pds iss
>                   where addmult' (p,d) (i,s) = (p+i,d*s)
>
> simFringe n pat = fringe n (Cl [(0,0)] (sim pat))
>
> fringe 0 (Cl note cls) = [note]
> fringe n (Cl note cls) = concat (map (fringe (n-1)) cls)
>
> -- this just converts the result to Haskore:
> simToHask s = let mkNote (p,d) = Note (pitch p) d []
>               in line (map (chord . map mkNote) s)
>
> -- and here are some examples of it being applied:
>
> sim1 n = Instr "bass"
>            (Trans 36
>               (Tempo 4 1 (simToHask (simFringe n pat1))))
> t6 = test (sim1 4)
>
> sim2 n = Instr "piano"
>            (Trans 53
>               (Tempo 4 1 (simToHask (simFringe n pat2))))
> t7 = test (sim2 4)
>
> sim12 n = sim1 n :=: sim2 n
> t8 = test (sim12 4)
>
> sim3 n = Instr "vibes"
>            (Trans 48
>               (Tempo 4 1 (simToHask (simFringe n pat3))))
> t9 = test (sim3 3)
>
> sim4 n  = (Trans 60
```

```
>               (Tempo 2 1 (simToHask (simFringe n pat4'))))
>
> sim4s n = let s = sim4 n
>               l1 = Instr "flute" s
>               l2 = Instr "bass"  (Trans (-36) (revM s))
>           in l1 :=: l2
>
> ss     = sim4s 3
> durss  = dur ss
>
> t10    = test ss
>
> pat1,pat2,pat3,pat4,pat4' :: [SNote]
> pat1 = [[(0,1.0)],[(4,0.5)],[(7,1.0)],[(5,0.5)]]
> pat2 = [[(0,0.5)],[(4,1.0)],[(7,0.5)],[(5,1.0)]]
> pat3 = [[(2,0.6)],[(5,1.3)],[(0,1.0)],[(7,0.9)]]
> pat4' = [[(3,0.5)],[(4,0.25)],[(0,0.25)],[(6,1.0)]]
> pat4 = [[(3,0.5),(8,0.5),(22,0.5)],[(4,0.25),(7,0.25),(21,0.25)],
>         [(0,0.25),(5,0.25),(15,0.25)],[(6,1.0),(9,1.0),(19,1.0)]]
```

E General Midi

```
> module GeneralMidi where
>
> import MidiFile
>
> type GenMidiName = String
> type GenMidiTable = [(GenMidiName,ProgNum)]
>
> genMidiMap :: GenMidiTable
> genMidiMap =[
>  ("Acoustic Grand Piano",0),       ("Bright Acoustic Piano",1),
>  ("Electric Grand Piano",2),       ("Honky Tonk Piano",3),
>  ("Rhodes Piano",4),               ("Chorused Piano",5),
>  ("Harpsichord",6),                ("Clavinet",7),
>  ("Celesta",8),                    ("Glockenspeil",9),
>  ("Music Box",10),                 ("Vibraphone",11),
>  ("Marimba",12),                   ("Xylophone",13),
>  ("Tubular Bells",14),             ("Dulcimer",15),
>  ("Hammond Organ",16),             ("Percussive Organ",17),
>  ("Rock Organ",18),                ("Church Organ",19),
>  ("Reed Organ",20),                ("Accordion",21),
>  ("Harmonica",22),                 ("Tango Accordion",23),
>  ("Acoustic Guitar (nylon)",24),   ("Acoustic Guitar (steel)",25),
>  ("Electric Guitar (jazz)",26),    ("Electric Guitar (clean)",27),
>  ("Electric Guitar (muted)",28),   ("Overdriven Guitar",29),
>  ("Distortion Guitar",30),         ("Guitar Harmonics",31),
>  ("Acoustic Bass",32),             ("Electric Bass (fingered)",33),
>  ("Electric Bass (picked)",34),    ("Fretless Bass",35),
>  ("Slap Bass 1",36),               ("Slap Bass 2",37),
>  ("Synth Bass 1",38),              ("Synth Bass 2",39),
>  ("Violin",40),                    ("Viola",41),
>  ("Cello",42),                     ("Contrabass",43),
>  ("Tremolo Strings",44),           ("Pizzicato Strings",45),
>  ("Orchestral Harp",46),           ("Timpani",47),
>  ("String Ensemble 1",48),         ("String Ensemble 2",49),
>  ("Synth Strings 1",50),           ("Synth Strings 2",51),
>  ("Choir Aahs",52),                ("Voice Oohs",53),
>  ("Synth Voice",54),               ("Orchestra Hit",55),
>  ("Trumpet",56),                   ("Trombone",57),
>  ("Tuba",58),                      ("Muted Trumpet",59),
>  ("French Horn",60),               ("Brass Section",61),
>  ("Synth Brass 1",62),             ("Synth Brass 2",63),
>  ("Soprano Sax",64),               ("Alto Sax",65),
```

```
> ("Tenor Sax",66),
> ("Oboe",68),
> ("English Horn",70),
> ("Piccolo",72),
> ("Recorder",74),
> ("Blown Bottle",76),
> ("Whistle",78),
> ("Lead 1 (square)",80),
> ("Lead 3 (calliope)",82),
> ("Lead 5 (charang)",84),
> ("Lead 7 (fifths)",86),
> ("Pad 1 (new age)",88),
> ("Pad 3 (polysynth)",90),
> ("Pad 5 (bowed)",92),
> ("Pad 7 (halo)",94),
> ("FX1 (train)",96),
> ("FX3 (crystal)",98),
> ("FX5 (brightness)",100),
> ("FX7 (echoes)",102),
> ("Sitar",104),
> ("Shamisen",106),
> ("Kalimba",108),
> ("Fiddle",110),
> ("Tinkle Bell",112),
> ("Steel Drums",114),
> ("Taiko Drum",116),
> ("Synth Drum",118),
> ("Guitar Fret Noise",120),
> ("Seashore",122),
> ("Telephone Ring",124),
> ("Applause",126),
("Barinote Sax",67),
("Basoon",69),
("Clarinet",71),
("Flute",73),
("Pan Flute",75),
("Shakuhachi",77),
("Occarina",79),
("Lead 2 (sawtooth)",81),
("Lead 4 (chiff)",83),
("Lead 6 (voice)",85),
("Lead 8 (bass+lead)",87),
("Pad 2 (warm)",89),
("Pad 4 (choir)",91),
("Pad 6 (metallic)",93),
("Pad 8 (sweep)",95),
("FX2 (soundtrack)",97),
("FX4 (atmosphere)",99),
("FX6 (goblins)",101),
("FX8 (sci-fi)",103),
("Banjo",105),
("Koto",107),
("Bagpipe",109),
("Shanai",111),
("Agogo",113),
("Woodblock",115),
("Melodic Drum",117),
("Reverse Cymbal",119),
("Breath Noise",121),
("Bird Tweet",123),
("Helicopter",125),
("Gunshot",127)]
```

References

[AK92] D.P. Anderson and R. Kuivila. Formula: A programming language for expressive computer music. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[Ary94] K. Arya. A functional animation starter-kit. *Journal of Functional Programming*, 4(1):1–18, 1994.

[Bal92] M. Balaban. Music structures: Interleaving the temporal and hierarchical aspects of music. In M. Balaban, K. Ebcioglu, and O. Laske, editors, *Understanding Music With AI*, pages 110–139. AAAI Press, 1992.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, New York, 1988.

[Col84] D. Collinge. Moxie: A languge for computer music performance. In *Proc. Int'l Computer Music Conference*, pages 217–220. Computer Music Association, 1984.

[CR84] P. Cointe and X. Rodet. Formes: an object and time oriented system for music composition and synthesis. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programmming*, pages 85–95. ACM, 1984.

[Dan89] R.B. Dannenberg. The Canon score language. *Computer Music Journal*, 13(1):47–56, 1989.

[DFV92] R.B. Dannenberg, C.L. Fraley, and P. Velikonja. A functional language for sound synthesis with behavioral abstraction and lazy evaluation. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[For73] A. Forte. *The Structure of Atonal Music*. Yale University Press, New Haven, CT, 1973.

[HB95] P. Hudak and J. Berger. A model of performance, interaction, and improvisation. In *Proceedings of International Computer Music Conference*. Int'l Computer Music Association, 1995.

[Hen82] P. Henderson. Functional geometry. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programmming*. ACM, 1982.

[HF92] P. Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[HMGW96] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3), June 1996. available via `ftp://nebula.systemsz.cs.yale.edu/pub/yale-fp/papers/haskore/hmn-lhs.ps`.

[HS92] G. Haus and A. Sametti. Scoresynth: A system for the synthesis of music scores based on petri nets and a music algebra. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[IMA90] Midi 1.0 detailed specification: Document version 4.1.1, February 1990.

[JB91] D. Jaffe and L. Boynton. An overview of the sound and music kits for the NeXT computer. In S.T. Pope, editor, *The Well-Tempered Object*, pages 107–118. MIT Press, 1991.

[OFLB94] O. Orlarey, D. Fober, S. Letz, and M. Bilton. Lambda calculus and music calculi. In *Proceedings of International Computer Music Conference.* Int'l Computer Music Association, 1994.

[Sch83] B. Schottstaedt. Pla: A composer's idea of a language. *Computer Music Journal.* 7(1):11–20, 1983.

[Ver86] B. Vercoe. Csound: A manual for the audio processing system and supporting programs. Technical report, MIT Media Lab, 1986.

Perl for Swine: CGI Programming in Haskell

Erik Meijer
OGI and Utrecht University

Joost van Dijk
Utrecht University

Draft Lecture Notes

1 Introduction

Many documents on the web are *static*, i.e. they are the same each time they are returned by the server in response to a request by a client. On the other hand, *dynamic* documents are generated by running a *CGI script* on the server when the document is requested; the generated documents can then depend on the incoming request from the client.

Any programming language that can read from the standard input, write to the standard output, and access the environment variables is suited for writing CGI scripts. Most CGI scripts are written in languages like C or PERL. The latter language is popular because it has rich features for manipulating text (regular expressions).

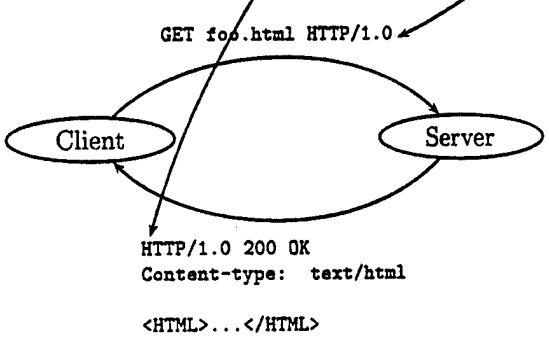
This paper describes a library for writing CGI scripts in Haskell. Input and output coding of CGI scripts is handled by a *wrapper*, a higher-order function that decodes the incoming request, passes it to the *worker*, and encodes the response that the worker produces. Thus the main part of the application, the worker function inside the wrapper, need not be at all concerned with the idiosyncrasies of the CGI protocol.

Unlike most CGI scripts written in Perl or C, our scripts build an explicit representation (a model) of the generated content, which is printed in the right form (a presentation) by the wrapper. For example, HTML is represented by a simple tree type. A set of combinators is provided from which complicated HTML pages can be assembled easily.

Many people ask "why bother writing CGI scripts now that we have Java, ActiveX, VBScript and JScript?" The answer is that both serve complementary, but equally useful purposes. CGI scripts perform server-side computations, while the others do client-side computations.

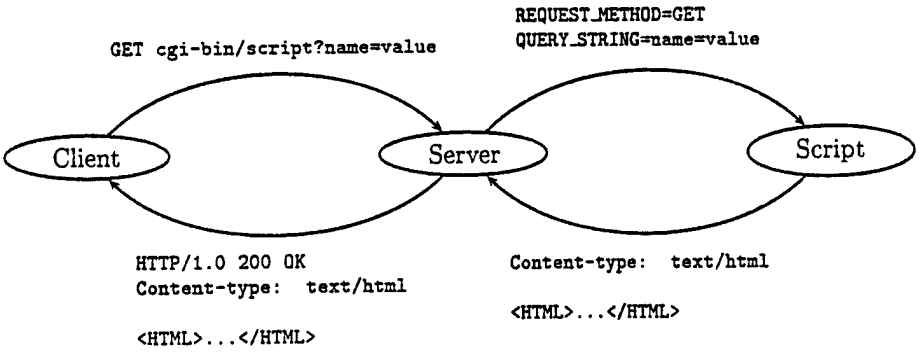
2 The World Wide Web

The World Wide Web is based on the *client-server* model of computation. In this model a *server* provides resources to potentially many *clients*. Server and clients communicate via a *protocol*. The client sends a *request* to the server, to which the server replies by sending a *response* back to the client.



Once such a cycle is completed, the client and server are no longer in contact; the HTTP protocol is *stateless*.

A *static* document is the same every time it is returned by the server. A *dynamic* document is generated by running a script on the server when the document is requested; the generated documents can then depend on the incoming request from the client. Moreover, the script can change the state of the server as a side-effect of the interaction.



The communication interface between the server and content-generating script is called the *Common Gateway Interface* (CGI). The server forwards the request from the client to the script via specific environment variables and the standard input. The script returns the response for the client to the server via the standard output in a format specified by the CGI standard.

The goal of this paper is to make programming of server-side scripts as simple as possible (section 4). But first we explain how to access (dynamic) documents from a client (section 3.1) and how to make the server distinguish between static and dynamic documents (section 3.2).

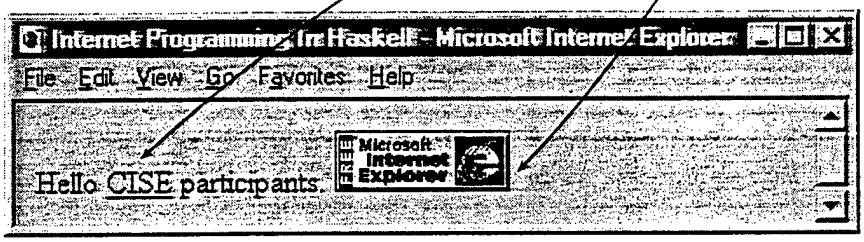
3 Computerized HTTP clients

Communicating with an HTTP server is rather tedious and best left to a machine. A web browser such the *Microsoft Internet Explorer* does exactly this. It takes an HTML document with embedded requests, or *hyperlinks*, and renders it on the screen. HTML is a domain specific language for programming the client side of the HTTP protocol, and a browser is an interpreter for this language.

For example the HTML document

```
<HTML>
 <HEAD>
  <TITLE>Internet Programming In Haskell</TITLE>
 </HEAD>
 <BODY>
 Hello <A HREF="http://www.evergreen.edu/user/CISE/">CISE</A> participants.
 <IMG SRC="http://tibet.cse.ogi.edu/Personal/images/ie_animated.gif">
 </BODY>
</HTML>
```

appears on the user's screen as



Request for images are evaluated eagerly and displayed inline. Requests for other pages are evaluated lazily, i.e. only when the user clicks on them. When this happens, the browser sets up a connection with the server `www.evergreen.edu` and the request `GET user/CISE/ HTTP/1.0`. Eventually, the browser replaces the current document by the document contained in the server's response.

3.1 Accessing active documents

The client sees no difference between hyperlinks to static or dynamic documents. You refer to cgi-scripts just as to any other URL. For example

```
http://www.cse.ogi.edu/~erik/cgi-bin/helloHTML.cgi
```

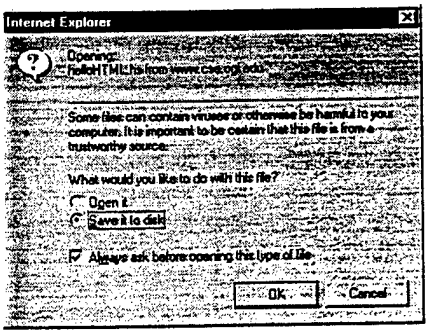
is a link to the simple "Hello World!" script of section 8.

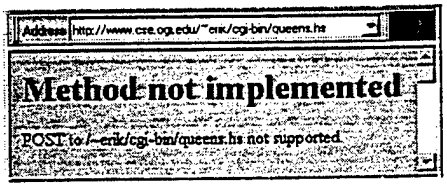
3.2 How the server tells static and dynamic documents apart

But how does the *server* know that it has to execute the script and not return the content of the file `helloHTML.cgi` as it does for a normal `.html` file?

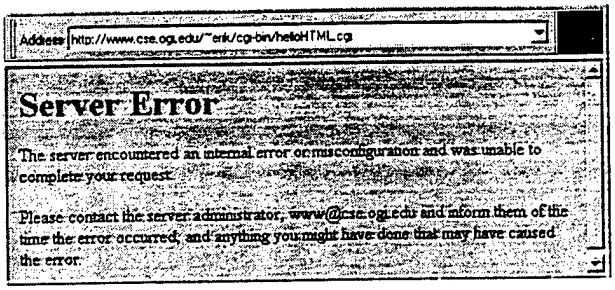
There are two alternatives. The first is that the server expects scripts to be in a special directory, usually called `cgi-bin`. It will treat any URL involving `cgi-bin` as an executable script. The second posibility is that the server uses a particular filename extension, usually `.cgi`, to distinguish scripts from other type of documents. In this case scripts can reside anywhere in the directory hierarchy.

The most common problem that occurs when using CGI scripts is that the server does not recognize that it should execute a script. When the browser tries to download a script, or displays its source, you know that the server has not recognized the URL as a script. Another symptom of this problem is the "Method not implemented" or "Cannot post to non-script" server error.





It can also happen that a script is executed, but that the server returns an empty page or an internal server error.



This means that the server has tried to execute the script, but something has gone wrong; usually because of wrong permissions or because the script crashed. Of course, scripts should be world executable and readable for the server to execute them. Under Unix, scripts run as user `nobody`, so all directories above the script should be executable. Remember that scripts are executed in a different environment then when executed from the command-line, in particular you should not rely on the `PATH` variable to search files or executables.

Under Unix, a handy way of debugging a script (say `foo.cgi`) that is recognized

by the server, but still crashes, is by wrapping it in a shell script that simply returns the exact output of the script:

```
#!/bin/sh
echo "Content-type: text/html"
echo
echo "<H1>Start of script's output</H1>"
echo "<PRE>"
foo.cgi
echo "</PRE>"
echo "<H1>End of script's output</H1>"
```

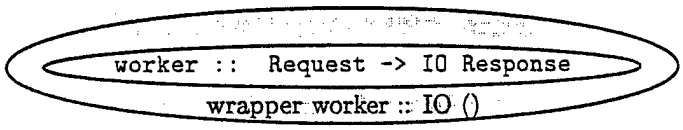
Of course, a script must run perfectly from the command-line to begin with.

4 Programming CGI scripts

Now that we know how to access dynamic documents from the client, where to put them on the server, and how to debug them, we can start looking at coding CGI scripts.

4.1 Worker/wrapper

The architecture of our CGI-library provides the programmer with the illusion of an idealized HTTP client (the *wrapper*), which interacts with an idealized HTTP server (the *worker*) to be supplied by the programmer:



All the low-level details of the communication between the actual HTTP server and the script are handled by the wrapper function. As far as the server is concerned, the wrapper is a standard CGI script that recieves the HTTP request forwarded from the client via specific environment variables and the standard input and returns the response to be returned to the client via the standard output. As far as the wrapper is concerned, the worker is a function that produces a result of type `Response` from an argument of type `Request`.

The work done by the wrapper is the same for all CGI scripts; the interesting bits are performed by the worker; the wrapper decodes the HTTP request into a value of the algebraic type `Request`, passes it to the worker to obtain a value of type `Response`, and encodes this into an actual HTTP response:

```
wrapper :: (Request -> IO Response) -> IO ()
wrapper worker
 = do{ request <- getRequest
     ; response <- worker request
     ; putResponse
     }
```

The worker function need not be concerned with any of the gory details of the CGI standard, it only has to produce an abstract response when given an abstract request. Most other CGI libraries do not decouple abstract and concrete requests and responses. The result is that scripts are less readable, less modular, less flexible, and take longer to develop.

To understand the logic of a CGI script, we can study the worker function in isolation. This is impossible if the decoding and encoding of requests and responses is intertwined with the actual computation of responses from requests.

As we show in sections 6.2 and 8.5.2, we can recursively split the worker function into a wrapper and a simplified worker and leverage of the claimed benefits yet another time. If there is no a priori distinction between worker and wrapper, this is obviously impossible.

Because the wrapper function abstracts all details of the CGI standard from the worker, it is easy to adapt to a platform such as Windows, which prefers a nonstandard interaction (ISAPI) between servers and scripts. In that case we only have to change the wrapper function once, instead of having to modify *all* our scripts.

4.2 Modelling requests and responses

The set of abstract HTTP requests is modelled by the data type `Request`. A client can either request to retrieve a document (using `GET`), or deposit some Mime content (using `POST`):

```
data Request = GET QueryString | POST Mime
```

The set of abstract HTTP responses is modelled by the data type `Response`. The server can either return some Mime content, a redirection to another location, or an error message:

```
data Response = Content Mime | Location Url | Status Code Reason
```

Mime types are the standard way of "typing" data transmitted over the internet. Examples include plain text (`text/plain`, section 7), HTML documents (`text/html`, section 8), url-encoded query strings (`x-application/url-encoded`, section 5.1), GIF pictures (`image/gif`) and MPEG movies (`video/mpeg`).

```
data Mime
 = ...
 | TextPlain  String
 | TextHtml   HTML
 | UrlEncoded [(String,String)]
 | ...
```

Besides the constructor functions for the data types `Request` and `Response`, we assume only that we can get requests from the outside world via function `getRequest` (section 5) and put responses to the outside world via function `putResponse` (section 6):

```
getRequest :: IO Request
putResponse :: Response -> IO ()
```

To implement functions **getRequest** and **putResponse**, we must exchange Mime types with the outside world, hence we also need:

```
getMime :: IO Mime
putMime :: Mime -> IO ()
mimeType :: Mime -> String
```

In order to implement functions **getMime** and **putMime**, we need to parse and unparse the various alternatives of data type Mime:

```
showHTML :: HTML -> String
...
readQuery :: String -> [(String,String)]
```

5 Decoding requests

The most four important environment variables that the HTTP server passes to the wrapper are `REQUEST_METHOD`, `QUERY_STRING`, `CONTENT_LENGTH`, and `CONTENT_TYPE`. By inspecting these variables, the wrapper can decode the incoming request.

Function `getRequest` does a simple case analysis on the environment variable `REQUEST_METHOD` to find out what request has been made:

```
getRequest
 = do{ method <- getEnv "REQUEST_METHOD"
     ; case method of {"GET"  -> getGET; "POST" -> getPOST}
     }
```

If `REQUEST_METHOD` equals `GET`, then the variable `QUERY_STRING` contains an url-encoded query string:

```
getGET
 = do{ query <- getEnv "QUERY_STRING"
     ; return $ GET (readQuery query)
     }
```

If `REQUEST_METHOD` equals `POST`, then function `getMime` decodes the Mime content the request:

```
getPOST
 = do{ mime <- getMime
     ; return $ POST mime
     }
```

Function `getMime` gets the first `CONTENT_LENGTH` bytes of the standard input, and then reads a value of Mime type `CONTENT_TYPE`:

```
getMime
 = do{ contentLength <- getEnv "CONTENT_LENGTH"
     ; stdin <- getContents
     ; let mime = take (read contentLength) stdin
     ; contentType <- getEnv "CONTENT_TYPE"
     ; case contentType of
         "application/x-url-encoded"
           -> return $ UrlEncoded (readQuery mime)
         ....
     }
```

5.1 Reading Mime type application/x-url-encoded

An url-encoded query string consist of a sequence of zero or more url-encoded *name=value* pairs separated by ampersands &:

$$query \quad ::= \quad [name{=}value\{\&name{=}value\}]$$

We can directly translate this grammar into a parser using standard parser combinators:

```
query :: Parser [(String,String)]
query
 = do{ name <- urlEncoded; string "="; value <- urlEncoded
     ; return (name,value)
     } `sepby` (string "&")
```

Names and values are url-encoded, which the parser will decode:

```
urlEncoded :: Parser String
urlEncoded
 = many (alphanum ++ extra ++ safe ++ space ++ hexencoded)
```

Alphanumeric characters and "safe" and "special" characters are unencoded:

```
extra :: Parser Char
extra = sat (`elem` "!*'(),")
```

```
safe :: Parser Char
safe = sat (`elem` "$-_.")
```

Spaces " " are encoded by plus signs "+":

```
space :: Parser Char
space = do{ char '+' ; return ' '}
```

Nonalphanumeric characters such as "%" are hex-encoded via an escape sequence that consists of a percent character % followed by two hexadecimal digits, for example the hex-encoding of % itself is %25.

```
hexencoded :: Parser Char
hexencoded
  = do{ char '%'; d1 <- hexit; d2 <- hexit
      ; return $ chr (readHex [d1,d2])
      }
```

Compare this with the Perl code of Steve Brenner's `cgi-lib.pl`, one of the most popular libraries for writing CGI scripts in Perl:

```
sub ReadParse {
   local (*in) = @_ if @_;
   local ($i,$key,$val);

   if (&MethGet){
      $in = $ENV{'QUERY_STRING'};
   } elsif {
      read(STDIN,$in,$ENV{'CONTENT_LENGTH'});
   }

   @in = split(/[&;]/,$in);
   foreach $i (0.. $#in){
      $in[$i] =~ s/\+/ /g;
      ($key,$value) = split (/=/,$in[$i],2);
      $key =~ s/%(..)/pack("c",hex($1))/ge;
      $val =~ s/%(..)/pack("c",hex($1))/ge;
      $in{$key} .= "\0" if (defined($in{$key}));
      $in{$key} .= $val;
   }

   return scalar(@in)
```

6 Encoding responses

Function putResponse puts a response to the standard output in the the exact format that is required by the CGI standard:

```
putResponse :: Response -> IO ()
putResponse response
 = case response of
      Content mime
       -> do{ putStr ("Content-type: " ++ mimeType mime)
            ; putStr "\n\n"
            ; putMime mime
            }
      Location url
       -> do{ putStr ("Location: " ++ url)
            ; putStr "\n\n"
            }
      Status code reason
       -> do{ putStr ("Status: " ++ code ++ " " ++ reason)
            ; putStr "\n\n"
            }
```

6.1 Preventing caching

Most web browsers cache the documents they request, so that the next time a document is requested it need not be downloaded from the server. For dynamic documents this is clearly not the right thing to do. Now the advantages of decoupling response generation and response presentation really kick in; we can just change function `putResponse` to print an extra `"Pragma: no-cache"` header in each response.

6.2 Specializing the wrapper

In practice, requests contain just url-encoded data and responses contain just HTML documents. We can capture this pattern by providing another wrapper function, which takes a simplified script into the general script that the wrapper expects:

```
cgi :: ([(String,String)] -> IO HTML) -> IO ()
cgi script = \request -> wrapper $
   do{ html <- script (fromRequest request)
     ; return (Content (HTML html))
     }
```

Function `fromRequest` extracts the query string from either a `GET` or `POST` request:

```
fromRequest :: Request -> [(String,String)]
fromRequest request
 = case request of
    GET query                -> query
    POST (UrlEncoded query) -> query
    otherwise                -> []
```

7 MIME type `text/plain`

Using mime type `TextPlain` we can write our first CGI script, the cut-and-dried "Hello World!" program. It ignores its environment argument, and returns the obvious content.

```
helloWorld :: IO ()
helloWorld = wrapper $ \request ->
   do{ return (textplain "Hello World!") }
```

where function `textplain` wraps a string in a plain text response:

```
textplain :: String -> Response
textplain s = Content (TextPlain s)
```

7.1 A more interesting script

One of the environment variables that is passed to a CGI script is `REMOTE_HOST`, which contains the fully qualified domain name of the client (such as `tibet.cse.ogi.edu`) that has sent the request to the server. The top-level domain (the name after the rightmost dot) gives us some information about the country where the client is located, and we will use that to generate a personalized salutation:

```
greetings :: Domain -> String
greetings d
 = case (domain d) of
     "edu" -> "Hi there!"
     "com" -> "Can you find everything OK today?"
     "nl"  -> "Hoi, hoe gaat het?"
     "uk"  -> "Good afternoon!"
     _     -> "Ahum, ..."
```

We can extract the top-level domain of a fully qualified domain name by first splitting it at every `'.'` and then taking the last element of the resulting list:

```
domain = last.split (== '.')
```

The function `split (== '.') "tibet.cse.ogi.edu"` returns the list `["tibet", "cse", "ogi", "edu"]`.

The script itself is straightforward. We lookup the `REMOTE_HOST` in the environment, and compute the greeting:

```
helloWorld = wrapper $ \query ->
   do{ host <- getEnv "REMOTE_HOST"
     ; return (textplain (greetings host))
     }
```

8 MIME type `text/html`

Most CGI scripts written in C or Perl print concrete HTML directly on the standard output. In Perl, the HTML variant of the "Hello World!" CGI script would look something like:

```
print "Content-type: text/html\n\n";
print "<html>\n";
print "<head>\n";
print "<title>Hello, world!</title>\n";
printf "</head>\n";
printf "<body>\n";
printf "<h1>Hello, world!</h1>\n";
printf "</body>\n";
printf "</html>\n";
```

This is not very flexible, especially when we want to generate more complicated HTML pages. Just for comparison, using the combinators we will develop in this section (8.1), the script:

```
helloHTML = cgi $ \query ->
   do{ return (page "Hello, world!" [h1 "Hello, world!"]) }
```

generates the same HTML content.

An HTML document consists of a number of nested *elements* such as headers (page title, section headings), paragraphs, lists (ordered, unordered), logical markup (citation, computer code), visual markup (italic, bold), hypertext links, images, fill-in forms etc.

Every HTML element is delimited by begin- and end-*tags* of the form <*tag*> respectively </*tag*>. Tags (and attributes) in HTML are not case sensitive, so for example <HTML> is equivalent to <html> or <HtMl>. Also, most browsers support elements that are not part of the official HTML standard. Nonstandard elements of competing browsers, or unsupported elements, are just ignored. Some tags such as <HR> and
 do not need a closing tag, but it does no harm to use one anyway.

Most elements take (optional) arguments, which are given as *name=value* pairs in the start tag. Boolean attributes are set by just giving their name, without a value.

HTML can be represented by a simple universal tree type. An HTML value is either just ordinary text, or a complex element with a tag, a list of attributes, and an embedded list of HTML values.

```
data HTML
 = Text{text :: String}
 | Element{ tag :: Tag
          , attributes :: [(Name,Value)]
          , html :: [HTML]
          }
```

For simplicity, all HTML related types such as `Tag`, `Name`, `Value`, and later on `Color`, `Face`, `Size`, etc. are synonyms for `String`.

8.1 Basic combinators

The basic HTML combinators `set`, `attributedElement` and `element`, and `prose` provide an abstract interface to construct values of type `HTML`.

```
set ::  [(Name,Value)] -> (HTML -> HTML)
attributedElement ::  Tag -> [(Name,Value)] -> [HTML] -> HTML
element ::  Tag -> [HTML] -> HTML
prose ::  String -> HTML
```

By hiding the construction of concrete HTML elements we can always decide to change the representation of the HTML data type. The combinators whose signatures are given in Figure 1 capture patterns that we have found convenient when generating HTML programatically.

```
page ::  String -> [HTML] -> HTML

format ::  Tag -> String -> HTML

h ::  Int -> String -> HTML
p ::  [HTML] -> HTML
font ::  Color -> [Face] -> Size -> [HTML] -> HTML

href ::  URL -> [HTML] -> HTML
name ::  String -> [HTML] -> HTML
image ::  String -> URL -> HTML

ul ::  [[HTML]] -> HTML
ol ::  [[HTML]] -> HTML
dl ::  [(String,[HTML])] -> HTML

table ::  [[ [HTML] ]] -> HTML
```

Figure 1: Advanced HTML combinators

8.2 Printing the environment

Script `envPassed` nicely formats the environemnt variables that are set by the server. It maps a list of pairs like

```
[("SERVER_NAME","www.cse.ogi.edu"),("REQUEST_METHOD","GET"),..]
```

into the HTML definition list

```
dl [ ("SERVER_NAME",    [prose "www.cse.ogi.edu"])
   , ("REQUEST_METHOD",[prose "GET"])
   , ...
   ]
```

A simple list comprehension does the job; for every `(dt,dd)` pair in the environment we construct a pair `(dt,[prose dd])`, and then wrap the resulting list in a definition list:

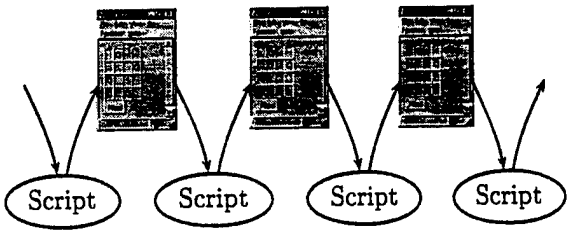
```
showEnv env
 = page "Environment"
    [ h1 "Environment"
    , dl [ (dt,[prose dd])
         | (dt,dd) <- env
         ]
    ]
```

The complete script first gets the list of all environment variables using function `getWholeEnv`, and then returns the requested HTML page:

```
envPassed = cgi $ \query ->
   do{ env <- getWholeEnv
     ; return (showEnv env)
     }
```

8.3 Forms

An interactive script can ask for input from the user by returning an HTML-form:



When the user submits the form, the request that contains the form-data is posted to the script that generated the form.

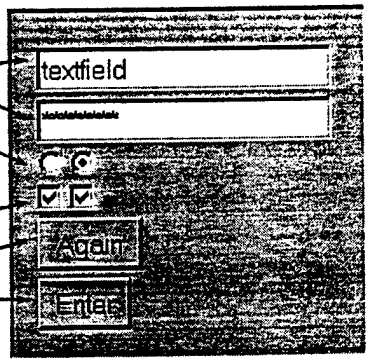
HTML forms can contain standard GUI elements such as text-fields, various kinds of buttons, menus, etc.

```
gui ::  [HTML] -> HTML
widget ::  Widget -> (Name,Value) -> HTML
menu ::  Name -> [Value] -> HTML
textarea ::  Name -> Int -> Int -> Wrap -> Value -> HTML
```

The `gui` combinator takes list of HTML elements into a form that collects the (name,value) pairs to be posted to the script which generated the form.

A widget `widget` *w* `(`*name*`,`*value*`)` associates the name *name* with a value, which is either the initial value *value* or a value that is supplied by the user.

```
gui
 [ widget "text"     ("t","textfield")
 , widget "password" ("p","password")
 , widget "radio"    ("r","r1")
 , widget "radio"    ("r","r2")
 , widget "checkbox" ("c","c1")
 , widget "checkbox" ("c","c2")
 , widget "reset"    ("x","Again")
 , widget "submit"   ("s","Enter")
 , widget "hidden"   ("h","invisible")
 ]
```



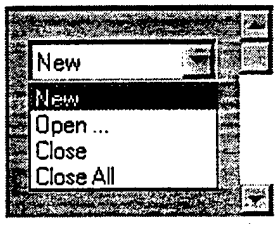
When the user clicks on the enter button, the url-encoded string

```
t=textfield&p=password&r=r2&c=c1&c=c2&s=Enter&h=invisible
```

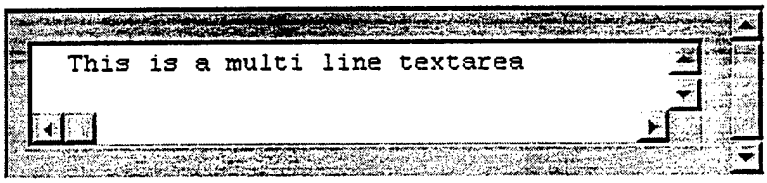
is submitted to the script. Hidden fields have no visible rendering, but do turn up in the request. Turning a `radio` button on turns off all others with the same name. Also note that names might occur more than once (for example `"c"`) in an url-encoded query string.

The menu element `menu` ***name*** `[...,` ***value_i***`, ...]` renders as a pull-down menu that associates name ***name*** with the alternative chosen by the user.

```
menu "File"
 [ "New"
 , "Open ..."
 , "Close"
 , "Close All"
 ]
```



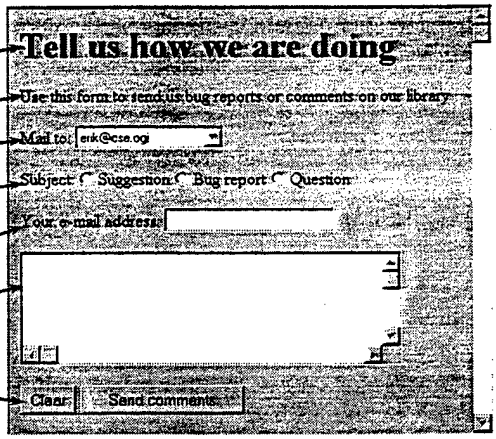
The combinator `textarea` ***name rows cols wrap value*** combinator renders as a multiline textarea whose content is paired with its name when the form is submitted.



8.4 A user feedback form

As an example application of HTML forms, we write a script that processes user feedback.

```
feedbackForm
 = page "User Feedback" []
    [ header
    , introduction
    , gui
       [ to
       , subject
       , from
       , body
       , resetORsubmit
       ]
    ]
```

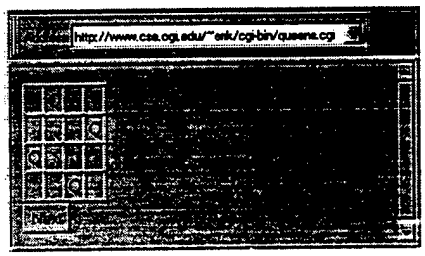


The script first parses its input into a mail message. If this fails it just returns the original form so that the user can try again. Otherwise it mails the message and returns an acknowledgement form.

```
feedback = \query -> cgi $
  case check query of
    { Nothing  -> do{ return feedbackForm}
    ; Just msg -> do{ sendMail msg; return (acknowledgeForm msg)}
    }
```

8.5 N Queens

Our next script is an interactive version of the n-queens problem.



We first construct a function advance that computes the next solution from a given board configuration (section 8.5.1). Only then we wrap this into a CGI script (section 8.5.2).

8.5.1 Non interactive version

A board is represented by a list of row positions:

```
type Board = [Row]
type Row   = Int
n = 8 :: Row
```

A possible board has n queens, one in each row:

```
possible :: Board -> Bool
possible board
 = and [length board == n, all (`elem` [1..n]) board]
```

A possible board is unsafe when other queens are on the same left-diagonal, the same row, or the same right-diagonal as the left-most queen:

```
unSafe :: Board -> Bool
unSafe []
 = False
unSafe (q:qs)
 = or [ onPath (\x -> x-1) q qs
      , onPath (\x -> x)   q qs
      , onPath (\x -> x+1) q qs
      ]
```

Function advance moves the left-most queen forward until the board is safe. If the queen already is in row n, the next queen is recursively advanced too, and the left-most queen starts again in row 1.

```
advance :: Board -> Board
advance []
 = []
advance (q:qs)
 = getSafe (if q == n then 1:advance qs else q+1:qs)
   where
      getSafe qs = if unSafe qs then advance qs else qs
```

Function samePath checks wether there are other queens on the indicated path:

```
onPath :: (Row -> Row) -> Row -> Board -> Bool
onPath next q []
 = False
onPath next q (q':qs)
 = next q == q' || onPath next (next q) qs
```

8.5.2 Interactive version

We now continue by embedding this version of the n-queens problem into an interactive CGI program.

```
nQueens = cgi $ \query ->
   do{ return $ (showBoard.advance.readBoard) query }
```

Function readBoard reads variable "Board" from the query. If this is an impossible board, it will return the possible board [1..n].

```
readBoard :: [(String,String)] -> Board
readBoard query
 = if possible board then board else [1..n]
   where
      board = read (lookup query "Board")
```

Function showBoard uses a table to present the current board to the user. It stores the actual configuration in the hidden field "Board".

```
showBoard :: Board -> HTML
showBoard board
 = page (show n ++ " Queens")
      [ gui [ showSolution board
            , widget "submit" "Next", widget "hidden" (show board)
            ]
      ]

showSolution board
 = table [ [ if q == i then [prose "Q"] else [prose " "]
           | i <- [1..n]
```

```
  ]
| q <- board
]
```

9 Conclusions

To be written.

10 References and further reading

To be written.

Acknowledgements

Many thanks to Jim Hook, Tim Sheard, and Daan Leijen for reading draft versions of these notes. Special thanks to Phil Wadler who encouraged me to write this paper in the first place, and to Simon Peyton Jones for suggesting some major restructuring.

Scripting COM components in Haskell

Simon Peyton Jones (simonpj@dcs.gla.ac.uk)
University of Glasgow and Oregon Graduate Institute

Erik Meijer (erik@cs.ruu.nl)
University of Utrecht and Oregon Graduate Institute

Daan Leijen (leijen@wins.uva.nl)
University of Amsterdam and Oregon Graduate Institute

December 15, 1997

Abstract

Designers of advanced languages, such as ML, Prolog, or Haskell, face an uphill struggle to persuade potential users of the merits of their approach. In fact, it has hitherto been impossible to find other than niche applications because (foreign language interfaces notwithstanding) it has been too difficult to integrate software components written in new languages with large bodies of existing code.

Microsoft's Component Object Model (COM) offers this community a new opportunity. Because the interface between objects is by design language independent and arms-length, it is possible either to write glue programs that integrate existing COM objects, or to write software components whose services can be used by clients written in more conventional languages.

We describe our experience of exploiting this opportunity in the purely-functional language Haskell. We describe a design for integrating COM components into Haskell programs, and we demonstrate why someone might want to script their COM components in this way.

This paper has been submitted to Software Reuse 1998.

keywords

Haskell, COM, CORBA, software components, lazy evaluation, functional programming, strong typing, polymorphism, scripting, interoperability, equational reasoning.

1 Introduction

Programming-language researchers have a serious marketing problem. Apart from a relative handful of enthusiasts, our languages are not widely used, because no potential customer is prepared to revolutionize the way they build their systems — and rightly so. Despite some work on foreign-language interfaces, it has been hard to provide an evolutionary path that would enable a potential customer to experiment with a new language at a low level of commitment.

Microsoft's Component Object Model (COM) is a widely-deployed, binary standard for software components [12]. Because its language independence, COM presents two new opportunities for programming-language researchers. COM makes it easier to use a new language either (a) to glue together, or script, a collection of existing COM components to make a larger application or component, or (b) to implement a new COM component that a client can use without knowledge of its implementation language.

We have begun to exploit the first of these opportunities in the context of the purely functional programming language Haskell [4]. In this paper we describe an interface between Haskell and COM that makes it easy to script COM components from a Haskell program. We make two main contributions:

- A graceful and strongly typed accommodation of COM within the host language is important. We present a design for how COM could appear to the Haskell programmer.
- If the exercise is to be more than just "Gosh, we can script COM in Haskell as well as in Visual Basic" then it is important to demonstrate some added value from using a higher-order, typed language. We offer such a demonstration, in the form of a case study.

2 The opportunity

Until recently it has been much easier for a client program to use software components (libraries, classes, abstract data types) written in the same language:

1. The specification of the interface between the component and its clients is usually given in a language-specific way; for example, as C++ class descriptions.

2. The calling convention between client and component is often language-specific, or perhaps even unspecified (because both client and component are assumed to be compiled with the same compiler)

3. Programmers can assume a rather intimate coupling between the address spaces of client and component; for example, the client might pass a pointer into the middle of an array, to be side-effected by the component.

COM encapsulates a software component in a way that contrasts with each of these three aspects:

- The interface between client and component is specified in IDL (COM's Interface Definition Language). For each particular language, tools are provided to convert IDL into the corresponding specification in that language (section 3.4).
- COM specifies the client/component interface at a binary level, independently of any particular language or compiler (section 3.1).
- Parameters are expected to be marshalled from the client's address space to the component's address space, and vice versa. Sometimes the two share an address space, in which case marshalling need do no copying, but all COM calls provide enough information to do such marshalling.
- Interfacing between two languages often carries performance overheads, because of differing data representation and memory-allocation policies. When the alternative is a native-language interface between client and component, these extra overheads can seem rather unattractive.

  However, anyone using COM has already bitten the bullet: they have declared themselves willing to accept a hit in programming convenience, and perhaps a hit in performance (for marshalling), in exchange for the advantages that COM brings.

These are not COM's only advantages. For example, one of the primary motivations for using COM concerns version control and upgrade paths for software components, which we have not mentioned at all so far. However, these additional properties are well described elsewhere, [11, 12, 1, 2, 3] and do not concern us further in this paper, except in so far as they serve as motivators for people to write and use COM components.

Also, COM is not alone in having these properties. Numerous research projects had similar goals, in particular CORBA [13]. In fact, almost everything in the rest of this paper would apply to CORBA as well as COM, because CORBA is largely compatible with COM. We stick to COM for the sake of being concrete (it has a well-defined, mature and stable specification) and because of its widespread use. With more than 200 million systems worldwide using it, COM offers designers of advanced languages the best opportunities for reusing software components.

3 How COM works

Although there are many very fat books about COM (e.g. [12]), the core technology is quite simple, a notable achievement. This section briefly introduces the key ideas. We concentrate exclusively on *how* COM works, rather on *why* it works that deal; the COM literature deals with the latter topic in detail.

Here is, in C, how a client might create and invoke a COM object:

```
/* Create the object */
err_code = CoCreateInstance ( cls_id
                            , iface_id
                            , &iface_ptr
                            );
if (not SUCCEEDED(err_code)) {
 ...error recovery...
}

/* Invoke a method */
(*iface_ptr)[3]( iface_ptr, x, y, z );
```

The procedure `CoCreateInstance` is best thought of as an operating system procedure. (In real life, it takes more parameters than those given above, but they are unimportant here.) Calling `CoCreateInstance` creates an instance of an object whose *class identifier*, or CLSID, is held in `cls_id`. The class identifier is a 128-bit *globally unique identifier*, or GUID. Here "globally unique" means that the GUID is a name for the class that will not (ever) be re-used for any other purpose anywhere on the planet. A standard utility allows an unlimited supply of fresh GUIDs to be generated locally, based on the machine's IP address and the date and time.

The code for the class is found indirectly via the *system registry*, which is held in a fixed place in the file system. This double indirection of CLSIDs and registry makes the client code independent of the specific location of the code for the class. Next, `CoCreateInstance` loads the class code into the current process (unless it has already been loaded). Alternatively, one can ask COM to create a new process (either local or remote) to run the instance.

3.1 Interfaces and method invocation

A COM object supports one or more *interfaces*, each of which has its own globally-unique *interface identifier* or IID. That is why `CoCreateInstance` takes a second parameter, `iface_id`, the IID of the desired interface; `CoCreateInstance` returns the *interface pointer* of this interface in `iface_ptr`. There is no such thing as an "ob-

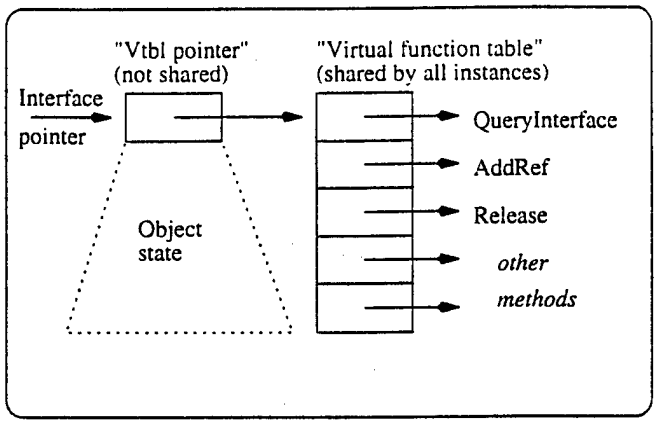


Figure 1: Interface pointers

ject pointer", or "object identifier"; there are only interface pointers.

The IID of an interface uniquely identifies the complete *signature* of that interface; that is, what methods the interface has (including what order they appear in), their calling convention, what arguments they take, and what results they return. If we want to change the signature of an interface, we must give the new interface a different IID from the old one. That way, when a client asks for an interface with a particular IID, it knows exactly what that interface provides.

A COM interface pointer is (deep breath) a pointer to a pointer to a table of method addresses (Figure 1). Notice the double indirection, which allows the table of method addresses to be shared among all instances of the class. Data specific to a particular instance of the class, notably the object's state, can be stored at some fixed offset from the "vtbl pointer" (Figure 1). *The format of this information is entirely up to the object's implementation; the client knows nothing about it.* Lastly, when a method is invoked, the interface pointer must be passed as the first argument, so that the method code can access the instance-specific state. Taking all these points together, we can now see why a method invocation looks like this:

```
(*iface_ptr)[3]( iface_ptr, x, y, z );
```

None of this is language specific. That is, COM is a binary interface standard. Provided the code that creates an object instance returns an interface pointer that points to the structures just described, the client will be happy. In theory, the parameter passing conventions for each method can be different (but fixed in advance). In practice, they match the `__stdcall` convention used by C and C++.

Interface pointers provide the sole way in which one can interact with a COM object. This restriction makes it possible to implement *location transparency* (a major COM war-cry), whereby an object's client interacts with the object in the same way regardless of whether or not the object is in the same address space, or even in the same machine, as the client. All that is necessary is to build a *proxy* interface pointer, that *does* point into the client's address space, but whose methods are stub procedures that marshal the data to and from across the border to the remote object.

3.2 Getting other interfaces

A single COM object can support more than one interface. But as we have seen before `CoCreateInstance` returns only one interface pointer. So how do we get the others? Answer: every interface supports the `QueryInterface` method, which maps an IID to an interface pointer for the requested IID or fails if the object does not support the requested interface. So, from any interface pointer (`iface_ptr`) on an object we can get to any other interface pointer (`iface_ptr2`) which that object implements, for example:

```
err_code = (*iface_ptr)[0]( iid2, &iface_ptr2 );
```

Why "`[0]`"? Because `QueryInterface` is at offset 0 in every interface.

The COM specification requires that `QueryInterface` behaves consistently. The `IUnkown` interface on an object is the identity of that object; queries for `IUnknown` from any interface on an object should all return exactly the same interface pointer. Queries for interfaces on the same object should always fail or always succeed. Thus, the call `(*iface_ptr)[0](iid2,&iface_ptr2);` should not succeed at one point, but fail at another. Finally, the set of interfaces on an object should form an equivalence relation.

3.3 Reference counting

Each object keeps a *reference count* of all the interface pointers it has handed out. When a client discards an interface pointer it should call the `Release` method *via* that interface pointer; every interface supports the `Release` method. Similarly, when it duplicates an interface pointer it holds, the client should call the `AddRef` method *via* the interface pointer; every interface also supports the `AddRef` method. When an object's reference count drops to zero it can commit suicide — but it is up to the object, not the client, to cause this to happen. All the client does is make correct calls to `AddRef` and `Release`.

3.4 Describing interfaces

Since every IID uniquely identifies the signature of the interface, it is useful to have a common language in which to describe that signature. COM has such a language, called IDL (Interface Definition Language) [6], but IDL is not part of the core COM standard. You do not have to describe an interface using IDL, you can describe it in classical Greek

```
[object,
 uuid(00000000-0000-0000-C000-000000000046),
 pointer_default(unique)
]
interface IUnknown {
 HRESULT QueryInterface( [in] REFID iid,
                         [out] void **ppv );
 ULONG   AddrRef( void );
 ULONG   Release( void );
}
```

Figure 2: The IUnknown interface in IDL

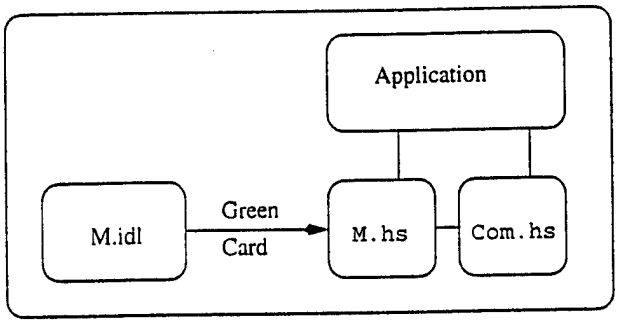


Figure 3: The big picture

prose if you like. All COM says is that one IID must identify one signature.

Describing an interface in IDL is useful, though, because it is a language that all COM programmers understand. Furthermore, there are tools that read IDL descriptions and produce language-specific declarations and glue code. For example, the Microsoft MIDL compiler can read IDL and produce C++ class declarations that make COM objects look exactly like C++ objects (or Java, or Visual Basic).

As a short example, Figure 2 gives the IDL description of the IUnknown interface, the interface of which every other is a superset. The 128 bit long constant is the GUID for the IUnknown interface.

4 Interfacing Haskell and COM

Our goal is to provide a convenient and type-secure interface between a Haskell program and the COM objects it manipulates. How could COM objects appear to the Haskell programmer?

Our approach, illustrated in Figure 3, is broadly conventional. We have built a tool, called Green Card, that takes an IDL module `M.idl`, and from it generates a Haskell module `M.hs`[1]. Object instances live in the C world (adding yet another level of indirection), and are accessed in the Haskell world using our previously developed foreign language interface to C[9]. Green Card automatically generates all required stub procedures and marshalling code to call C. The `M.hs` module, together with a library Haskell module `Com.hs`, is all that an application need import to access and manipulate all the COM objects described by `M`.

4.1 What Green Card generates

So what does the Haskell module `M` export?

- For each CLSID `Baz` in the IDL module, module `M` exports a value `baz` of type `ClassId`. This value represents the CLSID of class `Baz`. `ClassId` is an abstract type exported by `Com.hs`.
- For each IID `IFoo` in the IDL module, `M` exports:
  - A new, abstract, Haskell data type `IFoo`. Surprisingly, no operations are provided on values of type `IFoo`.
  - A value `iFoo` of type `Interface IFoo`. This value represents the IID for `IFoo`. `Interface` is an abstract type constructor exported by `Com.hs`.

  An interface pointer for an interface whose IID is `IFoo` is represented by a Haskell value of type `Com IFoo`. `Com` is an abstract type constructor exported by `Com.hs`.
- For each method `meth` in the interface `IFoo`, module `M` exports a Haskell function `meth` with the type:

  `meth :: a1 -> ... -> an -> Com IFoo -> IO r`

  Here, $a_1, \ldots, a_n$ are the argument types expected by `meth`, extracted from the method's IDL signature, and r is its result type. (If there are many results then `meth` would have a tuple result type `IO` $(r_1, \ldots, r_n)$.) The interface pointer is passed as the last argument for reasons we discuss later.

  Notice that `meth` cannot be invoked on any interface pointer whose type is other than `Com IFoo`, so the interface is type-secure.

  The result of `meth` has type "`IO r`" rather than simply "r" to signal that `meth` might perform some input/output. In Haskell, a function that has type `Int -> Int`, say, is a function from integers to integers, no more and no less. In particular it cannot perform any input/output. All functions that can perform I/O have a result type of the form $IO\ r$. This so-called *monadic I/O* has become the standard way to do input/output in purely functional languages [8].

[1] In fact, rather than reading the IDL text directly, the tool interrogates the *type library* for `M`, a COM object generated by a Microsoft tool from the IDL. The Microsoft tool does all the parsing and type-checking of the IDL. The type-library object it produces is essentially a parse tree with methods that allow its clients to navigate the parse tree. The tool itself is written in Haskell and has been bootstrapped to generate the Haskell module to access type library components.

- The library module `Com.hs` provides a generic procedure `createInstance`:

```
createInstance :: ClassId
               -> Interface i
               -> IO (Com i)
```

  Like `CoCreateInstance`, it takes a CLSID and and IID, and returns an interface pointer. Unlike the C++ procedure `CoCreateInstance`, however, we use polymorphism to record the fact that the interface pointer returned "corresponds to" the IID passed as argument. This somewhat unusual use of polymorphism elegantly captures exactly what we want to say, and achieves type safety without having to resort to type casts as in C or Java.

  The IO type has an exception mechanism that is used to deal with the failure of `createInstance`.

- The library module `Com.hs` provides a generic procedure `queryInterface`:

```
queryInterface :: Interface j
               -> Com i
               -> IO (Com j)
```

  The first argument is the IID for the desired interface. The second is the interface on which we want to query for another interface. The result is an interface for the desired interface. Again, we use polymorphism to make sure that the interface that is returned by `queryInterface` (of type `Com j`) corresponds to the IID (of type `Interface j`) passed as the first argument.

- There are no programmer-visible procedures corresponding to `AddRef` and `Release`. Instead, when Haskell's garbage collector discovers that a value of type `Com i` is now inaccessible, it calls `Release` on the interface pointer it encapsulates. This is just a form of *finalization*, a well-known technique in which the garbage collector calls a user-defined procedure when it releases the store held by an object.

4.2 The Agent example

These points make more sense in the context of a particular example. Suppose we took the IDL description for Microsoft Agent. After being processed by Green Card, we would have a Haskell module `Agent.hs` that exports (among other things) the types, functions, and values given in Figure 4.

Microsoft Agent implements cartoon characters that pop up on the screen and talk to you. The animation is supported by an *agent server* whose CLSID is `agentServer`, and whose main interface is `IAgent`. Once we have created an agent server, we can load a character from a file, getting a `CharId` in reply. Now we can generate instances of that character using `getCharacter`, getting an interface pointer for the character in return[2]. Having got a character, we can make it talk a sentence by calling `speak`, or play a little animation by calling `play`.

Here is a complete example program:

```
module Main where
import Agent

main = comRun $
 do server <- createInstance
                agentServer SERVER iAgent
    rob_id <- server # load "robby.acs"
    robby  <- server # getCharacter rob_id
    robby # moveTo centerScreen
    robby # show
    robby # speak "Hello world"
```

To make sense of this, we need to know the following Haskell lore:

- Left associative function application is written as juxtaposition. Thus `f a b` means "`f` applied to `a` and `b`". Right associative function application is written as `$`. Thus `f $ g a` means "`f` applied to `g a`".

- The function `#` is simply reverse function application.

```
(#) :: a -> (a->b) -> b
x # f = f x
```

  It is used here to allow us to write the interface pointer first in a method call, much as happens in an object oriented language. For example, `robby # speak "Hello"` means the same as `speak "Hello" robby`. It is for this reason that Green Card arranges that the interface pointer is the last parameter of each method call.

- The "do" notation is used to sequence a series of I/O-performing function calls. It is much more syntactically convenient than using the bind and unit functions of the monad, as the first papers about monadic I/O did [8, 10]. The statement `robby <- server # getCharacter rob_id` binds the result of performing the action `server # getCharacter rob_id` to the name `robby`.

Now we can read the example. The function `comRun` is exported by `Com.hs` and has type

```
comRun :: IO a -> IO ()
```

It encapsulates a computation that accesses COM, preceding it with initialization and following it with finalization.

[2] It is quite common for COM calls to return interfaces. Here, `getCharacter` plays the role of `createInstance`, returning an interface to the new character. The interface may be have been created inside the agent server by a call to `CoCreateInstance` but that does not concern us.

```
module Agent where

    -- The Agent class
  agentServer :: ClassID

    -- The IAgent interface
  data IAgent = ...              -- Agent interface type
  iAgent :: Interface IAgent     --  ...and its IID

  type CharId = Int
  load         :: String -> Com IAgent -> IO CharId
  getCharacter :: CharId -> Com IAgent -> IO (Com IAgentCharacter)
  ...etc other methods of IAgent...

    -- The AgentCharacter interface
  data IAgentCharacter = ...  -- Ditto IAgentCharacter
  iAgentCharacter :: Interface IAgentCharacter

  type ReqId = Int
  play  :: String -> Com IAgentCharacter -> IO ReqId
  speak :: String -> Com IAgentCharacter -> IO ReqId
  wait  :: ReqId ->  Com IAgentCharacter -> IO ReqId
  ...etc other methods of IAgentCharacter...
```

Figure 4: Exports from module Agent

Next, the call to createInstance creates an instance of the agent server. The next two lines load the animation file "robby.acs" and create one instance of the character. The curious intermediate value, rob_id, is an artifact of the Agent server design, and not relevant here. In practice we would abstract from this design quirk and define a new function createCharacter as:

```
createCharacter :: String -> Com IAgent
                -> IO (Com IAgentCharacter)
createCharacter agent server =
 do a <- server # load agent
    server # getCharacter a
```

Finally, the character appears in the center of the screen and is asked to speak a phrase. All the AddRef and Release calls are handled implicitly.

5 Why use Haskell?

One can, of course, invoke COM objects from Visual Basic or C++. So is this paper of any interest to a VB or C++ programmer? We believe that it may be, as we argue in this section.

When we program our first example in C++ we see that we need to do a lot more bookkeeping:

```
void main ()
{
 IAgentServer* server    = NULL;
 IAgentCharacter* robby  = NULL;
 HRESULT hr; int reqid; int charid;

 hr = OleInitialize(NULL);
 if (checkHR(hr))
 {
   hr = CoCreateInstance( CLSID_AgentServer, NULL,
   CLSCTX_SERVER, IID_IAgentServer, &server );
   if (checkHR(hr))
   {
     hr = server->load( L"robby.acs", &charid );
     if (SUCCEEDED(hr))
     {
      server->getCharacter( charid, &robby );
     }
     if (checkHR(hr))
     {
       hr = robby->show( &reqid );
       hr = robby->speak( L"Hello world", &reqid );
       robby->Release();
     }
     server->Release();
   }
   OleUnitialize();
 }
}

int checkHR( HRESULT hr )
{
    if (FAILED(hr)) showError(hr);
    return (SUCCEEDED(hr));
}
```

The error checking clutters the code a lot and it is not at

all trivial to be sure to call `Release` or `OleUnitialize` when an error happens. Maybe that is the reason that most C++ programs just leave it out.

For simple scripts, there is hardly any difference between Haskell and say Visual Basic (or Java). Except for the declaration of the variable `Dim Robby` our Agent example looks similar. The COM initialization and finalization is done automatically as are the calls to `Release`.

```
Dim Robby
AgentControl.Connected = True
AgentControl.Characters.Load "Robby",
   ".....\robby.acs"
Set Robby = AgentControl.Characters("Robby")
...
Robby.Move (300,400)
Robby.Show
Robby.Speak "Hello, World!"
...
```

The real difference shows when we want to abstract from commonly occurring patterns in scripts.

5.1 Extending the characters' repertoire

The methods `play` and `speak` are rather limited. We would like to be able to define new, compound method, so that

```
robby # dancesAndSings
```

would make `robby` execute a sequence of `play` and `speak` actions. Here's how we can do that in Haskell:

```
type Action = Com IAgentCharacter -> IO ReqId

dancesAndSings :: Action
dancesAndSings agent =
 do agent # speak "La la la"
    agent # play "Dance"
```

Here we have defined the type `Action` as a shorthand to denote actions that can be performed by an agent (like `play "Dance"` or `dancesAndSings`).

In C++ or Java one could define `dancesAndSings` as the method of a class that inherits from `IAgentCharacter`, using implementation inheritance to arrange to call the character's own `play` or `speak` procedure. To us, it seems rather unnatural to introduce a type distinction between agents that can `dance` and `sing` and agents that can `danceAndSing`. Object oriented languages are good in expressing new objects as extensions of existing objects, functional languages are good in expressing new functions in terms of existing functions. In Visual Basic we could certainly define a procedure like `dancesAndSings`, but than we could only call it using a different syntax than native class methods.

```
Sub DancesAndSings (Byref Agent)
  Agent.Speak ("La la la")
  Agent.Play  ("Dance")
End Sub
...
Robby.Speak ("Hello")
DancesAndSings (Robby)
...
```

If the sequence of actions a particular agent has to perform gets long, it becomes a bit tiresome writing all the "`agent #`" parts, so we can rewrite the definition as a little script, like this:

```
dancesAndSings :: Action
dancesAndSings agent =
   agent # sequence [speak "La la la", play "Dance"]
```

where `sequence` is a re-usable function that executes a list of actions from left to right:

```
sequence :: [Action] -> Action
sequence [a]    agent = agent # a
sequence (a:as) agent =
   do agent # a; sequence as agent
```

Notice that the type of the first argument of `sequence` is a *list* of *functions* that return *I/O performing computations*. The ability to treat functions and computations as first-class values, and to be able to build and decompose lists easily, has a real payoff. In Java, C++, or VB it is much harder to define custom control structures such as `sequence`. For example in Java 1.1 one would use the package `java.lang.reflect` to reify classes and methods into first class values, or use the Command pattern [5] to implement a command interpreter on top of the underlying language. Note that in our case `sequence [...]` is another composite method on agents, just as `dancesAndSings`, and is called in exactly the same way as a native method.

The low cost of abstraction in Haskell is even more convincing when we define a family of higher-order functions to ease moving agents around the screen. First we define a function `movePath` as:

```
type Pos  = (Int,Int)

movePath :: [Pos] -> Action
movePath path agent =
 agent # sequence [ moveTo pos | pos <- path ]
```

Function `movePath path robby` moves agent `robby` along all the points in the list `path`. In Visual Basic (or Java) we can define a similar function quite easily as well by using the built-in `For ... Each ...Next` control structure:

```
Sub MovePath (Byref Agent, Byref Path)
  For Each Point In Path
     Agent.MoveTo (Point)
  Next point
End Sub
```

However, in Haskell we don't have to rely on foresight of the language designers to built in every control structure we might ever need in advance, since we can define our own

custom control structures on demand. Lazy evaluation and higher order functions are essential for this kind of extensibility [7].

We can use function `movePath` to construct functions that move an agent along more specific figures, such as squares and circles:

```
moveSquare :: Pos -> Int -> Action
moveSquare (x,y) width agent =
 agent # movePath square
  where
    w = width `div` 2
    square = [ (x-w,y-w), (x+w,y-w)
             , (x+w,y+w), (x-w,y+w)
             , (x-w,y-w)
             ]

moveCircle :: Pos -> Int -> Action
moveCircle (x,y) radius agent =
 agent # movePath circle
  where
    circle = [ ( x + (radius*cos t)
               , y + (radius*sin t)
               )
             | t <- [0,pi/100..pi]
             ]
```

By re-using `sequence` and `movePath` we were able to define `moveSquare` and `moveCircle` very easily. Because Haskell uses lazy evaluation, the lists of points are generated on demand and therefore never completely in memory.

5.2 Synchronization

The Agent server manages each character as a separate, sequential process, running concurrently with the other characters. Suppose we want one character to sing while the other dances, we just write:

```
do erik  # sings
   simon # dances
```

It looks as if these take place sequentially, but actually they are done in parallel. Each character maintains a queue of requests it has got from the server and performs these in sequence. Hence a call such as `erik # sings` returns immediately, while `erik` is still singing and then makes `simon` dance in parallel.

Now suppose we want `daan` to do something else only when both `erik` and `simon` have terminated: how can we ask the Agent server to do that? The answer is that every `Action` returns a *request ID*, of type `ReqId`, on which any character can `wait`, to synchronize on the completion of that request. Thus:

```
do erikDone  <- erik # sings
   simonDone <- simon # dances
   daan # wait erikDone; daan # wait simonDone
   daan # speak "They're both done"
```

You may imagine that in a complex animation it can be complicated to get all these synchronizations correct. We might easily wait for the wrong request ID, or get deadlocked, or whatever. What we would like to be able to do instead is to say something like:

```
(erik # sings) <|> (simon # dances)
<*>
(daan # speak "They're both done")
```

Here `<*>` is an infix operator used to compose two animations in sequence, and `<|>` composes two animations in parallel. Since all the synchronization is now implicit, it is much harder to get things wrong. We can now say what we want, since we have abstracted away from the details how we have to encode all the low-level synchronization between agents.

How can we program these "animation abstractions" in Haskell?

To perform two animations in sequence, we need to wait until all actions in the first animation are performed before we can start the second. If we assume that an animation returns the request-id of the very last action it performs, we can wait for that one and be sure that all other actions in that animation are also completed. In order to be able to make an animation wait for a request-id, we need to know all characters that will perform in that animation — its *"cast"*. Hence, we represent animations by a pair of an action that returns a request-id, and the cast for that action:

```
type Anim = (IO ReqId, [Com IAgentCharacter])
```

Using type `Anim`, we could (erroneously) try to define sequential composition of two animations as follows:

```
(action1, cast1) <*> (action2, cast2) =
 (action, cast1 `union` cast2)
 where
   action =
    do r1 <- action1
       cast2 `waitFor` r1
       action2
```

Unfortunately, this solution does not work because we can get a deadlock when an agent is part of both animations, in which case it could be waiting for itself to terminate. We therefore take the difference (`\\`) between the casts involved in the two animations.

A more subtle problem occurs when more than two animations are composed in sequence. Suppose we compose three animations thus, `(s1 <*> s2) <*> s3`, and suppose that agent `daan` plays a role in `s1` and `s3` but not `s2`. The deadlock-avoidance device means that `daan` will not wait for `s2` to conclude before starting whatever actions are scripted for him in `s3`. The solution is a little counter-intuitive: in the composition `s1 <*> s2`, make the cast of `s1` who are not involved in `s2` wait for the the cast of `s2` to finish.

Our final (and correct) version of `<*>` will therefore be:

```
(<*>) :: Anim -> Anim -> Anim
(action1, cast1) <*> (action2, cast2) =
 (action, cast1 `union` cast2)
 where
   action =
    do reqid1 <- action1
       (cast2 \\ cast1) `waitFor` reqid1
       reqid2 <- action2
       (cast1 \\ cast2) `waitFor` reqid2
```

The operation `waitFor cast reqid` makes every agent a in its input list `cast` wait on the given request-id `reqid`. Function as `` `waitFor` reqid `` always returns `reqid`.

```
waitFor :: [Com IAgentCharacter] -> ReqId
        -> IO ReqId
[]     `waitFor` reqid = return reqid
(a:as) `waitFor` reqid =
 do a # wait reqid
    as `waitFor` reqid
```

The definition of parallel composition is now easy. We let all the agents of the second animation wait for the first animation to complete and the other way around. Note the nice duality in the implementation of the sequential and parallel combinator: we just swap the middle two statements.

```
(<|>) :: Anim -> Anim -> Anim
(action1, cast1) <|> (action2, cast2)  =
 (action, cast1 `union` cast2)
 where
   action =
    do reqid1 <- action1
       reqid2 <- action2
       (cast2 // cast1) `waitFor` reqid1
       (cast1 // cast2) `waitFor` reqid2
```

In about 20 lines of code we have a very clear definition and implementation of two non-trivial combinators. Using the properties of a pure lazy language we can use equational reasoning to prove various of laws that we expect to hold for the combinators:

```
x <*> (y <*> z)  =  (x <*> y) <*> z
x <|> (y <|> z)  =  (x <|> y) <|> z
       x <|> y   =  y <|> x
```

Proving properties like these is not just a technical nicety! As we have already seen, obtaining correct synchronization among the characters is somewhat subtle, and conducting proofs of properties like these can reveal subtle bugs. This happened to us in practice: when proving the associative law for `<*>`, we discovered that our previous implementation was incorrect in a subtle way.

6 What next?

So far we have described how we may access COM objects from a Haskell program. The obvious dual is to encapsulate a Haskell program as a COM object. We plan to do this next, but there are some interesting new challenges. Chief among these is that a COM object implemented in Haskell must be supported by a Haskell run-time system and garbage-collected heap. While the code might be shared, we would prefer not to create a separate heap for each object; remember a COM object might represent a rather lightweight thing like a button or a scroll-bar. Instead, we would like all the Haskell objects in a process to share the same RTS and heap.

Besides encapsulating a Haskell *program* as a COM object, we also plan to encapsulate a Haskell *interpreter* as a COM object, which implements the `IScriptEngine` interface. This allows us to use Haskell programs to script interactive Web pages

```
<SCRIPT LANGUAGE="HaskellScript">
   do yes <- confirm ("Do you like Haskell?")
      document#write ( if yes then "I knew it!"
                       else "Are you sure?"
                     )
</SCRIPT>
```

or as embedded macro language for MS Office applications such as Word and Excell. Similar implementations already exist for Visual Basic, Java Script, Perl and Phyton.

7 Summary

The theme of this paper is that it is not only *possible* to script COM components in Haskell, but also *desirable* to do so.

We have described a simple way to incorporate COM objects into Haskell's type system, making use of polymorphism to enforce the connection between an IID and the interface pointer returned by `queryInterface`.

We have also shown how one can use higher-order functions, and first-class computations (that is, values of type `IO` τ), to define powerful new abstractions. In the Agent example, we built a little custom-designed sub-language, or combinator library, for expressing parallel behavior. The implementation of the combinators is terse enough that we were able to perform simple algebraic proofs of their properties.

All of this can doubtless be done in any programming language. Our only claim here is that higher-order, typed, functional languages make the job considerably easier.

Acknowledgments

We acknowledge gratefully the support of the Oregon Graduate Institute during our sabbaticals, funded by a contract with US Air Force Material Command (F19628-93-C-0069). Machines and software were supported in part by gifts from Microsoft Research.

References

[1] Kraig Brockschmidt. *Inside OLE (second edition).* Microsoft Press, 1995.

[2] David Chappel. *Understanding ActiveX and OLE.* Microsoft Press, 1996.

[3] Adam Denning. *ActiveX Controls Inside Out (second edition).* Microsoft Press, 1997.

[4] J. Peterson (editor). Report on the programming language HASKELL version 1.4. Technical report, http://www.haskell.org/, April 6 1997.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[6] Object Management Group. *The Common Object Request Broker: Architecture and Specification (revision 1.2).* Object Management Group, 1993. OMG Document Number 93.12.43.

[7] John Hughes. Why Functional Programming Matters. *Computer Journal.* 32(2):98–107. 1989.

[8] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL 20*, pages 71–84, 1993.

[9] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. Green card: a foreign-language interface for haskell. In *Proc. Haskell Workshop*, 1997.

[10] SL Peyton Jones and J Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[11] Microsoft Press. *Automation Programmers Reference*, 1997.

[12] Dale Rogerson. *Inside COM.* Microsoft Press, 1997.

[13] Jon Siegel. *CORBA Fundamentals and Programming.* John Wiley & Sons, 1996.

A Outline of proof of associativity of <*>

In order to prove that <*> is associative, we make some assumptions on the agent implementation.

The first assumption is that the call `as 'waitFor' r` behaves like the identity function with a side effect of letting all agents in as wait for request id `r`. We assume that waitFor has no other visible side effect. It then follows that waitFor distributes over set union:

```
(as 'union' bs) 'waitFor' r =
    do as 'waitFor' r; bs 'waitFor' r
```

or equivalently that waiting is commutative and idempotent:

```
do as 'waitFor' r; as 'waitFor' r =
      as 'waitFor' r
do as 'waitFor' r; bs 'waitFor' r =
   bs 'waitFor' r; as 'waitFor' r
```

The next law states that agents don't have to wait twice in a row:

```
as 'waitFor' r1;
(as 'union' bs) 'waitFor' r2 =
   (as 'union' bs) 'waitFor' r2
```

When there is no interaction between the set of agents that are waiting and the cast of a subsequent action then waiting can be delayed.

```
as 'waitFor' r1; r2 <- action =
   r1 <- action; as 'waitFor' r2
```

Using the above laws plus standard set theory, it follows that <*> is associative.

```
(action1,c1) <*> ((action2,c2) <*> (action3,c3))
```

First, we unfold the definition of <*>

```
do r1  <- action1
   (c2 'union' c3)\\c1  'waitFor' r1
   r23 <- do r2 <- action2
             c3\\c2 'waitFor' r2
             r3 <- action3
             c2\\c3  'waitFor' r3
   c1\\(c2 'union' c3) 'waitFor' r23
```

Next we flatten the sequence of actions

```
do r1 <- action1
   c2\\c1  'waitFor' r1
   c3\\(c1 'union' c2)  'waitFor' r1
   r2 <- action2
   c3\\c2  'waitFor' r2
   r3 <- action3
   r23 <- (c2\\c3) 'waitFor' r3
   c1\\(c2 'union' c3) 'waitFor' r23
```

We rearrange the statements by applying the various swap laws

```
do r1 <- action1
   c2\\c1  'waitFor' r1
   r2 <- action2
   c1\\c2  'waitFor' r2
   c3\\(c1 'union' c2)  'waitFor' r2
   r3 <- action3
   c2\\c3  'waitFor' r3
   c1\\(c2 'union' c3)  'waitFor' r3
```

and introduce nesting again

```
do r12 <- do r1 <- action1
```

```
            c2\\c1 `waitFor` r1
            r2 <- action2
            c1\\c2 `waitFor` r2
    c3\\(c1 `union` c2) `waitFor` r12
    r3 <- action3
    (c1 `union` c2)\\c3 `waitFor` r3
```

so that finally, we can fold the definition of `<*>`

```
((action1,c1) <*> (action2,c2)) <*> (action3,c3)
```