Pugs

Bootstrapping Perl 6 with Haskell

Autrijus Tang
OurInternet, Inc.
autrijus@autrijus.org

Abstract

Perl is a general-purpose language, known for its vast number of freely available extension libraries. The Perl 6 project was started in 2000 to create a more robust runtime environment, and to improve the language's support for multi-paradigmatic programming. However, the attempt to produce a concrete implementation remains a work in progress after 4 years of development.

This paper presents Pugs, a working implementation of Perl 6, written in Haskell. We review the challenges posed by Perl 6's unusual execution model, and how Pugs tackles them using features introduced in GHC 6.4, such as Template Haskell, Software Transactional Memory, and GADTs.

Moreover, we demonstrate how Pugs uses Haskell's FFI to embed Perl 5, Parrot and Haskell within Perl 6 programs, and how it mediates the execution context between these runtime environments.

Finally, we discuss Parrot, a register-based virtual machine designated as the reference implementation of Perl 6's runtime, and how Pugs can evolve from an interpreter to a compiler targeting Parrot.

Categories and Subject Descriptors D.3.m [*Programming Languages*]: Miscellaneous

General Terms Design, Languages

Keywords Perl, Haskell, Parrot

1. Introduction

The Perl language, developed by Larry Wall in the mid-1980s, was an attempt to synthesize the capabilities of the Unix toolset (C, shell, grep, sed, awk, etc.), presenting them in a way that maximizes ease of use, efficiency and completeness.

During the mid-1990s, Wall developed Perl 5, incorporating features from contemporary languages (Scheme, C++, Python, etc.), such as modules, closures and object-oriented programming.

The Perl 6 project was started in 2000, with a focus on improved cross-language interoperation and support for multiple programming paradigms. The design process continued well into 2005, producing the language specification in installments.

Haskell'05, September 30, 2005, Tallinn, Estonia.

However, the Perl 6 implementation, originally planned to proceed in parallel with the design process, failed to materialise. This has seriously impeded development and adoption of the new language.

This paper presents Pugs, an independent implementation of Perl 6 using Haskell as the host language, started in 2005. Haskell proved to be a capable choice for this task: we released the first working interpreter within the first week, and by the third week we had a unit testing framework written in Perl 6.

By having a working implementation of Perl 6, we are able to make the following non-technical contributions:

1. Provide a proving ground to resolve inconsistencies in language design
2. Encourage people to write modules and unit tests for Perl 6
3. Explore more rigorous definitions of Perl 6's execution model

The rest of this paper will proceed as follows. In section 2, we give an overview of the Perl 6 project's history, and of the language's execution model. Section 3 discusses previous implementation efforts. We then present the design of the Pugs interpreter in section 4.

Because Pugs is targeting GHC 6.4 instead of Haskell 98, it takes full advantage of the modern Haskell features GHC provides. In section 5, we review our use of GADTs, STM, as well as reified continuations built on higher-rank polymorphism. In section 6, we demonstrate how Pugs uses FFI and hs-plugins to dynamically load Perl 5, Haskell and Parrot modules at runtime.

Although the naïve tree-reduction interpreter served well for prototyping, it is not efficient enough for production use. In section 7, we discuss the ongoing work to adapt Pugs into a compiler, targeting the register-based Parrot virtual machine. Section 8 outlines our plans for future development.

2. Overview of the Perl 6 Project

Through Perl 5's release, major revisions of Perl were designed and implemented by Wall alone. However, Perl 6's development is driven by the Perl community's varied and often conflicting needs. As such, we will start with a brief overview of the Perl community, and how Perl 6 has come to be.

2.1 Historical Background

During Perl 5's beta development in 1993, a community of volunteers known as the *Perl 5 porters* was formed to port Perl to other platforms. The group's role evolved into maintenance and enhancement. As development continued, Wall appoints individual members of the group to act as *pumpkings* (release managers) to lead the design process and arbitrate patches submitted by other developers.

In 1995, Jarkko Hietaniemi and Andreas König established the Comprehensive Perl Archive Network (CPAN), an infrastructure for sharing freely reusable libraries, programs and documentation. Perl itself comes with a built-in client side utility for automatic downloading, testing and installation of CPAN modules. As of 2005, CPAN has more than 8000 modules, supported by a tool chain that covers smoke testing, issue tracking, cross-referencing, rating and discussion forums.

Driven by scores of Perl 5 porters and thousands of CPAN authors, Perl 5's development puts a strong emphasis on backward compatibility. Developers avoid adding new built-in functions, as they might clash with user-defined functions of the same name. Conversely, deprecated features are almost never fully retracted, and new incompatible features are consistently rejected. For example, one cannot rewrite the reference-counting garbage collector, because existing programs may rely on its object destruction semantics.

In 2000, many developers felt that Perl 5 development was stuck. Some have described the C-based runtime as a tower of Jenga, where new features are constrained to re-use existing semantics, making it ever more fragile.

In response to this situation, Perl developers started the Perl 6 project to rewrite Perl from scratch, dropping language-level backward compatibility for the first time. To use existing Perl 5 programs, one would need to embed a Perl 5 runtime inside Perl 6's runtime, or recompile them into Perl 6.

2.2 Execution Model

The Perl 6 project started with a *requirement analysis* phase. During this two-month period, 26 working groups produced 361 RFC submissions as initial input to the design team, calling for improved functional, object-oriented, concurrency-based, data-driven and logic programming support in the base language. In order to support these requirements, the design team focused on Perl 6's metaprogramming capabilities.

Unlike most languages, Perl 5 has a dynamic grammar: its parser is tightly coupled with its evaluator, so compile-time computation can affect how the parser handles the rest of the program. This allows for a limited form of macros known as *source filters*, where a Perl module can transform the importing program's source code before its parsing.

Many CPAN modules use source filters to implement Perl dialects and embedded domain-specific languages. However, as the transformation operates on the source string level instead of the AST level, multiple filters cannot reliably work together.

Perl 6 proposes an unusual execution model to address this problem. The parser and compiler are implemented in Perl 6, with an initial set of Rules to parse source code into ASTs. Rules are composable, first-class functions, written in an embedded language in Perl 6 that defines parsers for context sensitive, infinite look-ahead grammars.

The parser parses the program with a top-level `program` rule, composed of sub-rules that handle parts of the program. Rules may contain embedded Perl 6 code to produce side effects. For example, the `BEGIN {...}` syntax defines a block of code to be executed during compilation.

Right after parsing a `BEGIN` block, the compiler will generate object code for the block, then immediately run that code. The generated code has access to the compilation environment; it may rewrite the partially produced parse tree, define new grammar rules, or replace the compiler entirely from the next token onwards.

With a default grammar that supports user-defined operators of arbitrary fixity and precedence, programmers can easily introduce new language semantics under this execution model.

During runtime, the program still has access to the built-in parser and compiler modules. The `eval` primitive triggers the compilation and linking process again, with full access to the current runtime environment.

2.3 Language Interoperability

Perl 6 is designed to operate in a mixed-language environment. The Inline.pm framework already implements this capability in Perl 5; it provides a unified interface to three strategies for interoperating with other languages.

1. Translate: Sometimes it is possible to transform the foreign language into equivalent Perl code. In that case, it is a simple matter of applying a source filter, or of `eval`ing the translated code in its own namespace.
2. Native call: Statically compiled languages (such as C, C++ and assembly) are compiled into dynamically loaded shared libraries. Inline.pm automatically generates wrappers to call foreign functions; it also exports Perl's runtime API so the embedded language can call functions defined in Perl.
3. Runtime harness: For languages with a rich runtime (e.g. Java and Python), Inline.pm will run them alongside Perl's own runtime, sharing the execution context in a coroutine-like fashion. Types and global bindings are encoded both ways; the Perl program can invoke a foreign object's methods, and vice versa.

In addition to these strategies, Perl 6 is also designed to run on a shared high-level virtual machine, such as the Java VM or .Net CLR. As part of the Perl 6 project, the Parrot VM is the reference implementation for a cross-language runtime to support Perl 6 and other dynamic languages.

3. Related Work

Perl 6 is designed to be self-hosting, unlike previous versions that were implemented in C. Its execution model demands a mixed parsing, compilation and execution cycle, which complicates the bootstrap process. In this section, we briefly review various bootstrap strategies proposed over the years.

3.1 Perl6::Rules

To explore and prototype Perl 6's Rules mechanism, Damian Conway implemented `Perl6::Rules`, a Perl 5 module that translates Perl 6 Rules into Perl 5's regular expressions. An unreleased prototype was written in 2003, which led to extensive modifications of the design of Rules.

In 2004, Conway released a revised version of `Perl6::Rules`. The implementation was constrained by limitations of Perl 5's non-reentrant regular expression engine; moreover, `Perl6::Rules` relied on various experimental features that no longer work with newer Perl 5 revisions. As such, `Perl6::Rules` remains a partial prototype, not suitable as a basis for implementing a parser for the Perl 6 grammar.

3.2 P6C

The initial plan was to use Perl 5 for bootstrapping: First write the Perl 6 to Parrot compiler in Perl 5, then translate the compiler itself to Perl 6 by instrumenting Perl 5's bytecode compiler to emit Perl 6 code.

Started in 2002, P6C was the first concrete prototype of Perl 6. It is written in Perl 5 and distributed as one of the language prototypes in the Parrot codebase. Instead of `Perl6::Rules`, P6C uses the `Parse::RecDescent` module to construct a recursive-descent parser for Perl 6.

P6C's design is based on multiple passes on the parse tree. In the first pass, `Parse::RecDescent` produces `raw parse objects`:

each node has a `tree` method that turns itself to *op-tree objects*, produced by recursively calling the `tree` method on its subnodes. In the next pass, the compiler traverses the op-tree to annotate each term with its inferred context. Finally, the `val` method is recursively called on each node to produce the compiled code in PIR, a high-level assembly code for Parrot.

With this design, each term's reduction rules are hard-coded in PIR assembly, making it prohibitively difficult to reason about the execution. Because P6C cannot check the resulting assembly code for correctness, subtle bugs often go unnoticed for a long time.

P6C's development continued till 2004, when it was discontinued in favor of PGE. In its final state, it could handle a few builtins, simple expressions, basic rules matching and user-defined functions. It provides no support for `BEGIN` blocks, `eval`, modules and objects. Code generated by P6C cannot interoperate with other languages targeting Parrot.

3.3 PGE and PAST

In July 2004, Perl 6 developers announced a revised bootstrapping plan, based on two new projects:

1. Parrot Grammar Engine (PGE), a compiler that takes Perl 6 Rules and generates the corresponding parser functions in PIR code.
2. Parrot Abstract Syntax Tree (PAST), a higher level representation of PIR programs as Parrot objects.

The developers intend to define Perl 6's grammar as a set of Rules that produce Parrot AST objects. Once PGE is capable of processing those rules, PGE/PAST will effectively become a Perl 6 to Parrot compiler; from there we can make their APIs available via Perl 6 modules, completing the bootstrapping process.

As of this writing, both PGE and PAST are under development: A subset of Python's AST has been ported to Parrot, along with a yacc/lex-based parser for its concrete syntax. PGE can handle most static Rules constructs, but does not support calling out to other functions, so the parser returns only simple match objects, not AST nodes.

Because the Perl 6 grammar has not been specified in Rules, the planned Perl 6 compiler, based on PGE/PAST, is yet to be written.

4. Pugs the Interpreter

When interpreting a Perl 6 program, Pugs first populates an evaluation environment with primitive bindings. The parser takes the initial environment, including the source code string, and computes a modified environment containing the parsed AST. The evaluator then reduces the tree to its final value, in the standard call-by-value order.

4.1 The RuleParser Monad

The Pugs parser is based on Parsec, a monadic parser combinator library. The parser monad is defined as:

```
type RuleParser a = GenParser Char RuleState a
data RuleState = MkRuleState
    { ruleEnv           :: !Env
    , ruleDynParsers    :: !DynParsers
    }
data DynParsers = MkDynParsersEmpty
                | MkDynParsers
    { dynParseOp        :: !(RuleParser Exp)
    , -- ...more parsers...
    }
```

We chose the name `RuleParser` in anticipation of compiling Rules into Parsec combinators in the future, although currently the parser functions are written in Haskell. The `Char` token type is used to combine lexing and parsing into a single step.

Using Parsec's support for user-defined parser states, the parser maintains a `RuleState` containing an evaluation environment. Compile-time bindings, such as the declaration of new types and global functions, update this environment directly.

Perl 6 programs may introduce user-defined operators of arbitrary fixity and precedence, which is an unsolved difficulty in P6C's static parser. Parsec's dynamic nature lets us tackle this problem directly. Instead of using a fixed precedence table, we dynamically compute a table for each token:

```
ruleExpression :: RuleParser Exp
ruleExpression = do
    ops ← operators
    buildExpressionParser ops ruleExpression
operators :: RuleParser (RuleOperatorTable Exp)
operators = do
    env ← gets ruleEnv
    buildOperators (currentFunctions env)
```

To avoid needless computation, we cache all dynamically constructed parsers in the `ruleDynParsers` field, and clear this cache whenever a compile-time evaluation is performed.

4.2 Compile-time Evaluation

Many Perl 6 constructs are defined as code executed during parse time. For example, a simple subroutine definition:

```
sub add1 ($x) { 1 + $x }
```

is defined as being equivalent to:

```
BEGIN { our &add1 := sub ($x) { 1 + $x } }
```

Here the `BEGIN` block is evaluated as soon as it is parsed. It constructs an anonymous subroutine, closing over the current lexical scope, then binds it to a package-wide variable. The `&` sigil indicates that the variable implements storage for functions.

The `BEGIN` block may perform IO actions. For example, we may define a constant function that returns the time it was parsed:

```
sub compiled_at () { BEGIN { time() } }
```

This is similar to the use of `runIO` in Template Haskell:

```
{-# OPTIONS_GHC -fth #-}
compiledAt :: String
compiledAt = $( runIO $
    fmap (LitE . StringL . show) getClockTime )
```

Because the RuleParser monad is *pure*, we need to use `unsafePerformIO` to get to the evaluator:

```
unsafeEvalExp :: Exp → RuleParser Exp
unsafeEvalExp exp = do
    env ← getRuleEnv
    let val = unsafePerformIO $ do
            runEvalIO (evalExp exp)
    return (Val val)
```

The call to unsafePerformIO is *safe*, because we immediately use its result value in the strict constructor `Val`; the result is not reused anywhere else.

4.3 The Eval Monad

The code snippet above uses `runEvalIO` to enter the Eval monad from IO. Inside the Eval monad, it calls `evalExp` to reduce the AST from Exp to Val:

```
runEvalIO :: Env → Eval Val → IO Val
evalExp :: Exp → Eval Val
```

Currently, we define the Eval monad as:

```
type Eval x =
    EvalT (ContT Val (ReaderT Env SIO)) x
newtype EvalT m a = EvalT { runEvalT :: m a }
```

The `SIO` is an abstraction over IO and STM computations; we will discuss it in section 5.2. On top of `SIO`, a `ReaderT` monad transformer provides the evaluation environment, `Env`. We chose `ReaderT` over `StateT`, because `ReaderT` facilitates lexical bindings to sub-terms. Lexical and global symbol bindings are stored as fields in the `Env` record structures:

```
data Env = MkEnv
    { envLexical :: !Pad
    , envGlobal  :: !(TVar Pad)
    , -- ...more fields...
    }
```

As such, sub-terms can override global bindings by writing to the shared `TVar` storage, but new lexical bindings will never leak out.

We implement control flow with the `ContT` monad transformer, because it can directly support Perl 6's continuation and coroutine constructs. We represent the dynamic scope with `resetT`/`shiftT`: calling a subroutine will push a new prompt via `resetT`, so we can simply express the `return` primitive as:

```
doReturn :: Val → Eval a
doReturn = shiftT . const . return
```

Finally, we layer a new transformer EvalT on top of `ContT`, so we can override the monad transformer library's default `callCC`, `ask` and `fail` definitions. For example, in order to provide full stack traces, and support user-defined error propagation logic, `fail` is defined as:

```
fail :: String → Eval a
fail str = do
    pos ← asks envPos
    doReturn (VError str [pos])
```

Evaluation inside the Eval monad is controlled by `evalExp`:

```
evalExp :: Exp → Eval Val
evalExp exp = do
    evl ← asks envEval
    evl exp
```

This makes use of the envEval field in the Env structure, which defines the active evaluator of type `(Exp → Eval Val)`. This allows the user to change the reduction logic during evaluation, such as by adding watch points or profiling instruments.

5. Experimental Features

Pugs depends on the Glasgow Haskell Compiler (GHC), because GHC is the only widely available Haskell implementation, of which we are aware, which offers reasonable performance as a host language. Because of that, we are able to improve clarity and efficiency in the Pugs source code with GHC-specific extensions, such as pattern guards, liberalized type synonyms and generalized derived instances for newtypes.

GHC 6.4 was released in March 2005, one month after Pugs's inception. Pugs dropped support for GHC 6.2 in the same month, as we found it difficult to keep source code portable between GHC 6.2 and GHC 6.4. This was due to incompatible changes in Template Haskell, as well as new module interfaces for Data.Map and Data.Set.

During the three months that followed, we explored GHC's new features to clarify and expand on Perl 6's current design, as demonstrated with prototype implementations in Pugs. Below we discuss three such uses of GHC's experimental features.

5.1 "Tied" Variables with GADTs

Perl 6 distinguishes *value types* from *implementation types*. A container's implementation type is the type of object that implements the container. This design is a generalization of Perl 5's `tie` function, which binds a container to an object that handles all future access to the container.

For example, consider this Haskell code:

```
import Data.HashTable (insert)
-- Insert ("PATH", "/tmp") into x
insert x "PATH" "/tmp"
-- Do the same to y
insert y "PATH" "/tmp"
```

Here both `x` and `y` must bind to a value of the `(HashTable String String)` type; the `insert` for both of them refers to the same function. In contrast, the two lines below do different things in Perl 6:

```
# modify an in-memory storage
%foo{"PATH"} = "/tmp";
# calls setenv() to change the environment
%ENV{"PATH"} = "/tmp";
```

Here the operation is polymorphic: Perl looks up the containers that `%foo` and `%ENV` bind to, then check if they have associated *implementation objects*. If they are, Perl then calls the `STORE` method on them, passing the key and value as arguments.

There is a level of indirection between containers and implementation objects, as the `tie` function can change a container's implementation at runtime. As such, a reference in Perl always points to a container, instead of to a specific implementation object or a value.

Thanks to GADTs [3], Pugs can conveniently represent these levels of indirection, as well as generic get/set operations on containers:

```
type VHash = Map VStr Val
data VRef where
    MkRef   :: (Typeable a) ⇒ !(IVar a) → VRef
data (Typeable v) ⇒ IVar v where
    IHash   :: HashClass   a ⇒ !a → IVar VHash
    IArray  :: ArrayClass  a ⇒ !a → IVar VArray
    IScalar :: ScalarClass a ⇒ !a → IVar VScalar
    -- ...more implementation types...
readIVar :: IVar v → Eval v
writeIVar :: IVar v → v → Eval ()
```

Intuitively, `(IVar VHash)` is a container that can store a `VHash`. However, instead of implementing all operations in terms of `readIVar` and `writeIVar`, the `HashClass` interface defines more methods, with their fallback definitions:

```
type HashIndex = VStr
class (Typeable a) ⇒ HashClass a where
    hash_iType      :: a → Type
    hash_fetch      :: a → Eval VHash
    hash_fetchKeys  :: a → Eval [HashIndex]
    hash_deleteElem :: a → HashIndex → Eval ()
    -- ...more methods...
```

With this, we can use a singleton data type `IHashEnv` to represent the implementation type of the builtin variable `%ENV`:

```
data IHashEnv = MkHashEnv deriving (Typeable)
instance HashClass IHashEnv where
    hash_iType = const $ mkType "Hash::Env"
    hash_fetch _ = do
        envs ← liftIO getEnvironment
        return . Map.map VStr $ Map.fromList envs
```

```
hash_deleteElem _ key = liftIO $ unsetEnv key
-- ...more implementations...
```

Although it is possible to use existential quantification to achieve the same result, GADTs allow for a much more straightforward definition for the `(IVar a)` type. Pugs developers new to Haskell often find GADTs easier to understand, even for regular algebraic data types:

```
data Maybe a where
    Just    :: a → Maybe a
    Nothing :: Maybe a
```

Unfortunately, we could not rewrite all the data types in Pugs as GADTs, as GHC 6.4 is unable to derive Show and Eq for regular data types written this way.

5.2 Composable Concurrency

In 1998, Malcolm Beattie implemented threading support for Perl 5. This design was based on sharing all visible variables across child threads, relying on explicit locks to avoid race conditions.

However, because many programs and modules relied on global variables, this share-everything design caused massive synchronization problems in real-world settings. Moreover, it put locks around all internal access to variables, which hurt runtime performance even when there was only a single thread.

In 2000, Perl 5.6 introduced a new *share nothing by default* threading model, known as *interpreter threads* or *ithreads*. When the program spawns a thread, all global and lexical variables are copied to a new interpreter in the child thread. To share a variable, one needs to mark it explicitly, using the `tie` mechanism to connect it to a storage object visible across threads.

This design presents an unified interface for various underlying threading libraries, including POSIX and non-POSIX threads, and even threads simulated using `fork()` and sockets. The main drawback of ithreads is the overhead of copying the entire interpreter's state upon thread creation. There are plans to use copy-on-write to reduce this overhead, but that is still being implemented.

GHC 6.4 introduced Software Transaction Memory (STM) [2], an abstraction that allows the programmer to define `atomic` functions which are free of deadlocks and races. The Perl 6 code below illustrates this idea with a classic race condition:

```
my $a is shared = 1;
async{ atomically{ my $b = $a; $a = $b+1 } };
async{ atomically{ my $c = $a; $a = $c+1 } };
say $a;     # This always prints 3
```

STM provides an excellent foundation for Pugs to experiment with Perl 6's concurrency features. By using `TVar` instead of `IORef` as containers, we can implement the share-everything model natively without fear of synchronization problems, or adopt the share-nothing model with robust copy-on-write semantics.

To implement `atomically` for Perl 6, we introduce the `SIO` monad, as well as the MonadSTM type class for lifting STM actions:

```
data SIO a =
    MkSTM !(STM a) | MkIO !(IO a) | MkSIO !a
class (Monad m) ⇒ MonadSTM m where
    liftSTM :: STM a → m a
instance Monad SIO where
    return a = MkSIO a
    (MkIO io)   ≫= k =
        MkIO $ do { a ← io; runIO (k a) }
    (MkSTM stm) ≫= k =
        MkSTM $ do { a ← stm; runSTM (k a) }
    (MkSIO x)   ≫= k =
        k x
instance MonadSTM SIO where
    liftSTM stm = MkSTM stm
```

The SIO monad encapsulates three types of strict computations: pure functions (`MkSIO`), actions that modify memory storage (`MkSTM`), and IO actions with full side effects (`MkIO`). We use the `runIO` and `runSTM` functions to convert SIO computations to IO and STM ones. One can safely lift STM actions into the IO monad with GHC's `atomically` function, but casting IO actions to STM will throw an exception:

```
runIO (MkSTM stm) = atomically stm
runSTM (MkIO _)   = fail "Unsafe IO in STM"
```

With this definition, we can implement the `atomically` primitive by imposing a *safe mode* that denies all IO actions inside it:

```
runEvalSTM :: Env → Eval Val → STM Val
runEvalSTM env = runSTM . (`runReaderT` env) .
    (`runContT` return) . runEvalT
opRunAtomically :: Exp → Eval Val
opRunAtomically exp = do
    env     ← ask
    liftSTM (runEvalSTM env $ evalExp exp)
```

The `async` primitive also becomes trivial to implement:

```
runEvalIO :: Env → Eval Val → IO Val
runEvalIO env = runIO . (`runReaderT` env) .
    (`runContT` return) . runEvalT
opRunAsync :: Exp → Eval Val
opRunAsync exp = do
    env     ← ask
    lock    ← liftSTM newEmptyTMVar
    let fork | rtsSupportsBoundThreads = forkOS
             | otherwise               = forkIO
    tid     ← liftIO . fork $ do
        val ← runEvalIO env $ evalExp exp
        liftSTM $ tryPutTMVar lock val
        return ()
    return $ VThread (MkThread tid lock)
```

Prompted by our success in introducing STM to Pugs, Parrot developers are working to integrate STM support into the virtual machine, so Perl 6 programs compiled to run on Parrot can continue to take advantage of STM.

5.3 Reified Continuations

In the previous subsection, we saw how Pugs uses `runEvalIO` and `runEvalSTM` to resume execution from the IO and STM monad. Unfortunately, although both can restore the evaluation environment, they invalidate previously captured continuations by starting a new `runContT`, rendering coroutines unusable across an `async` boundary:

```
coro foo () { yield 1; yield 2 }
async { say foo() }     # fails to resume "foo"
```

To solve this problem, we need to represent the continuation context as data structure. Here we make use of the `CC_2CPST` monad from Oleg Kiselyov et al, based on previous work by Dybvig et al [1].

We further use Kiselyov's `ZipperD` design to represent the evaluation as a traversal on the `Exp` type, keeping the position as a `[Direction]` type. This allows us to re-express the reduction logic as a traversal function over the `Exp` type. We can freely augment the AST with new nodes; each AST node's position is uniquely represented as a `[Direction]`.

Because the new Eval monad will update the cursor at each reduction step, we can define a pair of serialize/deserialize functions:

```
data EvalD = MkEvalD Exp Env [Direction]
reifyEval  :: Eval Val → EvalD
runEval    :: EvalD → Eval Val
```

This *suspend to memory* capability is sufficient to make coroutines work across `async{}` calls. In the future, we may also implement a *suspend to disk* feature, by snapshotting the values of all `TVars` in the `Env` structure into a byte-string image:

```
showEvalD  :: EvalD → STM String
readEvalD  :: String → STM EvalD
```

This has many interesting applications, including web continuations, omniscient debugging, and mobile code. Since Parrot also supports reified continuation objects, we plan to work with Parrot developers to add serialization capabilities as well.

6. Cross-Language Interoperation

New computer languages often suffer from a lack of available libraries. Conversely, programmers are much more likely to adopt a new language if they can reuse libraries from an existing language. As one Perl developer put it: *CPAN is my programming language of choice; the rest is just syntax.*

Therefore, we were strongly motivated to work on Pugs's support for modules written in languages other than Perl 6, as well as making Pugs embeddable from other languages. As most Pugs developers came from Perl 5, Haskell and Parrot backgrounds, those three were the first targets for interoperation.

6.1 Perl 5

Embedding Perl 5 is achieved by linking the `libperl.so` runtime with Pugs. Perl 5 represents all values as `(SV *)` structures, similar to the Val type in Pugs. As both sides allow first-class closure functions, their C-side eval/apply interfaces are straightforward:

```
/* foreign imports to FFI */
SV *perl5_eval (
    char *code,
    void *env, int cxt
);
SV **perl5_apply (
    SV *sub, SV *inv, SV** args,
    void *env, int cxt
);

/* foreign exports from FFI */
extern Val *pugs_eval (
    char *code, int cxt
);
extern Val **pugs_apply (
    Val *sub, Val *inv, Val **args, int cxt
);
```

The import and export APIs are almost identical: both use an optional invocant to denote a method call, allow multiple return values, and take a "context" parameter for Perl's return type polymorphism.

The only difference is the extra `(void *env)` pointer. Pugs passes the evaluation environment `Env` as a StablePtr in calls into Perl 5. When the Perl 5 function calls back, Pugs can then dereference the saved pointer and resume the environment with `runEvalIO`, effectively sharing Pugs's lexical scopes with Perl 5's. Similar to `async{}`'s use of `runEvalIO`, this treatment invalidates captured continuations.

When one side accesses an aggregate container originating from another side, it needs to yield control to the other runtime. We implement this using the `tie` mechanism, described in section 5.1. For example, to fetch an element from an array held in `(SV *)`, Pugs creates an accessor function in the Perl 5 side:

```
-- Opaque pointer for (SV *)
type PerlSV = Ptr ()
instance ArrayClass PerlSV where
    array_fetchVal sv idx = do
        idxSV   ← fromVal $ castV idx
        perl5EvalApply
            "sub { $_[0]→[$_[1]] }"
            [sv, idxSV]
    -- ...more methods...
```

The Perl 5 side, too, can fetch an array element held in Pugs's Val structure with this definition:

```
package pugs::array;
sub FETCH {
    my ($self, $index) = @_;
    pugs::guts::eval_apply(
        'sub ($x, $y) { $x.[$y] }',
        $$self,
        $index
    );
}
```

We further extend Pugs's `use` syntax with a `--perl5` suffix to load Perl 5 modules and import them as Pugs-side classes. Pugs programs can therefore easily use OO modules installed from CPAN:

```
#!/usr/bin/pugs
use DBI--perl5;
my $dbh = DBI.connect('dbi:SQLite:test.db');
my @tables = $dbh.tables;
...
```

Conversely, Perl 5's `Inline.pm` also provides a simple way to call Pugs-side functions from Perl 5 programs:

```
#!/usr/bin/perl
use pugs;   # Here is some Perl 6 code...
sub postfix:<!> { [*] 1..$_ }
sub sum_factorial { [+] 0..$_! }
no pugs;    # Here is some Perl 5 code...
print sum_factorial(3); # 21
```

6.2 Haskell

For compiling and loading Haskell programs at runtime, Pugs uses the `System.Eval.Haskell` module from the hs-plugins package [4]. We started with a trivial `eval_haskell` primitive:

```
evalHaskell :: String → Eval Val
evalHaskell code = do
    ret ← liftIO $
        System.Eval.eval_ code [] [] [] []
    case ret of
        Right (Just x) → return $ VStr x
        Right Nothing  → fail "error"
        Left x         → fail $ unlines x
```

However, this approach gives a rigid type `String` to the Haskell expression contained in `code`. We can change it into a record type that defines a more complex API, but the user code will still need to adjust to that API. Moreover, as `System.Eval` only handles expressions, the user cannot put named top-level functions in `code`.

What we really want is a way to inline Haskell programs with arbitarily typed top-level function definitions, and export them as Perl 6 functions:

```
inline Haskell ⇒ '
```

```
import qualified SHA1
sha1 :: String → String
sha1 = SHA1.sha1
';
say sha1(''); # da39a3ee...
```

We implement this by analysing the function definitions, using the `parseModule` function from `Language.Haskell.Parser`. Once we have the top-level function names and signatures, we can generate wrappers for it with Template Haskell:

```
class (Typeable n, Show n, Ord n) ⇒ Value n where
    fromVal :: Val → Eval n
    castV   :: n → Val
wrapUnary :: String → IO Dec
wrapUnary fun = do
    [(ValD _ body decs)] ← runQ [d|
        name :: [Val] → Eval Val
        name = \[v] → do
            v' ← fromVal v
            return (castV ($(dyn fun) v'))
        |]
    return $ ValD
        (VarP (mkName ("extern__" ++ fun)))
        body decs
```

This handles inlined Haskell functions of type `(Value a, Value b) ⇒ a → b`; we can wrap functions with other arities in a similar fashion.

6.3 Parrot

Started in 2001, Parrot is an attempt of unifying the similar but incompatible designs of dynamic language interpreters, such as Perl 5, Ruby, Python and Tcl. To that end, Parrot offers the following features:

1. Platform-independent bytecode format
2. Portable interpreter written in C, with reasonable performance
3. Alternative fast runloops: direct threaded, JIT and emitting C code
4. APIs for extending and embedding the interpreter
5. One polymorphic boxed object type, known as PMC (Parrot Magic Cookie)
6. Three unboxed value types: Integers, Numbers, Strings
7. First-class continuations that can be stored in PMCs
8. Generational garbage collection system
9. Security contexts: quota, privilege checks

The bytecode format has a concrete syntax, known as Parrot Assembly (PASM). It resembles register-based CPU instructions, with a large number of built-in operators, including both low-level operations on unboxed types, and high-level operations like managing lexical pads, subclassing PMCs, manipulate multi-method dispatch tables, and accessing aggregate PMCs.

Subroutines written in PASM are called in continuation-passing style, via the `invokecc` and `tailcall` op codes, and a calling convention that regulates the allocation of function parameters and return values.

To implement the `eval` function of dynamic languages, the Parrot interpreter keeps a name-indexed table of *compilers* that can turn strings into loadable bytecode. Parrot has a built-in compiler for PIR (Parrot Intermediate Representation), a higher-level assembly language. The PIR compiler handles allocation of named variables, temporary registers and subroutine calls; because of this, practically all higher-level languages targeting Parrot generate PIR code, instead of PASM code.

From Pugs, we link against libparrot and use FFI to hook into the PIR compiler to evaluate PIR programs that take no arguments and returns no values:

```
evalPIR :: String → IO ()
evalPIR code = do
    interp  ← initParrot
    sub     ← withCString code $ \p → do
        parrot_imcc_compile_pir interp p
    withCString "vv" $
        parrot_call_sub_vv interp sub
foreign import ccall "imcc_init"
    parrot_imcc_init :: ParrotInterp
        → IO ()
foreign import ccall "imcc_compile_pir"
    parrot_imcc_compile_pir :: ParrotInterp
        → CString → IO ParrotPMC
```

Here, the `vv` is the calling convention signature that says that both its parameter and return values are the void type. For more complex function signatures, we can parse the PIR subroutine's signature and generate a corresponding wrapper, similar to the way Pugs deals with inline Haskell functions.

Parrot exports a `compreg` operation; at runtime, a program can register the name of a higher-level language, and associate it with a compiler function that turns a string into compiled Parrot code. Since Pugs can generate PIR from Perl 6 source string, we can register a `Pugs` language with a FFI-exported function pointer that delegates the resulting PIR to the builtin PIR compiler:

```
initParrot :: IO ParrotInterp
initParrot = do
    callback    ← mkCompileCallback
        compileToParrot
    pugsStr     ← withCString "Pugs"
        (const_string interp)
    parrot_compreg interp pugsStr callback
compileToParrot :: ParrotInterp
    → CString → IO ParrotPMC
compileToParrot interp cstr = do
    str     ← peekCString cstr
    code    ← doCompile "Parrot" "-" str
    withCString code $
        parrot_imcc_compile_pir interp
```

Pugs can also embed other Parrot-targeting languages, such as Tcl, by loading the Tcl compiler, call it with the embedded Tcl source string, and then invoke the resulting Parrot code object in the same way as it invokes embeded PIR code.

7. Compiling Perl 6

Pugs was initially designed to evaluate the source code directly, without transforming the parse tree into other forms. To turn a Perl 6 program into a native executable, Pugs saves the parsed AST as a Haskell program, then uses GHC to compile it and link with Pugs itself. The resulting executable therefore runs with the same speed as before, saving only the parsing time.

In April 2005, we added two independent compiler backends to Pugs to generate Parrot PIR and Haskell code respectively. They only support a small subset of the Pugs AST, but the generated code runs with speed comparable with Perl 5. This initial result encouraged us to make Pugs a more robust compiler; in this section, we discuss our ongoing work in this area.

7.1 Compiling to PugsAST

Pugs currently simply evaluates the parse tree in the Eval monad, and relies on the `Syn :: String → [Exp] → Exp` data con-

structor to represent loops, conditionals and other statement constructs that do not act as a normal function application. As such, reduction rules for `Syn` terms usually consist of `evalExp` calls to other dynamically constructed terms.

We plan to decouple the program's compile-time behaviour from its runtime behaviour, by separating the original `evaluate :: Exp → Eval Val` function into two steps:

```
compile :: Exp → Compile RunEnv
interpret :: RunEnv → Eval Val

type Compile a = ReaderT CompileEnv IO a
data RunEnv = MkRunEnv
    { runGlobals    :: !Pad
    , runExpression :: !(PugsAST Val)
    }
```

Here the Compile monad rewrites the parse tree of type `Exp` into a `RunEnv`, under the post-parse compile environment `CompileEnv` that contains top-level bindings; all constant folding, term rewriting, type checking and are done in the `Compile` monad.

The resulting `RunEnv` contains a `PugsAST`, a much reduced AST that represents the dynamic part of the evaluation. We use GADTs to capture all the nuances of Perl 6 evaluation strategy:

```
data PugsAST a where
    Stmts   :: [PugsAST a] → PugsAST b
                → PugsAST b
    Var     :: Name → PugsAST Val
    App     :: PugsAST (Code a)
                → Maybe (PugsAST Val)
                → [PugsAST Val] → Cxt
                → PugsAST a
    Lam     :: [[Param]] → PugsAST (Typ a)
                → PugsAST (Code a)
    Lit     :: Val → PugsAST Val
    Pad     :: (Name, PugsAST a) → PugsAST b
                → PugsAST b
    Prim    :: Tag → Dynamic → PugsAST a
                → PugsAST a
```

We can then adapt the existing interpreter code to evaluate `PugsAST` terms instead. Additionally, we can introduce compiler backends that turns the `PugsAST` trees into lower-level syntax trees, such as that of a PIR program.

7.2 Targeting Parrot

Previously, Pugs's prototype PIR compiler backend emits PIR from the parse tree directly, using the `Text.PrettyPrint` library within the `Eval` monad:

```
class (Show x) ⇒ Compile x where
    compile :: x → Eval Doc
    compile x = fail $
        "Unrecognized construct: " ++ show x
instance Compile Exp where
    compile (Syn ";" stmts) = fmap vcat $
        mapM compile stmts
    -- ...more compilation rules...
```

Because the generated `Doc` is not type-checked against PIR's abstract syntax, it could easily generate bogus PIR programs. To address this issue, we need to come up with an abstract syntax data type for PIR, which would look something like this:

```
data RegType = RegInt | RegNum | RegStr | RegPMC
data Rel = R_LT | R_LE | R_EQ | R_NE | R_GE | R_GT
data PIR a where
    RegHard :: RegType → Int → PIR Var
    RegTemp :: RegType → Int → PIR Var
    Ident   :: Ident → PIR Var
    Emit    :: [PIR PASM] → PIR Decl
    TypReg  :: RegType → PIR Typ
    TypPMC  :: Ident → PIR Typ
    Return  :: [PIR Var] → PIR Stmt
    NS      :: Ident → [PIR Decl] → PIR Decl
    Goto    :: Ident → PIR Stmt
    -- ...more PIR instructions...
```

At the time of this writing, the effort to compiling PugsAST into corresponding PIR structures has just started. Aside from the syntax transformation itself, we expect to run into additional issues, such as embedding Haskell and Perl 5 runtime in Parrot; we will update the final version of this paper with our findings on these issues.

8. Conclusion and Future Work

This paper introduced our experience in using Haskell to implement the Perl 6 language. We discussed how modern features in GHC helped to express Perl 6's semantics, which in turn improved our understanding on how to compile Perl 6 to the Parrot virtual machine.

Aside from continuing the various subsystems discussed in the paper, we see some additional directions that Pugs's development can embark on:

1. Tighter integration of GHC (and therefore Pugs) inside the Perl 5 runtime, by improving hs-plugins's support for marshalling Dynamic types as C structures.
2. Compile PugsAST to other backends, such as ANF, C– or back to Perl 5.
3. Run the Pugs codebase on Parrot VM, via a PIR backend for GHC or JHC.
4. Define the RuleParser monad with Perl 6 Rules, by translating the Rules sublanguage into Parsec primitives.

The source code for Pugs is freely available at:

`http://svn.openfoundry.org/pugs/`

For more information about Pugs, please refer to the Pugs homepage at `http://pugscode.org/`.

Acknowledgments

I am deeply grateful to my fellow committers in the Pugs team. Regrettably, I cannot list all 100+ names here; please refer to the AUTHORS file in the Pugs source tree instead. Thanks also to all *lambdacamels* in the IRC channel `irc.freenode.net/#perl6`, as well as to the Perl 6 design team, for their feedback on drafts.

Moreover, I'd like to thank the OSSF project of Institution of Information Science, Academia Sinica, Taiwan, for providing development infrastructure and hosting for the Pugs project.

References

[1] R. K. Dybvig, S. P. Jones, and A. Sabry. A monadic framework for subcontinuations. 2005. Submitted to *Higher Order and Symbolic Computation*.

[2] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. 2005. Submitted to *PPoPP 2005*.

[3] S. P. Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types. July 2004. Submitted to *POPL'05*.

[4] A. Pang, D. Stewart, S. Seefried, and M. M. T. Chakravarty. Plugging haskell in. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 10–21, New York, NY, USA, 2004. ACM Press.