

Exchanging Sources Between Clean and Haskell

A Double-Edged Front End for the Clean Compiler

John van Groningen Thomas van Noort Peter Achten Pieter Koopman Rinus Plasmeijer

Institute for Computing and Information Sciences, Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
{johnvg, thomas, p.achten, pieter, rinus}@cs.ru.nl

Abstract

The functional programming languages Clean and Haskell have been around for over two decades. Over time, both languages have developed a large body of useful libraries and come with interesting language features. It is our primary goal to benefit from each other's evolutionary results by facilitating the exchange of sources between Clean and Haskell and study the forthcoming interactions between their distinct languages features. This is achieved by using the existing Clean compiler as starting point, and implementing a double-edged front end for this compiler: it supports both standard Clean 2.1 and (currently a large part of) standard Haskell 98. Moreover, it allows both languages to seamlessly use many of each other's language features that were alien to each other before. For instance, Haskell can now use uniqueness typing anywhere, and Clean can use newtypes efficiently. This has given birth to two new dialects of Clean and Haskell, dubbed Clean* and Haskell*. Additionally, measurements of the performance of the new compiler indicate that it is on par with the flagship Haskell compiler GHC.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features; D.3.4 [Programming Languages]: Processors - Compilers

General Terms Design, Languages

Keywords Clean, Haskell

1. Introduction

The year of 1987 was a founding one for two pure, lazy, and strongly typed functional programming languages. Clean (Brus et al., 1987) was presented to the public for the first time and the first steps towards a common functional language, later named Haskell, were taken (Hudak et al., 2007).

Clean was conceived at the Radboud University Nijmegen as a *core language* that is directly based on the computational model of functional term graph rewriting to generate efficient code. It also serves as an intermediate language for the compilation of other functional languages (Koopman and Nöcker, 1988; Plasmeijer and van Eekelen, 1993). For these reasons, it deliberately used a sparse syntax (van Eekelen et al., 1990): “... at some points one

can clearly recognize that [...] Clean is a compromise between a functional programming language and an intermediate language used to produce efficient code. For instance, a minimal amount of syntactic sugar is added in [...] Clean.”. Later, the core language was sugared. One particularly important factor was its adoption of uniqueness typing (Barendsen and Smetsers, 1993) to handle side-effects safely in a pure lazy language. Based on this concept, a GUI library (Achten and Plasmeijer, 1995; Achten et al., 1992) was developed, which was used in large applications such as the Clean IDE, spreadsheet (de Hoon et al., 1995), and later the proof assistant Sparkle (de Mol et al., 2002). In 1994, Clean 1.0 appeared, which basically added the syntactic sugar to core Clean that was necessary to develop such large libraries and large applications. In the following years Clean turned open source, and extended its arsenal of functional language features with dynamic typing (Pil, 1999), and built-in generic programming (Alimarine and Plasmeijer, 2002), obtaining Clean 2.1 (Plasmeijer and van Eekelen, 2002). Whenever we refer to Clean in this paper, we mean this version.

Very shortly after the presentation of Clean, Haskell was born as a *concepts language* out of the minds of a large collaboration that idealized an open standard to: “reduce unnecessary diversity in functional programming languages” and “be usable as a basis for further language research”. After three years, this effort resulted in the Haskell 1.0 standard (Hudak et al., 1992) and later the (revised) Haskell 98 standard (Peyton Jones, 2003; Peyton Jones and Hughes, 1999). Early this year, Haskell 2010 was announced and the Haskell' standard is under current active development. Haskell especially enjoyed the benefits of a rapidly growing community; evolving and adapting standards quickly. The downside being that the term ‘Haskell’ became heavily overloaded. It is often not clear to what it refers: one of the standards, a specific implementation of the flagship Haskell compiler GHC, or something in between? Whenever we refer to Haskell in this paper, we mean Haskell 98 and explicate any deviations.

We did not take part in the Haskell collaboration and chose to explore the world of functional programming on our own. After diverging onto different paths more than 20 years ago, we believe it is time to reap the benefits by exchanging (some of) each other's evolutionary results. Both languages have developed interesting language features and concepts (e.g., uniqueness typing in Clean and monads with exceptions in Haskell) and many useful libraries (e.g., the workflow library *iTask* and the testing library *Gast* in Clean, and the parser combinator library *Parsec* and testing library *QuickCheck* in Haskell). Our long-term goal is to facilitate the exchange of such libraries and study the forthcoming interactions between languages features that are distinct to Clean or Haskell. There are many ways to achieve this goal. A naive approach is to define a new functional language that is the union of Clean and Haskell. The resulting language would become very

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Haskell'10, September 30, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0252-4/10/09...\$10.00

baroque due to different syntax in Clean and Haskell for very similar, but not identical, concepts. A second approach is to develop two separate compilers that translate Clean to Haskell and vice versa. This would require an incredible amount of work and is quite hard since features from one language do not always easily project to the other language. This can be simplified by disallowing such features to be used in the libraries under exchange, but that restricts the application of libraries too much. Instead, we develop dialects of Clean and Haskell, dubbed Clean* and Haskell*, that include just enough extra language features to use each other's libraries conveniently. Both new languages are realised in a double-edged front end for the Clean compiler that runs in two modes:

- Clean* mode that accepts Clean 2.1 programs extended with Haskell 98 features.
- Haskell* mode that accepts Haskell 98 programs extended with Clean 2.1 features.

Although Clean and Haskell are both pure and lazy functional languages, there are many subtle differences. An overview of most of the syntactical differences has been given in (Achten, 2007). In this paper we mainly focus on the semantic differences and describe our effort to marry them within the two extended languages. We do not aim to give a complete and detailed overview, but instead identify the biggest challenges and describe the intuition behind their solution and implementation. Concretely, our contributions are the following:

- We identify the most salient differences between Clean and Haskell: modules, functions, macros, newtypes, type classes, uniqueness typing, monads, records, arrays, dynamic typing, and generic functions (Sections 2 to 12).
- With each difference we discuss if and how Clean* and Haskell* support the exchange and briefly explain how this is incorporated in an implementation.
- We provide a concrete implementation of the front end that supports Clean, Haskell, and their dialects Clean* and Haskell*¹.

We give a brief comparison of the current performance of the front end in relation to GHC (Section 13). We end this paper with related work (Section 14) and conclude with a discussion and future work (Section 15).

Since Clean and Haskell are syntactically so much alike, it can be quite hard to disambiguate examples from both languages. Therefore, we choose to start each code fragment with a comment line, `// Clean` or `-- Haskell` respectively, choosing redundancy over opacity. Similarly for the dialects of the languages, we start with a comment line `// Clean*` or `-- Haskell*`.

2. Modules

Clean and Haskell come with many libraries. Instead of migrating these manually, we aim to support the exchange of sources via the front end. It allows Clean modules to import Haskell modules and vice versa. In this section we first briefly compare the two module systems (Section 2.1) and corresponding compilation strategies (Section 2.2). Then we discuss how the front end facilitates mixed compilation of modules in Clean* and Haskell* (Section 2.3).

2.1 Module systems

From the beginning, Clean has used a module system that is very similar to that of Modula-2 (Wirth, 1985). Implementation mod-

ules reside in `.ic1` files and contain all implementations of functions, datastructures, and type classes. Definition modules reside in `.dc1` files and specify the corresponding interfaces by the exported definitions. Besides importing an entire module, Clean allows the explicit import of elements of a module, distinguishing between the sort of element (functions, types, type classes, etc.). This has been included in Haskell* during this project.

Although Haskell 1.0 also used a module system with separate module interfaces, these were abandoned as of Haskell 1.3 because they were increasingly perceived as compiler-generated artifacts, rather than interface definitions (Hudak et al., 2007). Instead, the header of a module enumerates its exported symbols. This perception fits within the language philosophy of Haskell to have the programmer specify only what is required to successfully compile a program. For instance, in Haskell it is allowed to export an identifier `x` in a module `M` but not its type, and to import `x` in another module `N`. Because `x` is not in scope in module `N`, it cannot be given an explicit type. However, the compiler can, and has to, find this type by inspecting module `M`. Haskell prescribes no relation between module names and files, but by convention each module resides in a `.hs` or `.lhs` file. Haskell provides fine-grained control over the names of imported definitions. This is achieved via hiding specific definitions, qualified imports of modules, and hierarchical modules (this last feature is an extension of Haskell). These constructs have been included in Clean* during this project.

User-defined definition modules as used in Clean have as advantage that a programmer obtains a clear description of the offered interface of a specific library module, which is very useful from an engineering point of view. A disadvantage of the approach is that a definition module cannot be used by a compiler to provide additional information about the actual implementation, which might be used for optimizations such as inlining.

2.2 Compilation strategies

When the Clean compiler compiles an implementation module, it is first verified that the exported definitions match the corresponding implementation. Imported definition modules are assumed to match their implementation and an implementation module is only recompiled if it is new, or required by its timestamp. Compilation of modules takes place from top to bottom. When the compiled version of an imported module is up to date, it suffices to inspect only the definition modules of the imported modules, which significantly speeds up the compilation process. Clean modules are compiled to intermediate ABC code (Koopman et al., 1995), from which object code is generated.

The compilation process of a Haskell program is more involved. Because modules can confine themselves to exporting definitions only, but not their types, all sources of imported modules must be available. During compilation, interface files are generated that can be used instead. In the end, object files are generated that are used by a linker to create an executable.

2.3 Mixed compilation

The support of mixing Haskell* or Clean* modules in the Clean compiler is based on definition modules. In the Clean world, these definition modules are still defined separately. The definition module of a Haskell* module is generated by the compiler. When Clean* with Haskell* modules are mixed, the compiler has to switch between compilation strategies: Clean* modules are compiled top down as usual, while Haskell* modules have to be compiled bottom up in order to generate the required definition modules. The compiler has to know with what kind of module it is dealing with. If the module is a `.ic1` file, it is assumed that there is a manually defined `.dc1` file available. Otherwise, if the module is an `.hs` or `.lhs` file, an accompanying `.dc1` file is generated.

¹The front end is under active development, current releases are available via http://wiki.clean.cs.ru.nl/Download_Clean

If a previous compilation of a Haskell* module already generated such a definition module, the new definition module is compared to the old one. If they are identical, the old definition module is kept, leaving its timestamp unchanged. Otherwise, it is replaced by the new definition module. Before a module is compiled, the definition modules of all imported modules have to be available. If these do not exist or are out of date since their timestamp is newer than the one of the definition module, the corresponding Haskell* modules have to be compiled first in order to generate the required definition modules. As we will see in the following sections, generated definition modules from Haskell* modules sometimes include additional information to inform the compiler of typical Haskell* constructs. For efficiency reasons it is sometimes worthwhile to define definition modules of Haskell modules by hand. In Section 6 we see an example where we manually include specialization information in exported function types.

3. Functions

The semantics of the core of Clean is based on term-graph rewriting. The expression that is computed is a computation graph and functions are sugared versions of term-graph rewrite rules. Sharing is explicit in both computation graphs and functions. In Clean, the signature of a function reveals information about its arity, strictness, and uniqueness properties. The first two concepts are discussed in this section, the third in Section 7.

Sharing is explicit in Clean functions. Variable names in function argument patterns, and `case` patterns as well, really point to a subgraph in the computation graph after matching a redex. Multiple occurrences of these variables on the right-hand side of a function and `case` patterns implies that these are shared. Similarly, local graph definitions (i.e., using `let` or `where`) on the right-hand side of a function are also always shared. The local function definitions are always lambda lifted. In all cases, `=` is used as a separator between the left-hand side and right-hand side of a function or local definition. Locally, graph definitions are considered to be constant definitions, and hence, these are shared. If the programmer intends a function of arity zero, this is denoted using `=>` as a separator, or by providing an explicit type signature. Haskell does not explicitly specify what must be shared, but every implementation uses similar rules as stated above. At the top level of a Clean module, every definition is considered to be a function definition. If the programmer intends a constant in applicative form (CAF), this is denoted by using `=:` as a separator. As an example, we define the well-known efficient list of fibonacci numbers as a constant:

```
// Clean
fibs =: [1 : 1 : [x + y \\x <- fibs & y <- tl fibs]]
```

If we used `=` as a separator instead, this would result in recomputing the list for each invocation.

In Haskell, a top-level function without arguments is assumed to be a CAF, unless it has an explicit overloaded type signature. Hence, the above example can be expressed as a function without risk of recomputation:

```
-- Haskell
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

In Clean the programmer can make the tradeoff between (possible) recomputation and space usage. In Haskell this choice is fixed to storing the results and hence usage of space.

The arity of term-graph rewrite rules can be greater than one, in contrast to functions considered from a λ -calculus perspective as chosen by Haskell. For this reason, function signatures in Clean show the arity of their implementation, while signatures are curried in Haskell. The advantage to knowing the arity of a function is efficiency: a function application knows when it is fully saturated. It

is important to observe that this is a syntactic issue: it neither limits the type system nor the use of currying in Clean. As an example, consider the following function that combines the application of the well-known functions `map` and `concat` (named `flatten` in Clean):

```
// Clean
concatMap :: (a -> [b]) [a] -> [b]
concatMap f xs = flatten (map f xs)
```

The function type exposes the arity of the implementation, which is two in this case. Hence, if we change the definition to a point-free notation, the type of the function changes. We use the infix Clean function `o` for function composition, in contrast to Haskell's Prelude `.` notation:

```
// Clean
concatMap :: (a -> [b]) -> ([a] -> [b])
concatMap f = flatten o map f
```

(It should be noted that, as usual, the right-most brackets can be omitted because `->` associates to the right.) Now, the arity of the function is one, which is reflected in its type by the insertion of a function type. Moving the first argument inwards changes the arity of the type again, making it of arity zero:

```
// Clean
concatMap :: ((a -> [b]) -> [a] -> [b])
concatMap = \f -> flatten o map f
```

The parentheses around the function type express that this is a constant function. In Haskell, all these implementations are given the same type, namely:

```
-- Haskell
concatMap :: (a -> [b]) -> [a] -> [b]
```

Consequently, such a type does not reflect the arity of its implementation.

Similar effects occur in the use of type synonyms in function signatures. Suppose that we define the following type synonym:

```
// Clean
:: ListF a b ::= a -> [b]

-- Haskell
type ListF a b = a -> [b]
```

In Haskell, `ListF a b -> ListF [a] b` is also a valid type for any of the implementations of `concatMap`, but in Clean `(ListF a b) -> ListF [a] b` is only valid for the second definition with arity one.

Since its first version, Clean comes with a strictness analyzer (Nöcker, 1994) as well as strictness annotations for function signatures. Strictness information is crucial to generate efficient code. The programmer can add strictness annotations to function arguments, and hence export this information in the corresponding definition module. Haskell has no support for strictness information in function signatures. Clean and Haskell both support strictness annotations in datatypes in very similar ways, therefore this is not discussed.

Exchange Clean* functions can be used easily by Haskell* and vice versa without modification. Haskell* function definitions are interpreted as term graph rewrite rules as described above. In Haskell* function signatures can be given strictness annotations in the same fashion as in Clean*. Strictness information is derived during compilation and exported in the corresponding definition module. Below is discussed how the arity information is derived and exported.

Implementation The issue with function arity shows up in interfaces between Clean* and Haskell* modules. The front end transforms user-provided Haskell* types for exported functions in the

generated definition module and makes the arity of a Haskell* function explicit. Suppose we have the following Haskell* definition of the `concatMap` function:

```
-- Haskell*
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f xs = concat (map f xs)
```

When a Haskell* module exports this function, the front end generates a Clean type for the definition module that reflects the arity of the implementation, which is two in this case:

```
concatMap :: (a -> [b]) [a] -> [b]
```

If we define this function in point-free notation, the arity of the implementation changes and the exported type becomes:

```
concatMap :: (a -> [b]) -> [a] -> [b]
```

Note that in this case, the exported type is syntactically identical to the original Haskell type, but explicitly states that `concatMap f` yields a function value.

Similarly, when a type synonym obscures the arity of a function, its exported type is transformed. Suppose we export the following functions with one identical Haskell* type:

```
-- Haskell*
concatMap2, concatMap1, concatMap0 ::
  ListF a b -> ListF [a] b

concatMap2 f xs =      concat (map f xs)
concatMap1 f      = \xs -> concat (map f xs)
concatMap0        = \f xs -> concat (map f xs)
```

With each version, the type synonym is expanded to match the arity of the implementation of the function. Thus, the definition module contains:

```
concatMap2 :: (a -> [b]) ![a] -> [b]
concatMap1 :: (a -> [b]) -> [a] -> [b]
concatMap0 :: ((a -> [b]) -> [a] -> [b])
```

Only `concatMap2` is strict in its list argument since `concat` and `map` are strict, and the other definitions return functions that still expect one or two arguments.

4. Macros

Clean 0.8 added macros to the language. A macro can be regarded as a function with one alternative and just named arguments. Macros are substituted at compile time, and hence are not allowed to be recursive. Naturally, it may use other recursive functions or define recursive functions locally. Note that the substitution is a graph reduction, and not a textual substitution. For instance, we define a macro to double a value:

```
// Clean
double x := x + x
```

Here, the application `double (fib 100)` is reduced at compile time to `let x = fib 100 in x + x`. Hence, the computation of `x` is shared.

In Haskell, the programmer can use the `INLINE` pragma to encourage the compiler to inline the body of a function. For instance, the above macro is defined as follows in Haskell as a function to be inlined:

```
-- Haskell
{-# INLINE double #-}
double x = x + x
```

Exchange Haskell* modules can import and use Clean macros, and define them using the same syntax. The `INLINE` pragma is not yet included in Clean*. However, macros subsume this concept.

Implementation Currently, it remains future work to export macros from Haskell*.

5. Newtypes

Although type synonyms are useful to document code and explain the purpose of a type, they suffer from the disadvantage that they cannot serve as an instance of a type class or be recursive. Clean's syntax for type synonyms indicates that they are just macros at the type level. Haskell 1.3 introduces `newtype` declarations (i.e., datatype renamings) which are syntactically identical to an algebraic datatype with exactly one constructor of arity one, but which intention is to behave semantically as a type synonym. For instance, here are two newtype definitions:

```
-- Haskell
newtype Nat = Nat Int
newtype Fix f = In (f (Fix f))
```

This eliminates the above mentioned drawbacks: `Nat` can be made an instance of say the type class `Integral`, and `Fix` is clearly a recursive type. The constructors are still included in patterns and construction, but are assumed to be erased by the compiler. Hence, every `Nat` instance behaves as an ordinary `Int` value and every `Fix f` behaves as a plain recursive function.

Clean does not support newtypes. The best approximation is to use an algebraic datatype with a strict argument:

```
// Clean
:: Nat = Nat !Int
:: Fix f = In !(f (Fix f))
```

Operationally, this version is more expensive than a version where these constructors are erased at compile time.

Exchange All Haskell* newtypes can be imported and used in Clean* modules and adhere to the assumed Haskell semantics. The mentioned Clean types are defined as newtypes in Clean* as follows:

```
// Clean*
:: Nat =: Nat Int
:: Fix f =: In (f (Fix f))
```

Note that this code fragment is also legal Haskell*.

Implementation The implementation of newtypes avoids the constructor overhead since all constructors belonging to newtypes are erased at compile time. Removing constructors is not as trivial as it seems. For example, consider the Haskell wrapper function `toNat`:

```
-- Haskell
toNat :: Int -> Nat
toNat = Nat
```

We have to introduce an identity function if the constructor `Nat` is erased. Also, constructors need to be erased from patterns in function definitions:

```
-- Haskell
fromNat :: Nat -> Int
fromNat (Nat _) = 10
```

If we would leave the constructor, the function becomes strict while the semantics requires a nonstrict function. The value `fromNat ⊥` must be rewritten to 10 and not to `⊥`.

Also, the newtypes itself must be erased at compile time in order to make annotations for uniqueness typing on the argument of the newtype effective. The type wrapped in the newtype obtains the type annotations of the newtype definition, instead of the strictness annotation shown earlier. This implies that `Nat` has to be replaced by `Int`. Evidently, this is not possible for recursive newtypes.

6. Type classes

Haskell has supported type classes from the very beginning. Clean, having started as a core language, added type classes to the language with version 1.0 in 1994. There are a number of differences that need to be discussed.

While Clean supports multi-parameter type classes, the parameters of a Haskell type class are restricted to one (although many Haskell implementations allow more parameters). For example, consider the following type class `Array a e` that is used for arrays of type `a` with elements of type `e`, as we will see in Section 10:

```
// Clean
class Array a e where
  createArray :: Int e -> (a e)
  size       :: (a e) -> Int
```

Type classes in Haskell can suggest default implementations for its members that can be overruled in specific instances. For instance in the equality type class:

```
-- Haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  x == y = not (x /= y)
  x /= y = not (x == y)
```

If an instance provides no definition, the default definition is used. In Clean, default members are defined using macros, which are described earlier in Section 4:

```
// Clean
class Eq a where
  (==) :: a a -> Bool

  (/=) x y := not (x == y)
```

The difference with Haskell is that default members via macros cannot be redefined.

In contrast to Haskell, Clean does support defaults on the level of instances. For example, consider the catch-all instance for `Eq`:

```
// Clean
instance Eq a where
  _ == _ = False
```

This instance is used whenever no other instance matches. Consequently, overlap occurs between instances, but this is only allowed on the top level. We cannot define both instances of `Eq` for both `(Int, a)` and `(a, Int)` in Clean.

As we discussed in Section 3, Clean enforces an explicit arity of function type signatures while Haskell types do not reflect the arity of their implementation. Hence, the members of the instances of a Clean type class must agree on their arity as specified by the type class. Instances of a Haskell type class can differ in arity from each other and the original type class definition.

To avoid the overhead of the dictionary-passing style translation of type class, Haskell includes the `SPECIALIZE` language pragma to generate specialized versions at compile time. For instance, in the overloaded equality on lists, we indicate that specialized definitions for `Int` and `Bool` are to be generated and used when possible:

```
-- Haskell
{-# SPECIALIZE eqL :: [Int] -> [Int] -> Bool #-}
{-# SPECIALIZE eqL :: [Bool] -> [Bool] -> Bool #-}
```

```
eqL :: Eq a => [a] -> [a] -> Bool
eqL [] [] = True
eqL [] _ = False
eqL _ [] = False
eqL (x:xs) (y:ys) = x == y && eqL xs ys
```

In Clean, any overloaded function is specialized within module boundaries. Therefore, only exported functions and instances possibly need to be specialized using the `special` keyword in a definition module:

```
// Clean
eqL :: [a] [a] -> Bool | Eq a special a = Int; a = Bool

instance Eq [a] | Eq a special a = Int; a = Bool
```

In contrast to Haskell, such specializations are specified by a substitution instead of the substituted type.

To avoid boilerplate programming, Haskell supports a deriving clause for data or newtype declarations. This relieves the programmer from writing instances of the type classes `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, `Read`, and `Ix` herself, but instead lets the compiler do the job. In Clean, this kind of type-directed boilerplate programming is achieved by generic functions, as we will discuss later in Section 12.

Haskell uses a rather elaborate system of type classes to organize numerical values: `Num`, `Real`, `Fractional`, `Integral`, `RealFrac`, `Floating`, and `RealFloat` for handling values of type `Int`, `Integer`, `Float`, `Double`, and `Rational`. Numeric denotations are overloaded: `0` is of the type `Num a => a` and is in fact the expression `fromInteger (0 :: Integer)`. Therefore, a Haskell programmer needs to add a type signature to disambiguate overloading from time to time. A default declaration provides another approach to disambiguate these cases. This consists of a sequence of types that are instances of the numeric classes. In case of an ambiguous overloaded type variable that uses at least one numeric class, the sequence of types are tried in order to find the first instance that satisfies the constraints. A module has at most one such declaration, and by default it is `default (Integer, Double)`. Clean uses a much simpler approach: numbers are either integer (`Int`) or floating point (`Real`) and their denotations are different: `0` is always of type `Int`, and `0.0` is always of type `Real`. Coercion between these types is achieved explicitly using any of the overloaded functions `toInt`, `toReal`, `fromInt`, or `fromReal`.

Exchange Haskell* supports the less restrictive multi-parameter type classes of Clean. Not only can we import such definitions in Haskell*, we can also define such type classes ourselves and provide instances.

When importing a type class from the other language, the semantics of default members remains the same: Clean* can redefine Haskell default members while Haskell* cannot redefine Clean macros.

The arity of the members of a concrete instance is determined by the importing language. Members of an instance of a Clean type class in Haskell* can be of any arity, while the arity of the members of a Haskell type class in Clean* is the number of arguments.

Specialization in the style of Haskell is not yet implemented. Recall that specialized definitions are generated within module boundaries, similar to Clean.

The type class hierarchy for numerical values in Haskell is available in Clean* as a library. Haskell's types for numerical values are currently not supported in Clean*. However, Haskell* can use Clean's numerical types by prefixing such a value with `'`. The value `'0` is of the Clean type `Int`, just like the Haskell value `0 :: Int`. Similarly, the value `'0.0` is of the Clean type `Real` like the Haskell value `0.0 :: Double`. Proper support for efficient `Float` values in Haskell* is still under active development.

Implementation The front end uses Clean macros to implement default members in Haskell*. The default members can be redefined, but their current form is restricted. A default member in Haskell* must have the same arity as the type it has been given, it can only consist of one alternative, and no infix-style definition is

allowed. Also, such default members cannot yet be exported, this is future work.

Since the arity of members of Haskell instances can differ, the generated definition module of a Haskell* module must include the types of the exported instance members to reflect their arity.

To facilitate efficient implementations of some of the Haskell Prelude functions, Clean includes redefinitions of exported specialized instances and functions. For example, the exported Haskell function that converts `Integral` values has the following signature:

```
fromIntegral :: (Integral a, Num b) => a -> b
special a = Int, b = Double :=> fromIntegralIntDouble
```

Here, we manually include a type signature in the definition module that defers the specialization to a more efficient implementation in `fromIntegralIntDouble`.

Derived instances in Haskell* are automatically included in the generated definition module such that these can be imported from another module. The implementation of the deriving construct in Haskell* is not as straightforward as it may seem. A fixed-point computation is required to determine the context by reduction, if some of the derived instances are already defined but with a more complicated context.

In Clean, CAFs are not allowed to be overloaded since such a value must have a single type in order to be a proper constant. In Haskell, overloaded CAFs without an explicit type signature are allowed, but overloading is resolved at compile time using the monomorphism restriction and the default rule as described earlier. Consequently, the type of an overloaded CAF cannot be determined just using its definition and the types of the functions it uses, but also by the uses of the CAF in the module. Therefore, we may have to type check the entire module before we can determine the type of the CAF. The following implementation is used:

1. The type of a CAF is determined without the monomorphism restriction and default rule. If it is not overloaded, type checking continues in the usual way.
2. If it is overloaded and used by another function, a preliminary type of this function is determined using the overloaded type of the CAF. The type of the use of the CAF, after unification, is remembered. If the function contains more than one use, the types of these are unified. Other CAFs that are used are remembered together with their types.
3. If a function with such a preliminary type is used by another function, this function is typed as if the function used the CAFs remembered in the preliminary type. Hence, a preliminary type is inferred that contains the types of the CAFs that are used (possibly indirectly) by this function. Note that a CAF that uses another CAF is treated in a similar way.
4. The remembered preliminary types of the CAFs are unified to determine their types.
5. All functions for which preliminary types were inferred are type checked again, but now using the no longer overloaded types of the CAFs.

7. Uniqueness typing

Uniqueness typing relies heavily on the fact that sharing is completely explicit in Clean, as discussed in Section 3. A value that is unique has a single path from the root of the computation graph to the value. A function demands such an argument using the `*` annotation in its signature. Function bodies that violate this constraint are not well typed, and hence are rejected during compilation. Values that have a single reference can be updated destructively without compromising functional semantics. This allows Clean to support arrays with in-place updates of its elements, as we discuss later

in Section 10. The programmer can annotate function arguments and datastructures with uniqueness attributes for the same purpose. Uniqueness can also be used to implement I/O, by annotating values that are somehow ‘connected’ with the outside world as being unique, which is discussed in Section 8.

As an example of uniqueness typing, consider a stateful map function, `mapS`, that threads a unique state of type `*s` (type variables need to be attributed uniformly):

```
// Clean
mapS :: (a *s -> (b, *s)) [a] *s -> ([b], *s)
mapS f [] s = ([], s)
mapS f [x:xs] s = ([y:ys], s2)
  where (y, s1) = f x s
        (ys, s2) = mapS f xs s1
```

Actually, the most general type for `mapS` is one that allows both nonunique and unique arguments. The `.` annotation ensures that the same type variable is assigned the same uniqueness attribute:

```
// Clean
mapS :: (.a .s -> (.b, .s)) [.a] .s -> ([.b], .s)
```

The type variable `.a` is either unique or nonunique in the signature, the same holds for `.b` and `.s`. For reasons of presentation, we usually omit these extensive type signatures.

World-as-value programming is supported syntactically in Clean using `#`-definitions, also known as let-before definitions. For instance, `mapS` is preferably written as:

```
// Clean
mapS :: (.a .s -> (.b, .s)) [.a] .s -> ([.b], .s)
mapS f [] s = ([], s)
mapS f [x:xs] s # (y, s) = f x s
                  # (ys, s) = mapS f xs s
                  = ([y:ys], s)
```

Note that this definition is a sugared version of the earlier `mapS` definition using local `where` definitions.

Exchange Haskell* accepts uniqueness typing in Clean style. It can use Clean functions that manipulate unique values. As an example, here is a function that uses Clean I/O to write data to a file using an accumulating parameter:

```
-- Haskell*
writeLines :: Show a => [a] -> *File -> *File
writeLines [] file = file
writeLines (x:xs) file =
  writeLines xs (fwrites (clstring (show x)) file)
```

We use Clean’s `StdFile` library function `fwrites` to write a string to a file and `clstring` to convert a Haskell string to a Clean string (their difference is discussed in Section 10).

Naturally, the uniqueness properties of Haskell* functions need to be verified. Types can be annotated with uniqueness attributes explicitly, or uniqueness information is derived and exported in the corresponding generated definition module. For instance, consider this Haskell* function to update an element in a list:

```
-- Haskell*
updateAt _ _ [] = []
updateAt 0 x (_:ys) = x : ys
updateAt n x (y:ys) = y : updateAt (n - 1) x ys
```

This function can be applied to a list that may contain unique values (`.a`) and preserves the uniqueness of the spine of the list (`u: [.a]`):

```
-- Haskell*
updateAt :: Num n => n -> .a -> u: [.a] -> u: [.a]
```

The uniqueness attributes in this type are identical to those of `updateAt` in Clean’s `StdList` module.

Uniqueness annotations can also enforce constraints. Consider the following function to swap an element in a possibly spine-unique list, instead of updating it:

```
-- Haskell*
swapAt :: Int -> .b -> u:[.b] -> (.b, v:[.b]), [u <= v]
swapAt _ x [] = (x, [])
swapAt 0 x (y:ys) = (y, x:ys)
swapAt n x (y:ys) = (z, y:zs)
  where (z, zs) = swapAt (n - 1) x ys
```

The source and result list now have different uniqueness attributes (*u* and *v* respectively), but they are related in the sense that the uniqueness of the source list is at least as unique as the result list ($[u \leq v]$). In this case it means that from a nonunique source list you cannot construct a unique result list (due to sharing of part of the list spine), but from a unique source list you can construct a nonunique result list.

Implementation The issues that are related to the monomorphism restriction and default rule, as discussed earlier in Section 6, needed to be solved in order to adopt Clean’s uniqueness typing in Haskell*.

8. Monads

Any practical programming language needs to be able to interact with the ‘outside’ world. Clean and Haskell have followed entirely different solutions for this challenge. In Clean 0.8, uniqueness typing has been included to support an explicit multiple environment passing style (i.e., the world-as-value style). In Haskell 1.3, monads were adopted in favor of the stream-based and continuation-based I/O of earlier Haskell versions.

The basic philosophy of monads is that a monadic value represents a recipe that, when performed, may have side-effects and yields a value of some type. Technically, a monad consists of the combinators `return` and `>>=`:

```
-- Haskell
infixl 1 >>=
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

A well-known instance of this class passes a state of type *s* from function to function. The state-passing function is wrapped in the newtype `StateF`:

```
-- Haskell
newtype StateF s b = StateF (s -> (b, s))

instance Monad (StateF s) where
  return x = StateF (\s -> (x, s))
  (StateF f) >>= g
    = StateF (\s -> let (x, s1) = f s
                      (StateF h) = g x
                      in h s1
                )
```

A very similar class `Monad` is defined in Clean:

```
// Clean
class Monad m where
  return :: a -> m a
  (>>=) infixl 1 :: (m a) (a -> m b) -> m b
```

The differences with the Haskell definition are the notation for the fixity of the `>>=` combinator and the explicit arity in the types.

Instead of a newtype for `StateF` we use an algebraic datatype, as described in Section 5. It should be noted that additional uniqueness attributes are required in the right-hand side of `StateF` to allow both *b* and *s* to be unique. We rely on uniqueness typing to ensure a correct single-threaded implementation:

```
// Clean
:: StateF s b = StateF !(s -> .(b, s))

instance Monad (StateF .s) where
  return x = StateF (\s -> (x, s))
  (>>=) (StateF f) g
    = StateF (\s -> let (x, s1) = f s
                      (StateF h) = g x
                      in h s1
                )
```

Monads are used to structure programs. Using this `Monad` class, the function `mapS` from Section 7 is expressed more elegantly:

```
// Clean
mapS :: (a -> m b) [a] -> m [b] | Monad m
mapS f [] = return []
mapS f [x:xs] = f x >>= \y ->
  mapS f xs >>= \ys ->
  return [y:ys]
```

In Haskell, monads are supported syntactically with `do`-notation. Hence we can choose for the definition of `mapS` for a notation similar to the Clean version or the version with `do`-notation:

```
-- Haskell
mapS :: Monad m => (a -> m b) -> [a] -> m [b]
mapS f [] = return []
mapS f (x:xs) = do y <- f x
                  ys <- mapS f xs
                  return (y:ys)
```

The `I0` monad in Haskell is used to sequence I/O operations. The world is hidden from the programmer, and hence there is no danger of violating the single threadedness of this value. In Clean, the world is not hidden from the programmer, and single threadedness is guaranteed by marking them unique. The programmer either chooses to pass these objects explicitly as in the previous section, or to hide the unique object in a monad and pass it implicitly.

The `I0` monad in Haskell also enables exception handling. Its single threadedness ensures a correct binding of exceptions to handlers in a lazy language.

Exchange Monads are integrated seamlessly with uniqueness typing. In the previous section we explained that unique types are available in Haskell*. The `I0` monad, as well as conversions from and to a unique world, is available in Clean* via:

```
-- Haskell*
newtype I0 a = I0 (!*World -> *(a, !*World))
```

Since this is an ordinary type, it is straightforward to pack a unique world in `I0` and to unpack it again.

Implementation The basic transformation scheme from `do`-notation to ordinary monadic constructors is given by Peyton Jones (2003). In order to achieve efficient execution, the code obtained by this transformation needs to be optimized. Currently our implementation of Clean* performs a number of optimizations, such as inlining the member definitions of the `I0` instance for `Monad`.

Also, the exception-handling mechanism is implemented in Clean* and Haskell*. The implementation maintains a stack of exception handlers and dynamically searches for the correct handler if an exception occurs. This makes installation of a handler via a `catch` relatively expensive, but prevents costs during ordinary evaluation.

9. Records

Records were introduced in Clean 1.0. A Clean record is an algebraic datatype of one alternative that does not have a constructor, but a nonempty set of field names. Records are allowed to use the

same (sub)set of field names. For instance, the following declarations happily coexist:

```
// Clean
:: GenTree a = {elt :: a, kids :: [GenTree a]}
:: Stream a = {elt :: a, str :: Stream a}
```

Field values are extracted via pattern matching on the field names or by using a field name as a selector. In case of overlapping field names, a programmer must disambiguate the expression by either providing one distinguishing field name in a pattern (e.g., {elt, kids} and {elt, str}) or by inserting the appropriate type constructor name (e.g., {GenTree | elt} in a pattern or x.GenTree.elt as a selector).

Records are created by exhaustively enumerating all field names or by updating a subset of the field names of an existing record. Here is an example of a function that updates an element of a stream:

```
// Clean
updStream :: Int a (Stream a) -> Stream a
updStream i x s={elt, str}
  | i < 0   = s
  | i == 0   = {Stream | s & elt = x}
  | otherwise = {s & str = updStream (i - 1) x str}
```

Haskell supports records only partially (since Haskell 1.3) in the form of field labels. All arguments of a constructor of an algebraic datatype are either addressed by their position or by field labels. A field label *f* is allowed in several alternatives of an algebraic datatype *T*, provided they have the same type *a*. Every field label brings a new function in scope, named *f* :: *T* -> *a*. For this reason, no two datatypes can use the same field label, even if they have the same result type.

To create a record, the corresponding constructor must be provided and a (possibly empty) set of field labels to be initialized. Any omitted nonstrict field label is silently initialized as \perp . It is illegal to omit strict field labels at initialization. Given a record value, a new record is created by updating a subset of the field labels. As an example, the *Stream* datatype and the *updStream* function look as follows in Haskell:

```
-- Haskell
data Stream a = Stream {elt :: a, str :: Stream a}

updStream :: Int -> a -> Stream a -> Stream a
updStream i x s@(Stream {elt = elt, str = str})
  | i < 0   = s
  | i == 0   = s {elt = x}
  | otherwise = s {str = updStream (i - 1) x str}
```

Exchange We allow both styles of records: a *Clean** program can still define record types with overlapping field names, and a *Haskell** program can define record types with multiple alternatives that use the same field labels. In *Haskell**, it is allowed to import and use *Clean* records. *Clean* record fields are selected with \sim , and the record type can be used to disambiguate field names. For instance, the *Clean* *GenTree* and *Stream* record types can be imported and used in the same *Haskell** module:

```
-- Haskell*
mkGenTree :: GenTree a
mkGenTree = {elt = 0, kids = []}

mkStream :: Stream a
mkStream = {elt = 0, str = mkStream}

rootGenTree :: GenTree a -> a
rootGenTree t = t~GenTree~elt
```

The *mkGenTree* and *mkStream* functions construct a *Clean* record.

Conversely, a *Clean** module can import Haskell records and their field selector functions as well. For instance, a Haskell module that exports the above definition of *Stream* can be used in *Clean**:

```
// Clean*
mkStream :: Stream a
mkStream = Stream 0 mkStream
```

```
hdStream :: (Stream a) -> a
hdStream s = elt s
```

A Haskell record is denoted as a vanilla algebraic datatype. *Clean** does not support the field label syntax at Haskell record value construction.

Implementation The mixed use of *Clean* records in *Haskell** gives rise to several parser issues. Consider the following example:

```
-- Haskell*
analyzeThis = C {elt = 0, kids = []}
```

This is either a normal Haskell record update in which *C* :: *GenTree* *a*, or the function *C* applied to a *Clean* record, but also a data constructor *C* with a *Clean* record of type *Stream* *a*:

```
-- Haskell*
data T a = C (Stream a)
```

In Haskell, the programmer can switch between layout-sensitive and layout-insensitive definitions within a function body. Layout-sensitive mode is assumed when no opening brace is encountered after one of the keywords *where*, *let*, *do*, or *of*. In *Clean*, layout-insensitive mode is switched on or off at the beginning of an entire module, simply by ending the module header with *;* (*on*) or *not (off)*. Hence in *Haskell**, using a local definition that pattern-matches a *Clean* record is very similar to a local layout-insensitive definition. Consider the two following definitions:

```
-- Haskell*
f = (elt, kids) where {elt = 3; kids = []}

g = (e, k)      where {elt = e, kids = [k]} = mkGenTree
```

Here, it can only be determined that a local layout-insensitive definition is given due to the use of *;* and *missing = ... right-hand side*. Currently, *Haskell** allows switching to layout-insensitive mode via *{*, but does not allow switching back.

10. Arrays

Clean has extensive language support for efficient arrays that can be updated destructively due to their uniqueness properties. Arrays with elements of type *a* come in three flavors: lazy (*{a}*, which is the default), strict (*{!a}*), and unboxed (*{#a}*). Since these are different types, array operations are organized as a multi-parameter type class *Array* *a e* where *a* is the array type, and *e* the element type. Array operations are bundled in module *StdArray*. Unboxed array elements can only be basic types, arrays, or records. Note that in *Clean* the *String* type is implemented as an unboxed array of *Char* values, and hence is synonym to *{#Char}*. In Haskell, *String* is synonym to a list of *Char* values.

Clean array values can be created in several ways:

```
// Clean
zeroes :: Int -> .(a Int) | Array a Int
zeroes n = createArray n 0

fibs10 ::          .(a Int) | Array a Int
fibs10 = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}

fibs  :: Int -> .(a Int) | Array a Int
fibs n = {fib i \ i <- [0..n - 1]}
```

All of these functions create an array of type `(a Int) | Array a Int`, where the `.` indicates that the array can be updated destructively. Here, `zeroes n` creates an array, via the `Array` class member function `createArray`, containing `n` zeroes, `fibs10` contains the first ten fibonacci numbers, and `fibs n` uses an array comprehension to construct the first `n` fibonacci numbers. It should be noted that usually the programmer decides for one particular array type (lazy, strict, or unboxed) for efficiency reasons, and uses overloaded versions typically for array libraries.

Arrays are updated destructively. The notation is similar to record updates, but instead of a field label, an index is provided. So, with `a` an array, then `{a & [i] = x, [j] = y}` destructively updates `a` at index positions (starting at zero) `i` and `j` with values `x` and `y` respectively. Array updates can be combined concisely with array comprehensions. For instance, the function `fibs` is defined more efficiently using a lazy array:

```
// Clean
fibs :: Int -> {Int}
fibs n = a
  where
    a = {createArray n 1 & [i] = a.[i - 1] + a.[i - 2]
         \i <- [2..n - 1]}
```

Here, `a.[i]` selects the element at index `i` in array `a`.

Indexes can also be used in patterns, making these either constants or variables. As an example, here is a palindrome checker for arrays:

```
// Clean
isPalindrome :: {e} -> Bool | Eq e
isPalindrome a = size a <= 1 || check (0, size a - 1) a
  where check (i, j) a = { [i] = x, [j] = y } =
        i >= j || x == y && check (i + 1, j - 1) a
```

Haskell provides only immutable arrays via the standard module `Array`. Arrays are implemented as an abstract datatype `Array a b` where `a` is the type of the bounds of the array and must be an instance of the `Ix` class, and `b` is the element type. Haskell lacks denotations for arrays, array patterns, and array selections. Arrays are created using two library functions:

```
-- Haskell
array      :: Ix a => (a, a) -> [(a, b)] -> Array a b
listArray  :: Ix a => (a, a) -> [b]      -> Array a b
```

In both cases, the first argument `(l, u)` defines the bounds of the array and the second argument influences the initial array elements. For `array`, each `(i, x)` in the (finite) list updates the array at index position `i` to value `x`. For `listArray`, the first `u - l + 1` entries from the (possibly infinite) list determine the initial values of the array. In both cases unaddressed positions are initialized with `⊥`.

The `//` operator creates a new array from an existing array:

```
-- Haskell
(//) :: Ix a => Array a b -> [(a, b)] -> Array a b
```

The result array is identical to the source array, except that each `(i, x)` in the list sets the value at index position `i` to `x`.

Exchange The `Array` module has been implemented in `Haskell*` and can be used in `Clean*`. `Haskell*` can import `Clean` arrays and manipulate them with the functions from the `StdArray` module. The `Clean` syntax of array element selection (`a.[i]`) conflicts with Haskell function composition and list notation. Hence, this is not supported in `Haskell*`. Instead, elements are selected with `a?[i]` which selects the element at index position `i` and returns the unaltered array `a`. Alternatively, the `Array` class member function `select` can be used. Also, we can denote `Clean` arrays in `Haskell*`. For instance, `{1, 2, 3}`, `{!1, 2, 3}`, and `{#1, 2, 3}` are legal denotations in `Haskell*`.

Implementation Haskell arrays in the `Array` module are implemented as strict `Clean` arrays:

```
-- Haskell*
data Array a b = Array !(!a, !a) !{b}
```

Due to this strict representation of arrays, all array operations come with strict arguments. Specialized versions of type `Int` are generated and exported, using `special` as discussed in Section 6, for the array operations that are overloaded in the `Ix` class. As an example, here are the exported signatures of `array` and `listArray`:

```
-- Haskell*
array      :: Ix a => !(!a, !a) -> ![(a, b)] -> Array a b
           special a = Int
listArray  :: Ix a => !(!a, !a) -> ![b]      -> Array a b
           special a = Int
```

Also, a distinction is made between arrays that have a zero lower bound and other lower bound values.

11. Dynamic typing

`Clean` supports a `Dynamic` type to wrap values into a black box together with its type, deferring type unification until run time. Haskell has no such feature, but GHC offers the `Data.Dynamic` library for similar but limited purposes. In `Clean`, a value is wrapped in a dynamic using the corresponding keyword:

```
// Clean
wrapInt :: Int -> Dynamic
wrapInt x = dynamic x :: Int
```

The type annotation is only required when polymorphically typed values are wrapped. Unwrapping a value is performed via pattern matching, specifying the expected type:

```
// Clean
unwrapInt :: Dynamic -> Int
unwrapInt (x      :: Int)           = x
unwrapInt (xs     :: [a])           = length xs
unwrapInt ((f, x) :: (a -> Int, a)) = f x
unwrapInt (f      :: A.a: [a] -> Int) = f [1..10]
unwrapInt _                = 10
```

In the second and third arm, `a` is a pattern variable and is unified with the concrete type that is stored in the dynamic value. Multiple occurrences of the pattern variable in the third arm forces unification of the components of the tuple type. In the fourth arm, `a` is universally quantified, and hence the value must be a polymorphic function on lists.

Any value can be (un)wrapped, as long as there is a value representation of its type available. This is guarded by the built-in type class `TC`. For example, consider the following universal wrapping function:

```
// Clean
wrap :: a -> Dynamic | TC a
wrap x = dynamic x
```

The context in which this function is used determines the type that is stored in the dynamic with the value. Analogously, unwrapping a value can depend on the type that the context requires:

```
// Clean
unwrap :: Dynamic -> Maybe a | TC a
unwrap (x :: a^ ) = Just x
unwrap _                = Nothing
```

Here, the type of the context determines with which type the dynamic content is unified. This is indicated by postfixing a type pattern variable with `^`, which ‘connects’ it with the type variable occurring in the type of function.

Exchange Since Haskell does not support dynamic typing like Clean, we only have to consider the effects of Clean’s dynamic typing in Haskell*. The type `Dynamic` and type class `TC` are imported via the module `StdDynamic` in Haskell* since these are built in. When a Clean function is used that returns a dynamic value, the Haskell* module has to be able to denotate such values. Therefore, it supports the `dynamic` keyword. For instance, we are able to define the `wrap` function in Haskell* as follows:

```
-- Haskell*
wrap :: TC a => a -> Dynamic
wrap x = 'dynamic x
```

The keyword is escaped using a `'` to avoid any naming conflicts with similarly named definitions in Haskell. Also, we can unwrap a value in a dynamic pattern match in Haskell*:

```
-- Haskell*
unwrap :: TC a => Dynamic -> Maybe a
unwrap (x :: a~) = Just x
unwrap _         = Nothing
```

Implementation Since the Clean compiler already supports dynamic typing, the implementation did not pose many challenges. The only issue arisen in the Haskell parser was due to the use of the `::` annotation which is obligatory when wrapping polymorphic values. It conflicts with Haskell where any expression can be annotated with a type using the same notation. For example, consider the following expression:

```
-- Haskell*
wrappedId :: Dynamic
wrappedId = 'dynamic id :: A.a: a -> a
```

It is unclear whether the type annotation is part of Clean’s dynamic typing system or Haskell’s expression. Whenever the parser recognizes the `'dynamic` keyword, the subsequent type annotation is part of the dynamic value. Otherwise, the type annotation is part of the expression.

12. Generic functions

Clean supports generic programming as advocated by Hinze (2000) which was adopted in Clean in 2001. The style of programming is very similar to Generic Haskell (Löh et al., 2003). Generic programming is used to avoid boilerplate programming, for essentially the same purpose as instances can be derived automatically for type classes in Haskell, as discussed in Section 6. Haskell has no language support for generic functions.

A generic function is a recipe that is defined in terms of the structure of datatypes, rather than the datatypes themselves. The key advantage is that there are only a few structural elements from which all custom datatypes can be constructed. For algebraic datatypes, the programmer needs to distinguish alternatives, products of (empty) fields, and basic types. As an example, here is an excerpt of the generic definition of equality:

```
// Clean
generic geq a :: a a -> Bool
geq{|Int|}      x          y          = x == y
geq{|UNIT|}    UNIT      UNIT      = True
geq{|EITHER|}  fx _ (LEFT x1) (LEFT x2) = fx x1 x2
geq{|EITHER|}  _ fy (RIGHT y1) (RIGHT y2) = fy y1 y2
geq{|EITHER|}  fx fy _     _         = False
geq{|PAIR|}    fx fy (PAIR x1 y1) (PAIR x2 y2) = fx x1 x2
                                                    && fy y1 y2
```

Note that this is not a single function definition, but rather a collection of function definitions that are indexed by a type constructor. They also do not need to reside in the same module, but can be defined anywhere provided that the generic type signature is in scope.

If the programmer wishes to have an instance of equality for her custom type, say `GenTree` and `Stream` defined in Section 9, then this is expressed as:

```
// Clean
derive geq GenTree, Stream
```

Such derived functions are exported in the same fashion.

A kind annotation is always provided for a generic function. For instance, if we wish to test two general trees `x` and `y` for equality, we write `geq{|*|} x y`. Naturally, overloaded equality can be defined as a synonym of the generic variant:

```
// Clean
instance Eq (GenTree a) | geq{|*|} a where
  x == y = geq{|*|} x y
```

The programmer can deviate from the generic recipe if she wishes. In that case, the generic function is specialized for a specific type. Suppose that two general trees are identical if they have the same elements when visiting the tree in left-first depth-first order:

```
// Clean
geq{|GenTree|} fx x1 y2 = length e1 == length e2
                        && and (zipWith fx e1 e2)
  where (e1, e2)        = (elts x1, elts y2)
        elts {elt, kids} = [elt : concatMap elts kids]
```

The `fx` parameter is provided by the generic mechanism and is the generic equality for the element types of the generalized tree. This specialization is exported using the `derive` syntax.

Exchange Haskell does not have any built-in support for generic functions, therefore, we only consider using Clean’s generic functions in Haskell*. Since every use of a generic function requires a kind annotation, Haskell* supports such annotations. When importing a generic function like `geq` in a Haskell* module, an instance for a Haskell* datatype is derived using the `derive` keyword. For similar reasons as `'dynamic` in Section 11, this keyword is escaped:

```
-- Haskell*
data BinTree a = Leaf a | Node (BinTree a) a (BinTree a)

'derive geq BinTree
```

We are even able to define generic functions in Haskell*. The earlier definition of `geq` remains the same, only its signature changes:

```
-- Haskell*
'generic geq a :: a -> a -> Bool
```

An escaped keyword is now used and the type no longer reflects the arity of its definition. Exporting generic functions and their derivations from a Haskell* module is not yet implemented.

Implementation The implementation did not pose any challenges since Clean already includes support for generic functions.

13. Performance

Although the implementation of the front end is not yet complete, it is already possible to compile a large class of Haskell programs to efficient code. We have compared the current implementation of the double-edged front end for the Clean compiler with GHC 6.12.2 by running the complete Haskell benchmark programs of Hartel (1993). We modified the `parstof` program slightly to prevent GHC from optimizing the program. It is intended that the computation is performed 40 times instead of once. To obtain good measurable execution times some of the input sizes of the programs were increased. Our benchmark environment used IA32 code on a computer with an AMD Opteron 146 2Ghz processor running the Windows XP X64 operating system.

Program	Front end (s)	GHC (s)	Ratio	Front end		GHC -O
				GC	Heap	
complab	0.81	1.03	0.79	c	8M	-H8M
event	0.64	1.23	0.52	c	32M	-H32M
fft	0.36	0.78	0.46	c	64M	-H64M
genfft	0.72	1.37	0.53	m	400K	
ida	0.84	0.87	0.97	c	16M	-H16M
listcompr	0.11	0.25	0.44	m	400K	
listcopy	0.11	0.26	0.42	m	400K	
parstof	0.23	0.19	1.21	m	8M	-H8M
sched	2.78	1.84	1.51	m	12M	
solid	0.81	1.11	0.73	c	4M	-H4M
transform	0.91	1.28	0.71	m	400K	
typecheck	0.77	0.86	0.90	m	400K	
wang	0.55	0.64	0.86	m	100M	-H100M
wave4	0.53	0.72	0.74	m	10M	-H10M

Table 1. Execution times of Haskell using the front end and GHC

The results are shown in Table 1. The columns show the name of the program, the execution times in seconds (elapsed wall clock time including startup), the ratio of execution times (comparing the execution time of GHC executables to the front end executables), and the provided options for the generated executables. For the front end we specify what garbage collector was used to obtain the best performance ('c' is the combination of a copying and compacting collector and 'm' is the combination of a marking and compacting collector) and the maximum heap size. With GHC we used the '-O' optimization option and for the executables that required larger heaps we used '-H' with the same heap size as for the Clean executables for the GHC executables, but only if this improved the performance.

All benchmarks are single-module Haskell programs. Hence, GHC cannot obtain an advantage by cross-module optimization over our compiler. Since the current implementation of the front end is work in progress, not all planned optimizations are implemented yet. When these optimisations are implemented we will study the benchmarks and the reasons behind the observed differences in-depth. Currently, the benchmarks just show that our compiler achieves competitive results.

14. Related work

Already in Fortran, the first programming language that offered functions, it was realized that it is sometimes convenient to use foreign functions, for instance to improve efficiency by directly using assembly functions. Soon after other languages were introduced, there was the desire to use parts of other programs. There are many programming languages that offer such interpretability, usually realized by a foreign function interface (FFI). A typical FFI offers a possibility to annotate a function as `external`. Then, the compiler assumes that the external function exists. It is the task of the linker to include that external function, which is compiled by the compiler of its host language, to the code generated for the program.

It is evident that this approach to exchange sources between languages imposes huge restrictions on the compiler as well as the language. Not only must the stack layout of both languages be identical, but also the memory layout of all datastructures used. For instance, both languages must use the same precision for integers, and layout for records and multidimensional arrays. An example of an issue in the interface is that Fortran starts array indices by one, while most modern languages starts array indices at zero. Moreover, the array dimensions in Fortran are reversed compared to languages like C. Hence the array declaration $A(n, m)$ in Fortran matches $A[m][n]$ in C. The element $A(i, j)$ in Fortran matches $A[j-1][i-1]$. To overcome such kind of problems, many languages

offer interface types which mimic their counterpart in the external language.

Both Haskell (Chakravarty, 2003) and Clean offer the possibility to exchange sources with C. Moreover, both languages offer support for using functions via this interface, GreenCard for Haskell and HtoClean for Clean. Exchanging sources between Clean and Haskell via this interface is very unattractive. The interface puts severe restrictions on the types that can be used. For instance, there is no notion of type classes and higher order functions, and parameterized recursive datatypes cause all kinds of problems. Also, such an interface is completely unsuited for lazy evaluation since this is not supported by C.

Since C is a subset of the C++, every valid C program is also valid C++. Hence, every compiler for C++ accepts C, which makes interoperability between these two languages very easy. Such an approach is not applicable for our purposes since Clean nor Haskell is a subset of the other.

The Microsoft .NET Framework supports multiple programming languages and focuses on language interoperability. It contains special designed languages like C#, F# and J#, as well as support for standard languages like Python and Lisp. Some alternative and free implementations of parts of this framework are Mono, CrossNet and Portable.NET. Since Haskell nor Clean is designed for such a framework this approach is not suited for our needs. Moreover, these frameworks are based on an object-oriented view of the world and have limited support for features in modern lazy functional languages.

There is some work to translate Haskell to Clean in order to obtain Haskell programs with the speed of Clean programs. First, Hegedus (2001) translated Haskell structures to Clean. Next, Diviánszky (2003) implemented a partial compiler from Haskell to Clean based on these concepts. Hackle (Naylor, 2004) is a compiler from a restricted subset of Haskell 98 to Clean. This compiler actually achieved performance gain compared to GHC for a number of programs. Although each of these approaches studied translating Haskell to Clean, the exchange of language features from both languages was not considered.

There are a number of stand-alone Haskell implementations. The flagship compiler GHC supports the complete Haskell 98 standard, as well as a wide variety of language extensions. Hugs 98 provides an almost complete implementation of the standard, but unfortunately the last release dates from 2006. Nhc 98 is a small compiler that is compliant to the standard, its last release stems from 2007. Yhc branched from Nhc 98, but is not yet a complete Haskell 98 compiler. The recent UHC supports almost the complete standard and adds several experimental language extensions. None of these Haskell compilers has support for interoperability with Clean.

15. Discussion and Future Work

In this paper we have described what it takes to exchange sources between Clean and Haskell. We discussed most of the differences in language features and the required extensions of both Clean and Haskell to denote them. This has resulted in two dialects, dubbed Clean* and Haskell* respectively. Also, we briefly explained how their exchange is facilitated in a concrete implementation. We have seen how some of the language features go together nicely hand-in-hand (e.g., uniqueness typing and monads), while others lead to subtle conflicts (e.g., records).

Besides the exchange of sources, the front end supports the exchange of features to a certain extent as well. Haskell programmers can now use uniqueness typing, dynamic typing, and generic functions. Clean programmers can use constructs like newtypes. Additionally, the front end comes with benefits for both Haskell and Clean programmers. For instance, Haskell programmers can use

the full-fledged IDE including project manager. Also, performance of compiled Haskell programs looks promising: on a par and for computation-intensive applications often slightly better than GHC. For Clean programmers, it is nice that their work becomes more easily accessible to the large Haskell community.

Although the most important features of Haskell 98 have been implemented, the list of remaining issues is still rather long since some features took much more work than expected. When we started this project about three years ago, we knew that Haskell is a more baroque language than Clean. But only after digging into the details of the language we discovered that Haskell was even more complicated than anticipated. For instance, since Haskell makes heavy use of overloading and monads, more effort was needed to retain the efficiency that Clean is well known for. Also, the number of Haskell libraries which are really Haskell 98 compliant is rather limited. To enable the practical reuse of Haskell libraries, we have to implement some of GHC's extensions, such as generalised algebraic datatypes and type families. This is challenging, not only in terms of the programming effort, but more because of the consequences it will have on features such as uniqueness typing. We believe this double-edged front end provides an excellent research and implementation laboratory to investigate these avenues.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. This work has been partly funded by the Technology Foundation STW through its project on "Demand Driven Workflow Systems" (07729).

References

- Peter Achten. Clean for Haskell 98 programmers - A quick reference guide. <http://www.st.cs.ru.nl/papers/2007/achp2007-CleanHaskellQuickGuide.pdf>, 2007.
- Peter Achten and Rinus Plasmeijer. The ins and outs of Concurrent Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.
- Peter Achten, John van Groningen, and Rinus Plasmeijer. High level specification of I/O in functional languages. In John Launchbury and Patrick Sansom, editors, *Proceedings of the 5th Glasgow Workshop on Functional Programming, GFP '92, Ayr, UK*, Workshops in Computing, pages 1–17. Springer-Verlag, 1992.
- Artem Alimarine and Rinus Plasmeijer. A generic programming extension for Clean. In Thomas Arts and Markus Mohnen, editors, *Selected Papers of the 13th International Workshop on the Implementation of Functional Languages, IFL '01, Stockholm, Sweden*, volume 2312 of *Lecture Notes in Computer Science*, pages 168–186. Springer-Verlag, 2002.
- Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems (extended abstract). In Rudrapatna Shyam-sundar, editor, *Proceedings of 13th Conference on the Foundations of Software Technology and Theoretical Computer Science, FSTTCS '93, Bombay, India*, volume 761 of *Lecture Notes in Computer Science*, pages 41–51. Springer, 1993.
- Tom Brus, Marko van Eekelen, Maarten van Leer, and Rinus Plasmeijer. Clean: a language for functional graph rewriting. In Gilles Kahn, editor, *Proceedings of the 3rd International Conference on Functional Programming Languages and Computer Architecture, FPCA '87, Portland, OR, USA*, pages 364–384. London, UK, 1987. Springer-Verlag.
- Manuel Chakravarty. The Haskell 98 Foreign Function Interface 1.0, an addendum to the Haskell 98 report. <http://www.cse.unsw.edu.au/~chak/haskell/ffi>, 2003.
- Péter Diviánszky. Haskell-Clean compiler. http://aszt.inf.elte.hu/~fun_ver/2003/software/HsCleanA112.0.2.zip, 2003.
- Marko van Eekelen, Eric Nöcker, Rinus Plasmeijer, and Sjaak Smetsers. Concurrent Clean (version 0.6). Technical Report 90-20, Radboud University Nijmegen, 1990.
- Pieter Hartel. Benchmarking implementations of lazy functional languages II - Two years later. In John Williams, editor, *Proceedings of the 6th International Conference on Functional Programming Languages and Computer Architecture, FPCA '93, Copenhagen, Denmark*, pages 341–349. ACM Press, 1993.
- Hajnalka Hegedus. Haskell to Clean front end. Master's thesis, ELTE, Budapest, Hungary, 2001.
- Ralf Hinze. A new approach to generic functional programming. In Tom Reps, editor, *Proceedings of the 27th International Symposium on Principles of Programming Languages, POPL '00, Boston, MA, USA*, pages 119–132. ACM Press, 2000.
- Walter de Hoon, Luc Rutten, and Marko van Eekelen. Implementing a functional spreadsheet in CLEAN. *Journal of Functional Programming*, 5(3):383–414, 1995.
- Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María Guzmán, Keving Hammond, John Hughes, Thomas Johnsson, Richard Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell, a non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5):1–164, 1992.
- Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In Barbara Ryder and Brent Hailpern, editors, *Proceedings of the 3rd Conference on History of Programming Languages, HOPL III, San Diego, CA, USA*, pages 1–55. ACM Press, 2007.
- Pieter Koopman and Eric Nöcker. Compiling functional languages to term graph rewriting systems. Technical Report 88-1, Radboud University Nijmegen, 1988.
- Pieter Koopman, Marko van Eekelen, and Rinus Plasmeijer. Operational machine specification in a functional programming language. *Software: Practice & Experience*, 25(5):463–499, 1995.
- Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In Colin Runciman and Olin Shivers, editors, *Proceedings of the 8th International Conference on Functional Programming, ICFP '03, Uppsala, Sweden*, pages 141–152. ACM Press, 2003.
- Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers - Sparkle: a functional theorem prover. In Thomas Arts and Markus Mohnen, editors, *Selected Papers of the 13th International Workshop on the Implementation of Functional Languages, IFL '01, Stockholm, Sweden*, volume 2312 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, 2002.
- Matthew Naylor. Haskell to Clean translation. Master's thesis, University of York, 2004. <http://www-users.cs.york.ac.uk/~mfn/hacle/hacle.pdf>.
- Eric Nöcker. *Efficient functional programming - Compilation and programming techniques*. PhD thesis, Radboud University Nijmegen, 1994.
- Simon Peyton Jones, editor. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.
- Simon Peyton Jones and John Hughes. *Report on the programming language Haskell 98*. University of Yale, 1999. <http://www.haskell.org/definition/>.
- Marco Pil. Dynamic types and type dependent functions. In Kevin Hammond, Tony Davie, and Chris Clack, editors, *Proceedings of the 10th International Workshop on the Implementation of Functional Languages, IFL '98, London, UK*, volume 1595 of *Lecture Notes in Computer Science*, pages 169–185. Springer-Verlag, 1999.
- Rinus Plasmeijer and Marko van Eekelen. *Functional programming and parallel graph rewriting*. Addison-Wesley Publishing Company, 1993.
- Rinus Plasmeijer and Marko van Eekelen. Clean language report (version 2.1). <http://clean.cs.ru.nl>, 2002.
- Niklaus Wirth. *Programming in MODULA-2 - 3rd, corrected edition*. Texts and Monographs in Computer Science. Springer-Verlag, 1985.