Helium, for Learning Haskell

Bastiaan Heeren Daan Leijen Arjan van IJzendoorn
Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
{bastiaan, daan, afie}@cs.uu.nl

Abstract

Helium is a user-friendly compiler designed especially for learning the functional programming language Haskell. The quality of the error messages has been the main concern both in the choice of the language features and in the implementation of the compiler. Helium implements almost full Haskell, where the most notable difference is the absence of type classes. Our goal is to let students learn functional programming more quickly and with more fun. The compiler has been successfully employed in two introductory programming courses at Utrecht University.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (Functional) Programming*; D.3.4 [**Programming Languages**]: Processors—*Compilers, Interpreters*

General Terms

Design, Human Factors, Languages, Measurement

Keywords

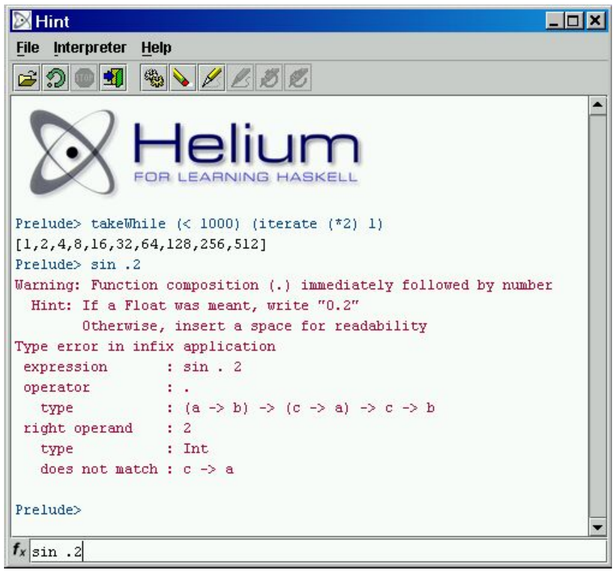
learning Haskell, error messages, type inference, education, error logging

1 Introduction

Helium [17] is a user-friendly compiler designed especially for learning the functional programming language Haskell. Our goal is to let students learn functional programming more quickly and with more fun. This is quite a challenge! The subtle syntax and sophisticated type system of Haskell are a double edged sword – highly appreciated by experienced programmers but also a source of frustration among beginners, since the generality of Haskell often leads to cryptic error messages.

Our experience in teaching functional programming to students was the motivation to develop a better learning environment for Haskell. Helium has a host of features that help to achieve this goal.

- Helium generates warnings and hints to warn for common programming mistakes and to stimulate good programming practices. Although an experienced programmer might be annoyed by a warning about a missing type signature, it is very helpful during a course on functional programming.
- We use a sophisticated type checker to improve type error messages. The type checker performs a global constraint analysis on the type inference graph to determine errors more accurately than the usual bottom-up algorithm.
- Helium can optionally log compiled programs to a central server. We have used this during introductory courses and have logged literally thousands of programs produced by participating students. We analyze these programs to determine the kind of mistakes that beginning programmers make, and use this collection to tune our heuristics in the compiler.
- Helium uses a wide range of heuristics to suggest improvements in the case of an error. For example, a probable fix is reported for missing function arguments and for misspelled identifiers.
- Helium implements a proper subset of Haskell 98. It supports almost full Haskell where the most notable difference is the absence of type classes. Other changes are less profound and include a simpler layout rule and a more restricted syntax for operator sections. We have mixed feelings about leaving out type classes: it improves the quality of error messages a lot, but also forces a different style of programming as many common Haskell functions are overloaded. In the end, we felt that in an educational setting it is paramount to have the highest quality error messages, but at the same time we are currently investigating the addition of a restricted form of type classes for which good error messages are still possible.
- Helium uses the Lazy Virtual Machine (LVM) [10] to execute the compiled code. When an exception occurs, the LVM not only shows the exception itself, but also the chain of demand that led to the exception, helping the student with debugging. Besides the standard exceptions like unmatched patterns, the LVM also checks all integer arithmetic against overflow, and detects many forms of unbounded recursion.
- A simple but effective graphical interpreter is built on top of the compiler (see Figure 1). On demand, it can jump to an



```
Prelude> takeWhile (< 1000) (iterate (*2) 1)
[1,2,4,8,16,32,64,128,256,512]
Prelude> sin .2
Warning: Function composition (.) immediately followed by number
  Hint: If a Float was meant, write "0.2"
        Otherwise, insert a space for readability
Type error in infix application
 expression       : sin . 2
 operator         : .
   type           : (a -> b) -> (c -> a) -> c -> b
 right operand    : 2
   type           : Int
   does not match : c -> a

Prelude>
```

```
sin .2
```

Figure 1. A screenshot of the Hint interpreter

error location in a user configurable editor, and it uses color to distinguish clearly between different kinds of errors and program output.

Helium is still ongoing work and is not a complete Haskell system. For example, it only implements the Haskell prelude and lacks other standard libraries. However, Helium has been used successfully during two introductory courses at the Utrecht University, and we have had very good experiences with it as an environment for teaching Haskell.

In this article, we compare Helium with other Haskell implementations. For the sake of presentation, we only consider two widely used implementations, namely GHC(i) and Hugs. GHC is an industrial strength compiler for Haskell, where much effort has been put in the type checker to generate high quality error messages. The Hugs interpreter is a more experimental system with generally lower quality error messages than GHC, but we decided to include it as it is still widely used in educational settings.

The paper is structured as follows. We start with a collection of examples where we compare the warnings and error messages produced by the Helium compiler with the messages reported by GHCi and Hugs. Section 3 discusses the implementation of the compiler. In Section 4, we talk about our experiences with the compiler during an introductory course on functional programming, and about the logging facility. Section 5 mentions related work, and Section 6 concludes this paper.

2 Examples

In this section we demonstrate Helium's error messages by giving a large number of examples. We start with warnings and hints that are reported for potentially dangerous parts of a program. Next, we show examples containing syntax errors (Section 2.2) and type errors (Section 2.3). Section 2.4 discusses the advantages of having a simplified type system without overloading, and finally, Section 2.5 considers error messages as they can occur at run-time.

2.1 Warnings and hints

In addition to rejecting incorrect programs, Helium also warns about potential mistakes. Furthermore, hints can be given that suggest fixes to certain problems. Since warnings and hints are based on heuristics, one must be careful when adding them: a wrong hint may be more confusing than giving no hint at all. Helium contains a logging facility that we have used to determine the kind of mistakes that are often made by students in practice. Based on these results we have added a range of warnings and hints that apply to these situations. It is beyond the scope of this paper to describe the log results in detail, but we describe some preliminary results in Section 4.

In this section we look at a few interesting examples of common warnings and hints. For example, take the following program.

```
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter p [] = []
myFIlter p (x:xs) =
        if p x
         then x : myFilter p xs
         else myFilter p xs
```

Although the program is legal Haskell and accepted by other interpreters without warnings, the Helium compiler spots many potentially dangerous constructions.

```
(4,1): Warning: Tab character encountered;
       may cause problems with the layout rule
  Hint: Configure your editor to replace tabs by spaces
(3,1) : Warning: Missing type signature:
        myFIlter :: (a -> Bool) -> [a] -> [a]
(2,10): Warning: Variable "p" is not used
(2,1), (3,1): Warning: Suspicious adjacent functions
              "myFilter" and "myFIlter"
```

First of all, we see that Helium can emit multiple warnings and that it gives precise source locations: line and column. Actually, the compiler maintains the entire error *range*, that can be used by a development environment to highlight the offending terms inside an editor. The last warning shows that messages can be attributed to multiple locations. A development environment could use this information to enable a user to jump from one location to the other.

The first warning is very helpful in education: layout errors due to invisible tabs are a source of frustration among students. The second warning about the missing type signature is typical for education and the suggested type signature is presented in such a format that it can be pasted directly into a program. For students, it is a good practice to write down type signatures, but in this particular case, the warning is caused by a typo. The last warning points directly to this hard to find problem: the name of the function in the last clause is spelled wrong. Note that variants of this error were registered several times by our logging facility.

For two adjacent definitions where only one definition has a type signature, Helium calculates an *edit distance* between the identifiers. The edit distance is the minimal number of edit operations that need to be performed to transform one variable into the other. Edit operations are the insertion of a character, the deletion of a character, and changing the case of a character. When the edit distance is low enough, the Helium compiler assumes that a typo is made and issues a warning.

The same mechanism is used to detect typos in the usage of variables and functions. Here is another student program that contains a typo.

```
maxLen :: [String] -> Int
maxLen = maximun (map length xs)
```

This program is incorrect since it contains two undefined variables.

```
(2,10): Undefined variable "maximun"
  Hint: Did you mean "maximum" ?
(2,30): Undefined variable "xs"
```

The first error contains a *hint* that the identifier is very similar to another variable in scope, namely `maximum`.

Sometimes, the hints are very specific for educational purposes. A common mistake among students is to write floating point numbers incorrectly.

```
test = sin .2
```

Syntactically, this is a legal Haskell program since the `(.)` is interpreted as an operator. Hugs, for example, just gives a confusing type error message for this program.

```
ERROR "lex1.hs" (line 1): Unresolved top-level overloading
*** Binding             : test
*** Outstanding context : (Floating b, Num (c -> b))
```

Note that the overloading of integer literals makes the error message hard to understand. Helium also gives a type error, but first warns about the dangerous use of a dot in front of a number.

```
(1,13): Warning: Function composition (.) immediately
        followed by number
  Hint: If a Float was meant, write "0.2"
        Otherwise, insert a space for readability

(1,13): Type error in infix application
 expression       : sin . 2
 operator         : .
   type           : (a -> b) -> (c -> a) -> c -> b
 right operand    : 2
   type           : Int
   does not match : c -> a
```

Hints like these are very helpful for students. In this case, the warning is maybe geared too much towards education as it names the dot operator as *function composition*, while in principle this operator could be redefined by the user. We consciously took the decision to target the messages to beginning programmers, but we plan to add compiler flags that allow Helium to adjust its messages according to the user's level of expertise.

2.2 Syntax errors

As a first step in learning Haskell, students have to become familiar with the syntax of Haskell. It is most helpful for students when messages about syntax contain exact position information. Furthermore, we use the parser combinator library Parsec [11] that maintains not only the location of parse errors but also a list of terms that would have been legal at that point in the input. The lexical analyser also catches some common mistakes that normally lead to cryptic parse errors. The following example demonstrates this.

```
test :: [(Int, String)]
test = [(1, "one"), (2, "two"), (3, "three")
```

In this example, the programmer forgot to add a bracket to close the list. Our logs indicate that the illegal nesting of parentheses, braces, and brackets is a very common lexical error. When lexing a program, Helium keeps a stack of brackets in order to identify these errors accurately.

```
(2,8): Bracket '[' is never closed
```

Unfortunately, GHCi does not maintain column information and is imprecise in the location of the syntax error. In addition, it supplies a misleading hint about the indentation.

```
syn3.hs:3: parse error (possibly incorrect indentation)
```

For this particular example, the message produced by Hugs is also interesting since it points to an unexpected '}'. This character does not occur in the program, but was inserted by the compiler to handle the layout rule.

```
ERROR "syn3.hs" (line 3): Syntax error in expression (un
expected '}', possibly due to bad layout)
```

We continue with a second example. Consider the following fragment of code.

```
remove :: Int -> [Int] -> [Int]
remove n [] = []
remove n (x:xs)
   | n = x     = rest
   | otherwise = x : rest
 where rest = remove n xs
```

The intended equality operator in the guard is confused with a '=': the guard should read `n == x`. Helium spots the mistake directly after the guard when a second '=' character is seen. The following error message is given.

```
(4,16): Syntax error:
    unexpected '='
    expecting expression, operator, constructor operator,
    '::', '|', keyword 'where', next in block (based on
    layout), ';' or end of block (based on layout)
```

The following parse error is reported by GHCi.

```
syn4.hs:4: parse error on input '='
```

Hugs produces a similar error message for this example. Note that the Helium error message is more accurate in two respects. Firstly, the precise location clarifies which of the two '=' characters was not expected by the parser. Secondly, the error message lists exactly what terms would have been legal at that point in the input.

2.3 Type errors

Our logging facility shows that most errors made in practice are type errors. Helium contains a constraint-based type inferencer that tries to improve on the cryptic messages produced by most implementations. An important feature of the type checker is that it maintains the entire inference graph of a program. When a type error occurs, Helium can inspect the type inference graph to produce a clear and precise error message. For instance, the type inferencer takes user-supplied type signatures into account to determine which part of the program should be reported as incorrect. Consider the following program.

```
makeEven :: Int -> Int
makeEven x = if even x then True else x+1
```

Here, the `if` branches contain a `Bool` and an `Int` expression. Guided by the type signature, Helium concludes that the `Bool` expression must be wrong, since there is more evidence that the return type of the conditional should be `Int`.

```
 (2,29): Type error in then branch of conditional
 expression       : if even x then True else x + 1
 term             : True
   type           : Bool
   does not match : Int
```

When the type signature is not taken into account, the error can be attributed to the other branch. For example, the type checker in Hugs, which suffers from a left-to-right bias and which does not use the type signature, gives an opposite result.

```
ERROR "tp2.hs" (line 2): Type error in conditional
*** Expression     : if even x then True else x + 1
*** Term           : x + 1
*** Type           : Int
*** Does not match : Bool
```

GHCi does not maintain column information. Unfortunately, it is not clear from the type error message to what branch the error is attributed, as the entire function body is shown.

```
tp2.hs:2:
    Couldn't match ‘Int’ against ‘Bool’
        Expected type: Int
        Inferred type: Bool
    In the definition of ‘makeEven’:
        if even x then True else x + 1
```

Type information about the complete program can be a big advantage, especially if infinite types are detected. Take for example the following (slightly modified) example from the Edinburgh type checker suite [19].

```
test = \f -> \i -> (f i, f 2, [f,i])
```

The Helium compiler reports that the elements in the list are not consistent.

```
(3,34): Type error in element of list
 expression       : [f, i]
 term             : i
   type           : Int
   does not match : Int -> a
```

Because of the type information from the rest of the program, there is no need to mention infinite types in the type error message. However, both Hugs and GHCi report that unification would give an infinite type. For instance, the following message is produced by Hugs.

```
ERROR "tp6b.hs" (line 3): Type error in application
*** Expression     : f i
*** Term           : i
*** Type           : a -> b
*** Does not match : a
*** Because        : unification would give infinite type
```

Besides the global approach, the type inferencer also contains heuristics to suggest hints and fixes to the program. For example, consider the following program.

```
test = map [1..10] even
```

The student has accidently given the arguments of `map` in the wrong order. Again, the logged student programs show that this is indeed a common mistake.

```
(1,8): Type error in application
 expression       : map [1 .. 10] even
 term             : map
   type           : (a -> b) -> [a]           -> [b]
   does not match : [Int]    -> (Int -> Bool) -> c
 probable fix     : re-order arguments
```

Helium uses a *minimal edit distance* algorithm to determine how terms can be changed to satisfy the type inferencer. Examples include the reordering of function arguments and the elements of a tuple, and the insertion or removal of function arguments. A correction is only suggested if it completely removes the type inconsistency. Also note that the (uninstantiated) type signature of `map` is given in the error message, nicely aligned to the inferred type. In contrast, GHCi chooses one of the function arguments.

```
tp4.hs:1:
    Couldn't match ‘a -> b’ against ‘[t]’
        Expected type: a -> b
        Inferred type: [t]
    In an arithmetic sequence: [1 .. 10]
    In the first argument of ‘map’, namely ‘[1 .. 10]’
```

The error message given by Hugs suffers from another problem.

```
ERROR "tp4.hs" (line 1): Type error in application
*** Expression     : map (enumFromTo 1 10) even
*** Term           : even
*** Type           : b -> Bool
*** Does not match : [a]
```

Since Hugs does not maintain complete source information, the arithmetic sequence has disappeared in the error message and is presented as `(enumFromTo 1 10)`.

Type synonyms assign an intuitive name to a complicated composed type. Unfortunately, type synonyms are often unfolded during the process of type inference, resulting in type errors that refer to unnecessary complex types. Helium stores information about type synonyms directly in the inference graph. Whenever possible, the error messages are given in terms of type synonyms instead of an unfolded type. This is especially important for domain-specific combinator libraries that may contain complicated type synonyms. For example, in a second year course we use a parser combinator library that defines a `Parser` type as a synonym to abstract over a function type – it is much better to get error messages in terms of `Parsers` than to see the underlying function types. Here is a simple example that contains `String` literals.

```
test :: String
test = xs : "def"
     where xs = "abc"
```

And indeed, the Helium error message is in terms of strings.

```
(2,11): Type error in constructor
 expression       : :
   type           : a      -> [a]    -> [a]
   expected type  : String -> String -> String
 probable fix     : use ++ instead
```

In contrast, both Hugs and GHCi forget the type synonym, and give their error in terms of the unfolded type. GHCi reports the following.

```
tp7b.hs:2:
    Couldn’t match ‘[Char]’ against ‘Char’
        Expected type: [[Char]]
        Inferred type: [Char]
    In the second argument of ‘(:)’, namely ‘"def"’
    In the definition of ‘test’: xs : "def"
```

2.4 Overloading

At the moment, Helium does not support overloading. In general, the type inferencer can produce better error messages when overloading is not present. Although it is undoubtedly a powerful feature of Haskell, we felt that for educational purposes it is better to have the best possible error messages. As an example of the problems associated with overloading, we return to an earlier example, where we replace the character lists with integer lists.

```
test = xs : [4, 5, 6]
     where xs  = [1, 2, 3]
```

The Helium message suggests (again) using concatenation.

```
(1,11): Type error in constructor
 expression       : :
   type           : a     -> [a  ] -> [a]
   expected type  : [Int] -> [Int] -> b
 probable fix     : use ++ instead
```

However, Haskell 98 prepends an implicit call to `fromInteger` to all integer literals. On the type level, this means that every integer literal is implicitly promoted to a type in the `Num` class. In GHCi, this leads to a rather confusing message as it tries to promote the integers to lists.

```
tp7.hs:1:
 No instance for (Num [a])
 arising from the literal ‘6’ at tp7.hs:1
 In the list element: 6
 In the second argument of ‘(:)’, namely ‘[4, 5, 6]’
```

Hugs does not do much better.

```
ERROR "tp7.hs" (line 1): Unresolved top-level overloading
*** Binding             : test
*** Outstanding context : (Num b, Num [b])
```

Note that this is no critique of the respective systems: giving good error messages in the presence overloading is known to be very difficult. However, the examples show that students are immediately exposed to complex parts of the Haskell type system, even when they are not consciously using those features.

The implicit promotion of integer literals appears to be the main cause of complex type errors. We consider the following example from the Edinburgh type checker suite [19] once more.

```
test = \f -> \i -> (f i, f 2, [f,3])
```

The program is ill-typed since the function `f` is erroneously placed in the same list as the literal `3`.

```
(2,34): Type error in element of list
 expression       : [f, 3]
 term             : 3
   type           : Int
   does not match : Int -> a
```

In Haskell, however, the type inferencer tries to find a `Num` instance for function types. GHCi reports the following.

```
tp6.hs:2:
    No instance for (Num (t1 -> t))
    arising from the literal ‘3’ at tp6.hs:2
    In the list element: 3
    In a lambda abstraction: (f i, f 2, [f, 3])
```

Hugs resolves type class constraints somewhat later. Information about the origin of the type class constraints is lost.

```
ERROR "tp6.hs" (line 2): Unresolved top-level overloading
*** Binding             : test
*** Outstanding context : (Num b, Num (b -> c))
```

Forcing students to write type signatures is not the solution to this problem. Let us add a type signature to the previous example.

```
test :: (Int -> a) -> Int -> (a,a,[Int -> a])
test = \f -> \i -> (f i, f 2, [f,3])
```

Even though the intended type of the list is known, there is still the possibility of an instance of `Num` for function types. GHCi reports the following.

```
tp6a.hs:3:
    Could not deduce (Num (Int->a)) from the context ()
    Probable fix:
        Add (Num (Int -> a)) to the type signature(s)
        for ‘test’ or add an instance declaration
        for (Num (Int -> a))
    arising from the literal ‘3’ at tp6a.hs:3
    In the list element: 3
    In a lambda abstraction: (f i, f 2, [f, 3])
```

Although the type error messages benefit from a type system without overloading, there are some drawbacks. Functions that are normally member of a type class have in Helium different names for different instances. For instance, to test for equality, `eqInt` and `eqBool` are available, but also `eqList`, which is given the type `(a -> a -> Bool) -> [a] -> [a] -> Bool`. Similarly, variants of the `show` function exist. A `show` and `eq` function are automatically derived for each type constructor that is introduced by a data type declaration or type synonym. Arithmetic and comparison operators are all specialized to work for integers. Therefore, `(+)` has type `Int -> Int -> Int`, and `(/=)` has type `Int -> Int -> Bool`. To obtain the variant that works for floats, a period is postfixed. For example, `(+.)` and `(/=.)` are defined by Helium’s standard Prelude. Special type inference heuristics suggest a fix if the wrong variant of the operator is used.

A second disadvantage is that it is harder to write and to use polymorphic functions that normally make use of overloading, such as `sort` and `nub`. In case of the function `nub`, an equality function must be passed explicitly as an additional argument. This function now has the type `(a -> a -> Bool) -> [a] -> [a]` (which is the type of Haskell’s function `nubBy`). Nonetheless, we believe that students are more aware what is going on when they have to pass these additional functions themselves. For education, this may be preferred over the (invisible) dictionaries that are inserted by the compiler.

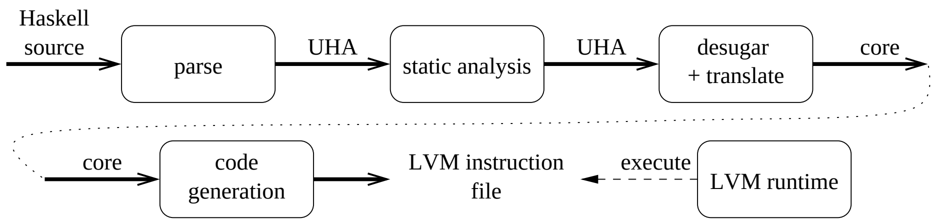


Figure 2. Pipeline of the Helium compiler

Given the disadvantages, we are currently investigating how we can add type classes to Helium while still maintaining the quality of our error messages. A promising direction is the introduction of 'closed world' style type classes [13] in combination with type specification files [9].

2.5 Runtime errors

Finally, we consider runtime errors. Most runtime errors are caused by non-exhaustive pattern matches. Take for example the following function definition.

```
inverseHead xs = case (head xs) of
                    0 -> 0
                    x -> 1/x
```

When we evaluate the expression (`inverseHead []`) in GHCi, we get the following response.

```
Program error: {head []}
```

This message is rather uninformative since it only mentions the `head` function and not the demand trace that led to the error. The Helium runtime maintains the demand trace and reports it.

```
exception: Prelude.head: empty list.

trace:
  demanded from "Run1.inverseHead"
```

In this example, the trace is rather simple, but in general the information is very helpful to students trying to debug their course assignments. The trace facility reflects the dynamic demand structure and is not nearly as sophisticated as dynamic debuggers like Hat or Freya [2]. However, we are currently investigating the integration of Buddha [12] into Helium.

The runtime further performs as many checks as possible without penalizing execution speed too much. For example, all arithmetic is checked against exceptional situations such as overflow and division by zero. Take for example the following program.

```
test :: Bool
test = sqr 12345678 > 0
     where sqr x = x*x
```

The expression (`sqr 12345678`) will overflow when 32 bit integers are used. An exception is returned instead of an unexpected result.

```
exception at "LvmLang.*":
  integer overflow.

trace:
  demanded from "LvmLang.>"
  demanded from "Run3.test"
```

3 Implementation

In this section we briefly discuss interesting parts of the implementation. We first present a general overview on how the compiler is implemented (Section 3.1), and then explain in more detail how we tackle the problem of type inference.

3.1 A research platform

We have put a lot of effort into making the implementation of the Helium compiler as simple and modular as possible. One of the goals of the Helium compiler is to facilitate research into new functional languages or language features. We therefore implemented the compiler naively with no hidden dependencies and with modular internal interfaces. All intermediate data types have a concrete syntax that can be communicated to files.

Figure 2 shows how the Helium compiler works internally. First, Haskell source code is parsed using the Parsec [11] combinator library. After parsing, the program is represented using a data type, called UHA (Unified Haskell Architecture), that closely resembles the concrete syntax of Haskell. It is important to retain the link to the concrete syntax at this point in order to give error messages in the same terms as the user wrote them – desugaring is postponed until all static checks have been made. Exact source ranges are stored in the nodes of the tree to facilitate the integration of the compiler with a development environment. UHA also covers Haskell features that are not supported by Helium, such as type classes, universal and existential quantification, and records.

Static analysis (including type inference) is then performed on UHA. Static checks analyze the program to catch mistakes like undefined identifiers. Many of those checks need the same kind of information during their analysis, for example, the current variables in scope. This information must somehow be attributed to the nodes of the UHA data type. A common solution in Haskell is to either pass this information during each transformation as an argument, or to extend the data type with extra fields. Unfortunately, the first solution leads to more complex code as a transformation pass has to compute different aspects within the same code. The second solution separates these aspects, but may lead to inconsistencies when the tree is transformed.

We solved this dilemma by using an attribute grammar system [14] that acts as a pre-processor to Haskell. We can now specify different aspects and attributes orthogonally to each other, and use them seamlessly for different static checks. The attribute grammar system takes these specifications and glues them together into a standard Haskell module. This aspect oriented approach makes it much easier to change certain parts of the compiler without having to modify unrelated code. Currently, a third-year student is adding attribute specifications to detect pattern matches that are not exhaustive. Especially for such a student project it is important to be

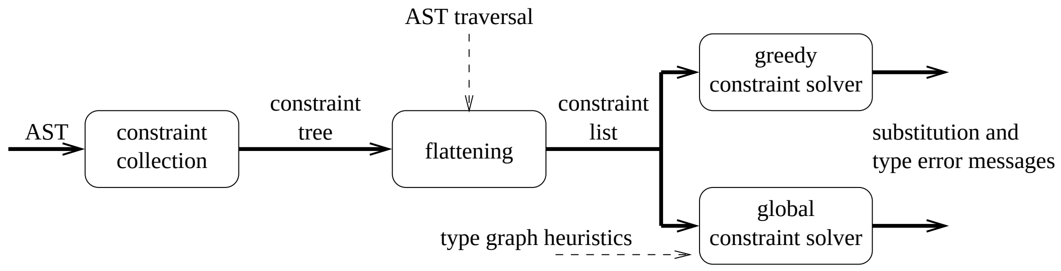


Figure 3. Constraint-based type inference

able to add code without having to understand every detail of the compiler.

After the checks are done, the UHA is translated into an enriched lambda calculus language, called *Core*. It is closely related to the core language of GHC. The main differences are that it is untyped and that it can contain structured meta-information to encode extra compiler dependent information, like types or specialized function instances. Furthermore, Core was designed separately from Helium as a general backend for (lazy) functional languages. As such, it does not contain any Haskell specific parts nor specific Helium dependent features. Currently, we have a fourth year student that performs Core to Core optimizations using Stratego [18], a program transformation system with programmable rewrite strategies.

Finally, the Core language is translated by a standard library into instruction files for the Lazy Virtual Machine (LVM) [10]. Just like the Java Virtual Machine, the LVM defines a portable instruction set and file format. However, the instruction set is specifically designed to execute non-strict, higher order languages. The interpreter itself is implemented in portable C (using the excellent OCaml runtime system), and it runs on many platforms, including Windows, various Unix's, MacOS X and 64-bit platforms like the DEC alpha. It supports asynchronous exceptions, a basic foreign function interface, generational garbage collection, and demand traces. We have only compared the system using simple benchmarks, but it runs an order of magnitude faster than Hugs and about three times as slow as unoptimized GHC code.

3.2 Type inference

The type checker is one of the more interesting parts of the Helium compiler. One of the difficulties of learning a higher-order, functional language such as Haskell is becoming familiar with its underlying type system. As we have seen in the examples, the sophisticated type system of Haskell can easily become a source of frustration as the error messages produced by most modern compilers are often hard to interpret, particularly for beginning students. Frequently, a type error reports an unexpected program location, that might be far from the actual mistake. There are a number of causes for the poor quality of type error messages.

1. Extensions to the type system, and in particular type classes, make the reported error messages harder to interpret. A beginner is immediately confronted with error messages concerning unresolved overloading. Currently, overloading is an integrated part of Haskell, e.g., all numerical literals are automatically overloaded.
2. Most type inferencers have been designed for good performance and suffer from a left-to-right bias, which tends to report type errors towards the end of the program.
3. Type errors concentrate on the two types that are not unifiable. Ideally, the type error message should guide the programmer in removing the type inconsistency, for instance by supplying additional hints.

Constraint-based type inference

Constraint-based type inference is an alternative to algorithms that are based on the bottom-up algorithm $\mathcal{W}$ [3]. Collecting the type constraints (the specification) can be completely separated from solving those constraints (the implementation). Several constraint solvers may be around, each with its own advantages, and, if desired, a set of constraints can be solved in a global fashion. Recent projects that follow the constraint-based approach show that this is a first step to improve the quality of type error messages [1, 6].

Figure 3 shows the process of type inference in Helium. From the abstract syntax tree of a program, a constraint tree is constructed that closely follows the shape of the original tree. By choosing a traversal over this tree, the constraint tree can be flattened to an (ordered) list of constraints. Then, the set of constraints can be solved by the constraint solver one prefers. Currently, there are two constraint solvers to choose from: a greedy constraint solver, and a global constraint solver which uses type graphs.

Greedy constraint solving

The greedy constraint solver handles the type constraints one at a time, and has two advantages: it is implemented in a straightforward way and has a good performance. However, this solver is highly sensitive to the order in which the type constraints are considered, and is biased just like Hugs and GHC. Well-known algorithms, such as $\mathcal{W}$ and the top-down algorithm $\mathcal{M}$, can be simulated by choosing an appropriate traversal over the abstract syntax tree. Similarly, an explicit type declaration can be *pushed down* in the corresponding function definition as the expected type[1] by choosing the moment at which to consider the type constraint generated for the type signature.

Global constraint solving

The global constraint solver considers the entire set of type constraints. Type constraints are incorporated into a type graph, which is an advanced representation of a substitution (see [8] for further details). A type graph also stores the reasons for the type unifications. Because a type graph can be in an inconsistent state, resolving type conflicts can be postponed until the very end. At this point,

[1]The GHC type inferencer is able to push down an explicit type. Providing a type signature for a function definition guides the process of type inferencing.

	exercise 1			exercise 2		exercise 3		
	week 1	week 2	week 3	week 4	week 5	week 6	week 7	total
lex error	3.2%	2.6%	3.4%	4.0%	5.6%	3.9%	5.3%	4.0%
parse error	13.6%	9.8%	7.4%	7.0%	7.9%	11.7%	9.0%	8.6%
static error	6.2%	9.0%	8.8%	9.6%	8.4%	13.5%	10.7%	9.6%
type error	28.1%	34.1%	34.7%	34.8%	25.7%	25.8%	29.6%	31.6%
correct	48.9%	44.5%	45.8%	44.6%	52.4%	45.2%	45.4%	46.3%
N	1109	3823	8230	5481	3871	3256	3661	29431

Figure 4. Compilation results

information about all the contributing sites is available to construct accurate type error messages. Heuristics are employed to determine which program location to report as the most likely source of the type conflict. Although the construction of a type graph requires extra overhead, the system behaves reasonably well for typical laboratory exercises. This constraint solver should undoubtedly be preferred in an educational setting, and is the default for Helium.

To resolve a type inconsistency, we first look at the number of constraints that support one type over another. For example, if there are three indications that variable `x` should be assigned type `Int`, but only one that supports type `Bool`, then `x` will be assigned type `Int`, and the reported error message will focus on the contradicting `Bool`.

In addition to the standard type error message, a number of strategies are applied that can suggest probable fixes to common mistakes. To prevent misleading hints, we only provide one if there is a *unique* correction that *completely* resolves a type inconsistency. Examples of hints include function arguments that are supplied in the wrong order, or whether an argument is missing or superfluous. An important strategy deals with *siblings*, semantically related functions with slightly different types. Examples include `(++)` and `(:)`, `curry` and `uncurry`, but also integer and floating point numbers. If an element of such a pair contributes to a type error, we test whether the type of the sibling would fit in that context. This heuristic has proven to work quite well in practice. Currently, the table of siblings is hard-coded, but we are working on a facility for adding siblings dynamically [9].

The heuristics described above cover a substantial part of the type incorrect programs, but if none succeeds, then soft heuristics decide which location is reported. For instance, a type error concerning an expression is to be preferred over one that refers to a pattern. Another *tie-breaker* is to consider the position of the constraints in the flattened list.

Consider the following program written by one of our students[2].

```
maxLength :: [String] -> Int
maxLength xs = max (map length xs)
```

The effort to write a function that returns the maximum length given a list of strings is almost correct, except that the binary function `max` is used instead of the function `maximum` that works on a list. Hugs reports the following error.

```
ERROR "A.hs":2 - Type error in explicitly typed binding
*** Term           : maxLength
*** Type           : [String] -> [Int] -> [Int]
*** Does not match : [String] -> Int
```

[2]This function was part of the first laboratory exercise. Several individuals made this same mistake.

The definition of `maxLength` is well typed (also because the type of `max`, which is `Ord a => a -> a -> a`, is polymorphic in its first argument), but the inferred type does not match the explicit type signature that was given. Arguably, GHC's error message is more clear.

```
A.hs:2:
    Couldn't match 'Int' against 'a -> a'
        Expected type: Int
        Inferred type: a -> a
    Probable cause: 'max' is applied to too few arguments
        in the call (max (map length xs))
    In the definition of 'maxLength': max (map length xs)
```

Given the type signature of `maxLength`, GHC notices that a second argument should be supplied to `max` in order to return a value of type `Int`, hence the probable cause in the message. However, the message is hard to understand if you are not aware of `max`'s type. Consider Helium's type error message.

```
(2,16): Type error in variable
 expression       : max
   type           : Int   -> Int -> Int
   expected type  : [Int] -> Int
 probable fix     : use maximum instead
```

This message is easier to understand because it mentions the type of `max`, and the type that was expected. In addition, it suggests a probable fix based on a pair of sibling functions.

4 Experience

As a special feature of the compiler, we have set up a logging facility to record compiled programs during an introductory course on functional programming at our institute. The students that took the course were informed about this experiment in advance, and, if desired, they could disable the logging of their programs. To pass the course, three laboratory exercises had to be completed. In a period of seven weeks, thousands of compiled programs produced by participating students were stored on disk. This collection contains both correct and incorrect programs together with the kind of error, and with historical information about a particular user. This anonymous collection of programs reflects the problems that students encounter when learning Haskell, although the choice of the exercises and the lectures of the course may influence the data. The collection is primarily used to further improve the compiler, but it can also give insights in the learning process over time.

Logger results

The collected programs have not yet been analyzed thoroughly, but some overall results can be presented. Figure 4 shows the ratio of programs that were accepted by the compiler, and the phases of the

static error	*N*	(%)
undefined variable	2682	49.9%
undefined constructor	726	13.5%
undefined type constructor	479	8.9%
type signature without definition	409	7.6%
arity mismatch for (type) constructor	361	6.7%
arity mismatch for function definition	228	4.2%
duplicated definition	212	3.9%
filename and module name don't match	87	1.6%
duplicated type signature	49	<1.0%
duplicated variable in pattern	42	..
pattern defines no variables	29	..
undefined type variable	16	..
undefined exported module	13	..
duplicated type constructor	10	..
duplicated constructor	10	..
fixity declaration without definition	9	..
type variable application	8	..
last statement is not an expression	4	..
recursive type synonym	3	..
total	5377	100.0%

Figure 5. Reported static errors

compiler in which an error was detected. Overall, 46% of the compiled programs were accepted by the compiler. We hope that this is an indication that the programs are constructed incrementally by using smaller helper-functions. For more than half of the rejected programs, a type error message is reported. This emphasizes the importance of understandable type error messages once more.

Figure 5 shows the frequencies of the reported static errors. Almost three quarters of the reported static errors are due to an undefined variable, constructor, or type constructor. Of course this includes the misspelling of variable names (a hint is given if there is a resembling variable in scope), but it also suggests that the scoping rules are not well-understood. We intend to analyze the collected data more carefully.

5 Related work

The Helium compiler is certainly not the first to identify the problems that are caused by cryptic error message when learning a language. For the Scheme language, the programming environment DrScheme [5] has been developed, which was initially targeted at students. To gradually become familiar with the language, DrScheme offers a set of language levels, that is, syntactically restricted variants of Scheme. Particularly interesting is the TeachScheme! project [15], which is not only in use at universities, but also at a large number of high schools.

Another interesting project is Pan# [4], a functional graphics language that is in development at Yale University. Pan# combines basic mathematical operations with functional abstraction and a simple vocabulary of images. The system has been used to teach high school students basic algebra operations using images. However, the primary focus is not on teaching a language and not much effort has been put into the quality of error messages. For instance, no type inferencer is present.

During the development of Helium, we paid particular attention to existing example sets, such as the catalogue of Hugs error messages collected by Thompson [16]. A set of type incorrect SML programs by Yang and others [19] was used to highlight the problems of type inference. Another source of inspiration was a recent lively discussion on the Haskell mailing list [7] about teaching Haskell, as well as the feedback we received from our students.

6 Conclusion

Helium is a user-friendly compiler for learning Haskell that produces clear and precise error messages and warnings. The compiler has been used during two introductory courses on functional programming at our institute with great success. Although the compiler is not yet mature, we have received numerous enthusiastic comments, suggestions, and remarks from participating students. In particular, we noticed that the students were no longer distracted by cryptic type error messages caused by overloading. However, at the same time the absence of type classes introduced other problems: dictionaries had to be passed explicitly and advanced textbook examples had to be adapted. Recently, Simon Thompson has released a supplement to his book on functional programming [16], that describes how the textbook can be used with Helium.

In the future, we plan to make Helium more compatible with existing text books, for example, by including a restricted form of overloading. A challenging constraint we impose on ourselves is that the type classes should not have a negative impact on the quality of the (type) error messages.

Further improvements that we would like to make are the integration of documentation into the interpreter, adding warnings about non exhaustive pattern matches, adding a source analyser that recognises common higher-order patterns like `map` and `filter`, and adding a GUI library to support more appealing demos and laboratory exercises. We also investigate the use of external type specifications that guide the type inferencer. These specifications can be used, for example, to specify domain specific type errors [9].

Acknowledgments

Special thanks to Rijk-Jan van Haaften for spending so much time on Helium while still being a student, to Big Brother Jurriaan Hage for building the logging facility, to Arie Middelkoop for saving Helium from the command-line by writing a very pleasant interpreter, and to Martijn Lammerts for designing the logo and website. Also thanks to Arthur Baars, Remco Burema, Atze Dijkstra, Jeroen Fokker, Maarten van Gompel, Andres Löh, Doaitse Swierstra, and to all the students who used Helium and inspired us to introduce new warnings and hints.

7 References

[1] The Chameleon system, `http://www.comp.nus.edu.sg/~sulzmann/chameleon`.

[2] O. Chitil, C. Runciman, and M. Wallace. Freya, Hat and Hood – a comparative evaluation of three systems for tracing and debugging lazy functional programs. In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th international conference on the Implementation of Functional Languages (IFL 2000)*, volume 2011 of *LNCS*, pages 176–193, Aachen, Germany, Sept. 2000. Springer-Verlag.

[3] L. Damas and R. Milner. Principal type schemes for functional programs. In *Principles of Programming Languages (POPL '82)*, pages 207–212, 1982.

[4] C. Elliott, O. de Moore, S. Finne, and J. Peterson. The Pan# functional graphics language. `http://haskell.cs.yale.edu/edsl/pansharp.html`.

[5] R. B. Findler, J. Clements, M. F. Cormac Flanagan, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

[6] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *Proceedings of the 12th European Symposium on Programming*, pages 284–301, April 2003.

[7] The Haskell mailing list, `http://www.haskell.org`.

[8] B. Heeren and J. Hage. Parametric type inferencing for Helium. Technical Report UU-CS-2002-035, Institute of Information and Computing Science, Utrecht University, August 2002.

[9] B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *International Conference on Functional Programming (ICFP'03)*, 2003. To appear.

[10] D. Leijen. The lazy virtual machine. Technical Report UU-CS-2003, Department of Computer Science, Utrecht University, 2003. `http://www.cs.uu.nl/~daan/pubs.html`.

[11] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Department of Computer Science, Utrecht University, 2001. `http://www.cs.uu.nl/~daan/parsec.html`.

[12] B. Pope. Buddha: A declarative debugger for Haskell. Honours thesis, Dept. of computer science, University of Melbourne, Australia, June 1998.

[13] M. Shields and S. Peyton Jones. Object-oriented style overloading for Haskell. In *First Workshop on Multi-language Infrastructure and Interoperability (BABEL'01), Firenze, Italy*, Sept. 2001.

[14] S. D. Swierstra, A. I. Baars, and A. Löh. The UU-AG attribute grammar system. `http://www.cs.uu.nl/groups/ST`.

[15] The TeachScheme! Project, `http://www.teach-scheme.org`.

[16] S. Thompson. *Haskell: The Craft of Functional Programming, Second Edition*. Addison-Wesley Longman, 1999. `http://www.cs.ukc.ac.uk/people/staff/sjt/craft2e`.

[17] A. van IJzendoorn, D. Leijen, and B. Heeren. The Helium compiler. `http://www.cs.uu.nl/helium`.

[18] E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, Sept. 1998.

[19] J. Yang, P. Trinder, J. Wells, and G. Michaelson. Examples to compare the error reportings from the $\mathcal{W}$, $\mathcal{M}$, $\mathcal{U}_{\mathrm{AE}}$, $\mathcal{IEI}$ algorithms. Technical Report RM/00/12, Department of Computing and Electrical Engineering, Heriot-Watt University, October 2000.