Hindley-Milner Type Inference in Domy

Author: Daniel Yuan

Abstract

This report focuses on automatic type inference implemented via the popular Hindley-Milner type inference algorithm. For background, we delve into the theoretical basis of the Hindley-Milner type system. To show a proof-of-concept, Algorithm W was chosen and implemented in JavaScript as part of the type inference system for a basic boolean-centric language, Domy.

Motivation

Strongly-typed systems have always been of particular interest to us, as they not only promote useful programming paradigms such as overloading and inheritance, but also enable developers to have a better experience through tools such as Intelli-sense (automatic documentation and method detection), which only work if variables are properly typed. Additionally, as Programming Languages is a proof-heavy class, we wanted some hands-on experience in interacting with the codebase of a programming language. To that end, we proposed a problem statement in line with our stated goals of producing a product involving strongly typed languages and creating a product beyond proofs.

Problem Statement

Given our class builds directly from the foundational beginnings of programming languages, we were curious how languages are able to implement types automatically from bases such as lambda calculus. To provide a better algorithmic base to build our intuition, the project focuses on the following problem statement:

How do you implement a type inference system?

Background and Literature Review

Hindley-Milner Overview

In order to accomplish the task of automatic type inference, Hindley-Milner (HM) type inference enables programmers to have strongly typed systems without the use of type hints [1]. Furthermore, HM boasts a nearly linear runtime due to the nature of its simple operations running in O(1) [2]. HM is used in a multitude of languages and most notably in the ML family of languages and other languages such as Haskell.

Types in Hindley-Milner

There are two main groups of types in HM: monotypes and polytypes [1].

Monotypes are the backbone of HM and are composed of the typical types that one would expect from programming languages. They include constants, such as `int, bool, string`, and other primitives. Monotypes also include applied types using these constants, such as `List[int]` , a simple data structure, or `a -> string`, a function definition. In the base HM algorithm, there is no support for subclasses, as they complicate type inference [3].

Polytypes are types which are inherently generalized, meaning they can be replaced by many different types. They are categorized by writing a "for all" symbol ($\forall$) in the type in lambda calculus. A simple example of a polytype is in the function `function (x) {return x;}`. As the identity function, we see that `x` can be any type and will return any type; as such, the type for the identity function is the polytype `∀ t. t -> t` as any `t` passed to the identity will return itself.

Hindley-Milner Operations

There are six core operations which form the basis for Hindley-Milner type inference. They are *variable* (variable/function access), *application*, *abstraction*, *let* (variable declaration), *instantiation*, and *generalization* [2, 5]. In each section below, we will briefly explain the typing rule in plain English.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad [\texttt{Var}]$$

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'} \quad [\texttt{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\ x\ .\ e : \tau \to \tau'} \quad [\texttt{Abs}]$$

$$\frac{\Gamma \vdash e_0 : \sigma \qquad \Gamma,\ x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau} \quad [\texttt{Let}]$$

$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma} \quad [\texttt{Inst}]$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall\ \alpha\ .\ \sigma} \quad [\texttt{Gen}]$$

Figure 1. The typing rules for HM type inference [4].

Var

Can be translated to: If x has type σ, then infer that x has type σ. Essentially allows for access of types via variables.

App

Can be translated to: If e_0 has type τ -> τ' and e_1 has type τ, then applying e_0 to e_1 results in type τ'. Allows for the application of functions to arguments, producing the stated type transformation.

Abs

Can be translated to: Assuming x having type τ infers e having type τ', then the abstraction of e with respect to x (λx.e) has type τ -> τ'. This allows for the creation of abstract functions, also known as lambdas.

Let

Can be translated to: If e_0 has type σ and assuming x has type σ infers e_1 has type τ, then substituting let x = e_0 into e_1 results in type τ. The *let* rule allows for the declaration of variables, which we will see extensively in the report's implementation section.

Inst

Can be translated to: If e has type σ' and σ is an instantiation/specialization of σ', then e has type σ. *Inst* allows for the destruction of polytypes into monotypes, where σ' is the polytype and σ is the monotype.

Gen

Can be translated to: If e has type σ and α is not free, then e has type σ independent of α's type. Working in tandem with *Inst*, *Gen* allows for the construction of polytypes, allowing for monotypes not bound in the context to become generalized.

Algorithmic Implementations of Hindley-Milner

There are two main algorithms which implement Hindley-Milner called Algorithm W and Algorithm J.

Algorithm W

Originally presented by Luis Damas and Robin Milner, Algorithm W is the more rigorous and also more commonly implemented version of the two algorithms [6]. Algorithm W applies substitutions liberally in order to unify types, with the *most general unifier* algorithm being the key difference between the two algorithms [2].

Algorithm J

Also presented in the same paper by Damas and Milner, Algorithm J produces far fewer substitutions and "simulates W in a precise sense" [2, 6]. Algorithm J is more concise and more efficient in its substitutions.

Implementation

Given the prevalence of Algorithm W, as well as the multitude of resources and aids in implementation, we chose Algorithm W for our development of HM type inference.

Domy Language

The Domy language is a simple language authored by Steven Yuan in 2019 [7]. In the language, the only base type is the `boolean`, and every variable, function, and operation can be evaluated to a boolean. As such, the interpreter is able to run almost any program regardless of its typing issues.

HM can be applied to Domy to identify the typing errors that a developer may introduce inadvertently, which is particularly useful since the interpreter will not make any indication that such operations are incorrect, as shown by the examples in the Example Code section below.

Methodology

Given that there are multiple examples of Algorithm W, we specifically chose a Python implementation by Robert Smallshire which works on lambda calculus [8]. To our knowledge, there is no readily available implementation of Algorithm W for JavaScript, which is the language that Domy is written in.

As Domy has its own syntax and programming paradigms, to match the format of Algorithm W as presented by Smallshire, we convert the abstract syntax tree (AST) of Domy into the compatible lambda-calculus-esque system of our Algorithm W implementation. Using the converted code, we can perform HM type inference on executed functions. We expected to see the following artifacts:

1. The converted lambda-calculus-esque syntax generated from the Domy AST
2. The output types for executed functions as inferred by HM

What Was Accomplished

Alterations on the Original Code

We found a bug in the original code's parser which prevented variables from being passed in as arguments to functions. Fixing the bug allowed the language to be more expressive and contain more complex code.

Translation of the Python code

We successfully translated the Python code in Smallshire's codebase into JavaScript. We were able to run the examples given in the original Python file and reproduce their outputs in JavaScript. For our understanding, we carefully implemented each function, especially the `unify` function containing the *most generalized unifier* that Algorithm W is known for. To solidify our understanding prior to implementation, we reviewed a blog by Jeremy Mikkola outlining Algorithm W in plain English, which was crucial in debugging the code and also stepping through the outputs [3].

Mapping the Domy AST to Lambda Calculus AST

We made the following mappings of the Domy token types to Lambda Calculus:

Identifier

These were mapped one-to-one from Domy to Lambda Calculus and represented variable and function names.

Variable Declaration and Assignment

As per the HM type rules, variable declarations and re-assignments are accomplished through `let` statements. To that end, a variable declaration would be translated from `my var = true` to `let var = true in ___`, where the blanks are the rest of the program. *Let* was the only typing rule that affected statements beyond its own and resulted in copying ASTs into multiple Lambda Calculus statements as needed; for example, two `print` functions in the same program would need two Lambda Calculus statements as only one type can result from a Lambda Calculus statement.

Function Declaration

The Domy language supports as many function arguments as desired, but the simple Lambda Calculus language only supports one argument per function. As such, we translated Domy argument lists into chained Lambda Calculus statements. For example, the function with the type (bool * bool) -> bool `do(arg1, arg2) { return arg1 & arg2; }` would be translated into the Lambda Calculus statement with the type bool -> bool -> bool `fn arg1 => fn arg 2 => ((and arg1) arg2))`. We can see that (bool * bool) -> bool and bool-> bool -> bool are isomorphic, so this transformation is legal.

Function Invocation

Similarly, we have to recombine the function arguments when calling a function. For example, a simple two argument function such as `and` (type (bool * bool) -> bool) must be translated as two one-argument *App* type rules. We chain *App*s as such for a three argument function:

```
Apply(Apply(Apply(fn, arg1), arg2), arg3)
```

More arguments can be added by wrapping another `Apply` around the above Lambda Calculus statement. We can see that the transformation is once again isomorphic between types (bool * bool) -> bool and bool -> bool -> bool.

Domy-Specific Operations (And, Xor, Or, Test, Print)

To translate these boolean essentials, we declare each function as part of the environment of the HM inference. The Smallshire implementation enables us to predeclare functions and their types; as such, for the functions besides `print`, we declare them with the type bool -> bool -> bool.

Print is a special function since it does not apply a transformation. As such, we treated the typing as the same as the identity function. We declared the polytype `var1` and set the type for `print` as var1 -> var1.

Example Code

1) Simple Inference

We declare a simple program below. Type inference is printed out at every `print` statement. We provide a comment inline to show what the type should be at that line. For the simple inference, we include verbose output for each line of Lambda Calculus to show the execution stack.

Source

```
my a = true;
print(a);           // 1. bool
my fn = do(arg1) {
    return arg1;
}
print(fn);          // 2. a -> a
print(fn(a));       // 3. bool
```

Type Inference Output

```
==Let: a = true in (print a)
==Identifier: true
==Apply: print a
==Identifier: print
==Identifier: a
bool (print(a);)              1.
==Let: a = true in (let fn = (fn arg1 => arg1) in (print
fn))
==Identifier: true
==Let: fn = (fn arg1 => arg1) in (print fn)
==Lambda: arg1 => arg1
==Identifier: arg1
==Apply: print fn
==Identifier: print
==Identifier: fn
```

```
(a -> a) (print(fn);)          2.
==Let: a = true in (let fn = (fn arg1 => arg1) in (print
(fn a)))
==Identifier: true
==Let: fn = (fn arg1 => arg1) in (print (fn a))
==Lambda: arg1 => arg1
==Identifier: arg1
==Apply: print (fn a)
==Identifier: print
==Apply: fn a
==Identifier: fn
==Identifier: a
bool (print(fn(a));)           3.
```

2) Polytypes

In this example, we showcase the identity function as well as another function which takes in two booleans. We also show a typing inference mistake caught by the inference algorithm but executes successfully in Domy.

Source

```
my identity = do(arg1) {
    return arg1;
}
my var = true;
my fn = do(arg1, arg2) {
    return arg1 ^ arg2;
}
print(identity);          // 1. a -> a
print(identity(var));     // 2. bool
print(identity(fn));      // 3. bool -> bool -> bool
print(fn(var, fn));       // 4. Type error (wrong second arg)
```

Type Inference Output

```
(a -> a)                                    1.
bool                                        2.
(bool -> (bool -> bool))                    3.
```

```
Type mismatch: (bool -> (bool -> bool)) != bool   4.
```

Domy Execution Results

```
{
  "type": "function-declaration",
  "args": [
    {
      "text": "arg1",
      "type": "word"
    }
  ],
  "value": {
    "type": "block",
    "value": []
  }
}                                                 1.
true                                              2.
true                                              3.
true                                              4.
```

3) Variable Declaration And Assignment

In this example, we show multiple variables and functions being declared. We also assign a previously boolean variable to become a function instead, showing a type error.

Source

```
my one = true;
print(one);                   // 1. bool
my two = false;
print(two);                   // 2. bool
my dofn = do(arg1, arg2) {
    return arg1 ^ arg2;
}
my fnres = dofn(one, two);
print(fnres);                 // 3. bool
two = true;
```

```
print(dofn(one, two));        // 4. bool
two = do(var) {
    return var;
}
print(dofn(one, two));        // 5. Type error (second arg)
```

Type Inference Output

```
bool                                  1.
bool                                  2.
bool                                  3.
bool                                  4.
Type mismatch: (a -> a) != bool       5.
```

Domy Execution Results

```
true                                  1.
false                                 2.
true                                  3.
false                                 4.
true                                  5.
```

Current Status

The type inference system is fully implemented and supports the basic functions mentioned in What Was Accomplished section. Users can freely create programs and run the `domy` command line and receive the benefits of type inference before execution. All type inference is fully optional, meaning that any type inference errors will not block the execution of the interpreter as evidenced by the last two examples above.

Code blocks, such as bodies of functions, are not fully supported. Only the keyword `return` and no additional state-changing `let` statements are allowed within the code block, although Domy supports `continue` and `break` as part of its grammar. This is due to to the limitations of Lambda Calculus and required copying of the AST to create a new Lambda Calculus statement during every *Let* variable declaration, as each code block creates a new scope, necessitating recursion. The AST recursively branching was almost impossible to do given the current implementation, which we hope to fix with a native Algorithm W model.

The project can be accessed at the GitHub repository linked here: https://github.com/dyuan2001/domy-lang-type-inference/tree/dyuan/type-inference. All type

inference mechanisms are available within the file `bin/inference.js`. All examples are available within the `examples/` folder with the file names below:

1. `simple_inference.do`
2. `polytypes.do`
3. `inference.do`

Future Work

Domy is an expressive language that supports modern looping paradigms as well as recursion. Due to the limitations of the Algorithm W implementation, `while` loops were too difficult to translate to Lambda Calculus. We found a blog post that was able to translate `for` loops to Lambda Calculus; however, since `while` loops are supersets of `for` loops, it would not be a one-to-one translation [9]. We were unable to find any other suitable translation for `while` loops that were available. It would still be a worthwhile endeavor to implement `while` loop functionality in Domy HM type inference.

For a more robust implementation that could more easily support all of Domy's capabilities, Algorithm W should be rewritten for Domy specifically to match Domy's pseudo-type system rather than relying on the translation to Lambda Calculus. Since Domy already supports paradigms that Lambda Calculus does not, we restricted the capabilities of the type inference by translating it into a weaker language. We hope to revisit Domy in order to create a native version of Algorithm W that can encapsulate the entire range of keywords and functionalities.

Related Work

There are many languages which utilize HM type inference as the basis for their automatic type inference. These include ML (Meta Language) and its family of languages, Haskell, and Elm among many others [1].

There are also multiple implementations of Algorithm W. We cite Smallshire's implementation in Python as our primary source [8]. There are also two implementations in Haskell [10, 11] as well as one implementation of Algorithm J in ML [12].

References

[1] P. H, "A reckless introduction to Hindley-Milner Type Inference," *LessWrong*, 05-May-2019. [Online]. Available:

https://www.lesswrong.com/posts/vTS8K4NBSi9iyCrPo/a-reckless-introduction-to-hindley-milner-type-inference. [Accessed: 29-Apr-2024]

[2] "Hindley–Milner Type System," *Wikipedia*, 09-Apr-2024. [Online]. Available: https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system. [Accessed: 29-Apr-2024]

[3] "What part of Hindley-Milner do you not understand?," *Stack Overflow*, 17-Apr-2020. [Online]. Available: https://stackoverflow.com/questions/12532552/what-part-of-hindley-milner-do-you-not-understand. [Accessed: 29-Apr-2024]

[4] A. K. Gupta, "So you still don't understand Hindley-Milner? part 3," *Amit's Blog*. [Online]. Available: https://legacy-blog.akgupta.ca/blog/2013/06/07/so-you-still-dont-understand-hindley-milner-part-3/. [Accessed: 29-Apr-2024]

[5] L. Damas and R. Milner, "Principle type-schemes for functional programs," *Steven Shaw*, 1982. [Online]. Available: https://steshaw.org/hm/hindley-milner.pdf. [Accessed: 30-Apr-2024]

[6] S. Yuan, "Syall/Domy-lang: A hand-rolled interpreted procedural boolean-centric programming language implemented in JavaScript.," *GitHub*, 18-Jul-2020. [Online]. Available: https://github.com/syall/domy-lang. [Accessed: 29-Apr-2024]

[7] R. Smallshire, "Sufficiently Small Dogs and Cat," *A Hindley-Milner type inference implementation in Python*, 11-Apr-2010. [Online]. Available: https://web.archive.org/web/20160324081825/http://smallshire.org.uk/sufficientlysmall/2010/04/11/a-hindley-milner-type-inference-implementation-in-python/. [Accessed: 29-Apr-2024]

[8] J. Mikkola, "Understanding Algorithm W," *Jeremy's Blog*, 25-Mar-2018. [Online]. Available: https://jeremymikkola.com/posts/2018_03_25_understanding_algorithm_w.html. [Accessed: 29-Apr-2024]

[9] "For loops and bounded quantifiers in Lambda Calculus," *Rising Entropy*, 07-Nov-2019. [Online]. Available: https://risingentropy.com/for-loops-and-bounded-quantifiers-in-lambda-calculus/. [Accessed: 29-Apr-2024]

[10] B. Yorgey, "Implementing hindley-milner with the Unification-Fd Library," *blog :: Brent -> [String]*, 08-Sep-2021. [Online]. Available:

https://byorgey.wordpress.com/2021/09/08/implementing-hindley-milner-with-the-unification-fd-library/. [Accessed: 29-Apr-2024]

[11] "Hindley-Milner Inference," *GitHub*, 19-Apr-2017. [Online]. Available: https://github.com/sdiehl/write-you-a-haskell/blob/master/006_hindley_milner.md. [Accessed: 29-Apr-2024]

[12] J. Fecher, "Jfecher/algorithm-J: A minimal implementation of Hindley-Milner's Algorithm J in OCaml," *GitHub*, 20-Jan-2022. [Online]. Available: https://github.com/jfecher/algorithm-j. [Accessed: 29-Apr-2024]