Aspects of efficiency in functional programming languages

by

Samuel Valdemar Grange

supervised by

Prof. Kim Skak Larsen

UNIVERSITY OF SOUTHERN DENMARK
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

Master's thesis in Computer Science

Contents

Part I

Compilers and languages

Chapter 1

Programming languages

Computers are devices which read a well-defined, finite sequence of simple instructions and emit a result. In theoretical analysis of computers, models have been developed to understand and prove properties. A finite sequence of instructions fed to a computer is called an *algorithm*, which is the language of high level computation [3]. In modern encodings of algorithms or programs, "high level" languages are used instead of the computational models. Such languages are then translated into instructions that often are much closer to a computational model. The process of translating programs into computer instructions is called *compiling*, or *transpiling* if the program is first translated into another "high level" language.

For the purpose of this dissertation, a simple programming language has been implemented to illustrate the concepts in detail. The language transpiles to *untyped lambda calculus*. For the remainder, the language will be refered to as L.

1.1 Untyped lambda calculus

The *untyped lambda calculus* is a model of computation, developed by Alanzo Church[1]. The untyped lambda calculus is a simple tangible language of just three terms.

$$x \tag{1.1}$$
$$\lambda x.E \tag{1.2}$$
$$YE \tag{1.3}$$

The first term is that of a *variable* (Equation 1.3). A variable is a reference to another lambda abstraction. Equation 1.2 shows a lambda *abstraction*, which contains a *bound* variable x and another lambda term E. A bound variable is one explicitly parameterized in the abstraction. In traditional programming languages bound variables are parameters to a function, whereas *free* variables are variables that exist outside of the function scope. It is easy to see that variables are either free or bound. Free varables are determined by Equation 1.4, Equation 1.5 and Equation 1.6.

$$Free(x) = \{x\} \tag{1.4}$$
$$Free(\lambda x.E) = Free(E) \backslash \{x\} \tag{1.5}$$
$$Free(YE) = Free(Y) \cup Free(E) \tag{1.6}$$

Finally, in Equation 1.3, *application*. An application of two terms can be interpreted as substituting the variable in the left abstraction Y with the right term E. It is also common to introduce the *let binding* to lambda calculus, which will be done when introducing typing in **??**.

Example 1.1.1. Let Y be $\lambda x.T$ and E be z, then YE is $(\lambda x.T)z$. Furtermore, substituteing x for E, such that Y becomes $T[x := E]$. Since $E = z$, substitute E for z, such that $T[x := z]$, read as "Every instance of x in T, should be substituted by z".

Remark 1.1.1. Substituting lambda terms is a popular method of evaluateing lambda calculus programs. Languages like Miranda, Clean and general purpose evaluation programs like the G-machine **??**, implement *graph rewriting*, which will be introduced in **??**

The untyped lambda calculus is in fact turing complete; any algorithm that can be evaluated by a computer can be encoded in the untyped lambda calculus. The turing completeness of the untyped lambda calculus can be realized by modelling numerics, boolean logic and recursion with the *Y-combitator*. Church encoding, is the encoding of numerics, arithmetic expressions and boolean logic [2]. For the remainder of the dissertation, ordinary arithmetic expressions are written in traditional mathematics. The expressiveness and simplicity of lambda calculus, makes it an exellent language to transpile to, which in fact, is a common technique.

1.2 Translation to lambda calculus

High level languages associated with lambda calculus are often also very close to it. The L language is very close to the untyped lambda calculus. See two equivalent programs, Equation 1.7 and Listing 1.1, that both add an a and a b.

$$(\lambda add.E)(\lambda a.(\lambda b.a + b)) \tag{1.7}$$

Listing 1.1: Add function

```
1 fun add a b = a + b;
```

Notice that in Equation 1.7 the term E is left undefined, E is "the rest of the program in this scope". If the program was to apply 1 and 2 to add, directly below in the high level representation (Listing 1.2) the lambda calculus equivalent would look like Equation 1.8.

$$(\lambda add.add\ 1\ 2)(\lambda a.(\lambda b.a + b)) \tag{1.8}$$

Listing 1.2: Add function applied

```
1 fun add a b = a + b;
2 add 1 2;
```

1.2.1 Scoping

Notice that Equation 1.7, must bind the function name "outside the rest of the program" or more formally in an outer scope. In a traditional program such as Listing 1.3, functions must be explicitly named to translate as in the above example.

Listing 1.3: A traditional program

```
1 fun add a b = a + b;
2 fun sub a b = a - b;
3 sub (add 10 20) 5;
```

Listing 1.4: An order dependant program

```
1 fun sub a b = add a (0 - b);
2 fun add a b = a + b;
3 sub (add 10 20) 5;
```

Notice that there are several problems, such as, the order of which functions are defined may alter whether the program is correct or not. For instance, the program defined in Listing 1.4 would not translate correct, it would translate to Equation 1.9. The definition of *sub*, or rather, the applied lambda abstraction, is missing a reference to the *add* function.

$$(\lambda sub.(\lambda add.(sub\ (add\ 10\ 20)\ 5))\ (\lambda a.(\lambda b.a + b)))\ (\lambda a.(\lambda b.add\ a(0 - b))) \tag{1.9}$$

lambda lifting is a technique where free variables (section 1.1), are explicitly parameterized [8]. This is exactly the problem in Equation 1.9, which has the lambda lifted solution Equation 1.10.

$$(\lambda sub.(\lambda add.(sub\ add\ (add\ 10\ 20)\ 5))\ (\lambda a.(\lambda b.a + b)))\ (\lambda add.(\lambda a.(\lambda b.add\ a(0 - b)))) \tag{1.10}$$

As it will turn out, this will also enables complicated behaviour, such as *mutual recursion*.

Moreover, lambda lifting also conforms to "traditional" scoping rules. *Variable shadowing* occurs when there exists $1 <$ reachable variables of the same name, but the "nearest", in regard to scope distance is chosen. Effectively, other variables than the one chosen, are *shadowed*. Variable shadowing is an implied side-effect of using using lambda calculus. Convince yourself that the function f in Listing 1.5, yields 12.

Listing 1.5: Scoping rules in programming languages

```
1 let x = 22;
2 let a = 10;
3 fun f =
4   let x = 2;
5   a + x;
```

1.2.2 Recursion

Complexity - "The state or quality of being intricate or complicated"

Reductions in mathematics and computer science are one of the princiral methods used developing beautiful equations and algorithms.

Listing 1.6: Infinite program

```
1 fun f n =
2   if (n == 0) n
3   else if (n == 1) n + (n - 1)
4   else if (n == 2) n + ((n - 1) + (n - 2))
5   ...
```

Listing 1.6 defines a function f, that in fact is infinite. Looking at the untyped lambda calculus, there are not any of the three term types that define infinite functions or abstractions, at first glance. Instead of writing an infinite function, the question is rather, how can a reduction be peformed on this function, such that it can evaluate *any* case of n?

Listing 1.7: Recursive program

```
1 fun f n =
2   if (n == 0) n
3   else n + (f (n - 1))
```

Listing 1.7 defines a recursive variant of f, it is a product of the reduction in Equation 1.11.

$$n + (n-1) \cdots + 0 = \sum_{k=0}^{n} k \tag{1.11}$$

But since the untyped lambda calculus is turing complete, or rather, if one were to show it were, it must also realize algorithms that are recursive or include loops, the two of which are equivalent in expressiveness.

$$(\lambda f.E)(\lambda n.\texttt{if}(n == 0)(n)(n + (f(n-1)))) \tag{1.12}$$

The naive implementation of a recursive variant, will yield an unsolvable problem, in fact, an infinite problem. In Equation 1.12, when f is applied recursively, it must be referenced. How can f be referenced, if it is "being constructed"? Substituting f with its implementation in Equation 1.13, will yield the same problem again, but at one level deeper.

$$(\lambda f.E)(\lambda n.\texttt{if}(n == 0)(n)(n + ((\lambda n.\texttt{if}(n == 0)(n)(n + (f(n-1))))(n-1)))) \tag{1.13}$$

One could say, that the problem is now recursive. Recall that lambda lifting (subsection 1.2.1), is the technique of explicitly parameterizing outside references. Convince yourself that f lives in the scope above its own body, such that, when referenceing f from within f, f should be parameterized as in Listing 1.8, translating to Equation 1.14.

Listing 1.8: Explicitly passing recursive function

```
1 fun f f n =
2   if (n == 0) n
3   else n + (f f (n - 1))
```

$$(\lambda f.E)(\lambda f.(\lambda n.\texttt{if}(n == 0)(n)(n + (f\ f\ (n-1))))) \tag{1.14}$$

The initial invocation of f, must involve f, such that it becomes $f\ f\ n$. The *Y-combinator*, an implementation of a fixed-point combinator, in Equation 1.15 is the key to realize that the untyped lambda calculus can implement recursion. Languages with functions and support binding functions to parameters, can implement recursion with the Y-combinator.

$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \tag{1.15}$$

Implementing mutual recursion is an interesting case of lambda lifting and recursion in untyped lambda calculus.

Listing 1.9: Mutual recursion

```
1 fun g = f;
2 fun f = g;
```

Notice in Listing 1.9 that g requires f to be lifted and f requires g to be lifted. If a translation "pessimistically" lifts all definitions from the above scope, then all required references exist in lexical scope.

Languages have diffirent methods of introducing recursion, some of which have very different implications, especially when considering types. For instance, OCaml has the `let rec` binding, to introduce recursive definitions. The `rec` keyword indicates to the compiler that the binding should be able to "see itself" (**??**).

1.3 High level abstractions

The lambda calculus is a powerful language that can express any algorithm. Expressiveness does not necessarily imply ergonomics or elegance, in fact encoding moderately complicated algorithms in lambda calculus becomes quite messy. Many high level techniques exist to model abstractions in tangible concepts.

1.3.1 Algebraic data types

Algebraic data types are essentially a combination of disjoint unions, tuples and records. Algebraic data types are closely related to types thus require some type theory to fully grasp. Types are explored more in depth in **??**.

Listing 1.10: List algebraic data type

```
1 type List a =
2     | Nil
3     | Cons a (List a)
4 ;
```

Listing 1.10 is an implementation of a linked list. The list value can either take the type of `Nil` indicating an empty list, or it can take the type of `Cons` indicating a pair of type a and another list. The list implementation has two type constructors and one type parameter. The type parameter a of the list algebraic data type defines a *polymorphic type*; a can agree on any type, it is universally quantified $\forall a$. `Cons` a `(List` a`)`. The two type constructors `Nil` and `Cons` both create a value of type `List a` once instatiated.

Listing 1.11: List instance and match

```
1 let l = Cons 1 (Cons 2 (Cons 3 Nil));
2
3 match l
4     | Nil -> 0;
5     | Cons n _ -> n;
6 ;
```

Once a value is embedded into an algebraic data type such as a list it must be extractable to be of any use. Values of algebraic data types are extracted and analysed with *pattern matching*. Pattern match comes in may forms, notably it allows one to define a computation based on the type an algebraic data type instance realizes (Listing 1.11).

Scott encoding

Pattern matching strays far from the simple untyped lambda calculus, but can in fact be encoded into it. The *scott encoding* (Equation 1.16) is a technique that describes a general purpose framework to encode algebraic data types into lambda calculus [10]. Considering an algebraic data type instance as a function which accepts a set of "handlers" allows the encoding into lambda calculus. The scott encoding specifies that type constructors should now be functions that are each parameterized by the type constructor parameters $x_1 \dots x_{A_i}$ where A_i is the arity of the type constructor i. Additionally each of the type constructor functions return a n arity function, where n is the cardinality of the set of type constructors. Of the n functions, the type constructor parameters $x_1 \dots x_{A_i}$ are applied to the i'th "handler" c_i. These encoding rules ensure that the "handler" functions are provided uniformly to all instances of the algebraic data type.

$$\lambda x_1 \dots x_{A_i}.\lambda c_1 \dots c_n.c_i x_1 \dots x_{A_i} \tag{1.16}$$

Example 1.3.1. The `List` algebraic data type in Listing 1.10 has two type constructors, `Nil` with the type constructor type Equation 1.17 and `Cons` with the type constructor type Equation 1.18. Equation 1.17 is in fact also the type of `List` once instantiated, effetively treating partially applied

functions as data.

$$b \to (a \to \texttt{List}\ a \to b) \to b \tag{1.17}$$

$$(a \to \texttt{List}\ a \to b) \to b \to (a \to \texttt{List}\ a \to b) \to b \tag{1.18}$$

Listing 1.12: List algebraic data type implementation

```
1 fun cons x xs =
2     fun c _ onCons = onCons x xs;
3     c;
4
5 fun nil =
6     fun c onNil _ = onNil;
7     c;
```

Encoding the constructors in L yields the functions defined in Listing 1.12. Pattern matching is but a matter of applying the appropriate handlers. In Listing 1.13.

Listing 1.13: Example of scott encoded list algebraic data type

```
1 let l = cons 1 (cons 2 (cons 3 nil));
2
3 fun head x _ =
4     x;
5
6 l head 0;
```

Efficiency can be a bit tricky in lambda calculus as it is at the mercy of implementation. A common method of considering efficiency is counting *β-reduction* since they evaluate to function invocations. The β-reduction is a substitution which substitutes an application where the left side is an abstraction in witch the bound variable is substituted with the right side term (Equation 1.19).

$$\beta_{red}((\lambda x.T)E) = T[x := E] \tag{1.19}$$

It should be clear that invoking a n arity function will take n applications. In the case of a scott encoded algebraic data types the largest term in regard to complexity is either the size of the set of "handler" functions or the "handler" function with most parameters. The time to evaluate pattern match is thus $O(\max_i(c_i + A_i))$.

Chapter 2

Typing and validation

Automatic validation is one of many reasons to use computers for solving various tasks including writing new computer programs. Spellchecking is a common and trivial instance of an input validation algorithm.

2.1 Types and validation

The spell checking equivalent for computer programs could be type checking; the subproblem of validating a programmer's intuition of a program's intent. Types also have other properties than simply validating they can in fact be related to theorems to which an implementation is the proof [7].

Listing 2.1: Head implementation

```
1 fun head l:  List a → a =
2     match l
3         | Cons x _  → x;
4         | Nil → ?;
5     ;
```

For instance, consider the implementation of the fuction with type `List a` → `a` in Listing 2.1. A total implementation of the function cannot exist.

The type system for the L language will be the Hindley-Milner type system [6, 9].

2.1.1 The language of types

Before delving into types, the lambda calculus defined in section 1.1 must be augmented with the *let expression* (Equation 2.1).

$$\texttt{let } x = Y \texttt{ in } E \tag{2.1}$$

It should be noted that the let binding can be expressed by abstraction and application (Equation 2.2).

$$(\lambda x.E)(Y) \tag{2.2}$$

The let expression has a nice property that will become apparent later when typing rules are introduced.

Types are an artificial layer atop of a program just as spell checking is an artificial layer atop text. There are two variants of types in the Hindley-Milner type system, the *monotype* and the *polytype*. A monotype is either a type variable, an abstraction of two monotypes or an application of a type constructor(Equation 2.3).

$$mono\ \tau = a \mid \tau \to \tau \mid C\tau_1 \ldots \tau_n \tag{2.3}$$

Atoms are terminal terms in a formula and are expressed either by type variable a or C with no type parameters. The application term of the monotype is dependent on the primitive types of the programming language. The types $\tau_1 \ldots \tau_n$ are monotype parameters required to construct some type C. In L the set of type constructors are $\{\texttt{Int}, \texttt{Bool}\} \cup \texttt{ADT}$. `Int` and `Bool` are type constructors of arity 0 thus only have one instantiation and are atomic. The set of type constructors `ADT` encapsulates the set of program defined algebraic data structures (**??**).

Example 2.1.1. Let $\texttt{ADT} = \{\texttt{List}\}$ where `List` is defined as in Listing 1.10. The type constructor for `List` has the signature `a` $\to$ `List a` stating that if supplied with some type `a` it constructs a type of `List a` (effectively containing the provided type). The type `List` is a type constructor with one type parameter `a`.

$\bot$ denotes falsity, in type systems a value of this type can never exist since that in itself would disprove the program. It is common in programming languages with strong type systems to let thrown exceptions be of type $\bot$ since it adheres to every type and indicates that the program is no longer running, since no instance of $\bot$ can exist. $\top$ denotes truth, in type systems every type is a supertype of $\top$. $\top$ is in practice only used to model side effects, since not all side effects return useful values. In programming languages with side effects $\bot$ and $\top$ are considerably more useful than in pure programming languages.

A polytype is a polymorphic type (Equation 2.4).

$$poly\ \sigma = \tau \mid \forall a.\sigma \tag{2.4}$$

Polymorphic types state that either they take the shape of a type variable or they introduce a type which all types adhere to.

Remark 2.1.1. All types adhere to a polymorphic type but polymorphic types do not adhere to any type other than polymorphic types. The concept of adherence in types is commonly called *subtyping*. Every subtype is a *at least* an implementation of it's supertype. Since this concept can be difficult to grasp from just text, observe Figure 2.1. Note that σ is controversial to

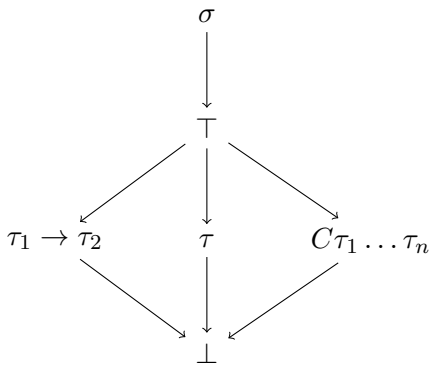

Figure 2.1: The type hierarchy of Hindley-Milner.

introduce to the type hierarchy and has only been so to illustrate the point of subtyping. σ is but a mechanism to prove type systems, σ is never a specific type.

A principal component of typing in Hindley-Milner is the *environment*. The environment Γ is a set of pairs of variable and type (Equation 2.5). $\Gamma \vdash x : \sigma$ implies a *typing judgment*, meaning that given Γ, the variable x can adhere to the type σ.

Remark 2.1.2. Judging a type does not necessarily mean that the judged type is the only type that x may take, it states that it is one *possible* type that x may take. The property of taking multiple possible types is what allows polymorphism. This is made more apparent in 2.1.2 where `id` may take type of $\forall a.a \to a$, `Int` $\to$ `Int` and $\forall a.(a \to a) \to (a \to a)$.

$$\Gamma \ = \epsilon \,|\, \Gamma, x : \sigma \tag{2.5}$$

Like in the untyped lambda calculus, types also have notions of free and bound type variables. Type variables are bound when introduced by quantification. Variables are bound when they have been introduced by a

quantification or exist in the environment.

$$free(a) = \{a\} \tag{2.6}$$

$$free(C\tau_1 \dots \tau_n) = \bigcup_{i=1}^{n} free(\tau_i) \tag{2.7}$$

$$free(\tau_1 \to \tau_2) = free(\tau_1) \cup free(\tau_2) \tag{2.8}$$

$$free(\Gamma) = \bigcup_{x:\sigma \in \Gamma} free(\sigma) \tag{2.9}$$

$$free(\forall a.\sigma) = free(\sigma) - \{a\} \tag{2.10}$$

2.1.2 Hindley-Milner rules

With the now introduced primitives, the Hindley-Milner type system is but a set of inference rules composed by said primitives. There are six rules in

$$\text{Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \text{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

$$\text{Abs}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \text{Let}\ \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2 : \tau}$$

$$\text{Ins}\ \frac{\Gamma \vdash e : \sigma_1 \qquad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash e : \sigma_2} \qquad \text{Gen}\ \frac{\Gamma \vdash e : \sigma \qquad a \notin free(\Gamma)}{\Gamma \vdash e : \forall a.\sigma}$$

Figure 2.2: Hindley-Milner type rules

the Hindley-Milner rules outlined in Figure 3.2.

- **Var** states that if some variable x with type σ exists in the environment, the type can be judged. In practice, when $x : \sigma$ is encountered in the expression tree it is added to the environment.
- **App** decides that if $e_1 : \tau_1 \to \tau_2$ and $e_2 : \tau_1$ has been judged to exist then $e_1 e_2$ implies the removal of τ_1 from $\tau_1 \to \tau_2$ such that $e_1 e_2 : \tau_2$.
- **Abs** is the typing rule of lambda abstractions. If $x : \tau_1$ exists in the environment from some type analysis of e and the abstraction's body e has been judged to be of type τ_2 then the abstraction of x must take the type of x to create the type of the body e.
- **Let** states that if e_1 has been judged to have type σ then the let expression's identifier $x : \sigma$ must exist in the environment when deriving the type of e_2. Note that by 2.1.2 x may be polymorphic in e_2.

The Variable rule states that if some variable x with type σ has been deemed to exist, then they must be in the environment. The Application rule states that if $e_1 e_2$ is of type τ_2 then e_1 must conform to a type that can produce a type τ_2 given a type τ_1 and e_2 must conform to the type of this τ_1. The Instantiate rule specifies a quantified type σ_1 to a specific one σ_2. Generalization lifts a type into a quantified type for all types which are bound. When inferring types in Hindley-Milner it is important to keep track of which variables are bound in a generalization if some instantiation were to happen.

One might ask themselves why the let expression is of any importance since it is expressible by a lambda abstraction. *Let polymorphism* is the property of being able to be polymorphic within different instantiations. When a parameter in a lambda expression is instantiated the first time it cannot adhere to polymorphism anymore, the let expression can, however. Let polymorphism is exemplified in 2.1.2.

Example 2.1.2. The identity function is a common example to illustrate type systems (Listing 2.3).

Listing 2.2: Identity function in L

```
1 fun id x = x;
2 id 4;
```

Listing 2.3: Identity function in lambda calculus with let

```
1 let id = (λx.x) in
2 id 4
```

Stating that id has the type $\forall a.a \to a$ and 4 has the type `Int` is this program correct? By applying the Hindley-Milner rules one can prove or disprove this statement. One pitfall to be aware of while proving types is that of let polymorphism. Figure 2.3 illustrates this point quite well by unifying too early such that `id` is no longer polymorphic. Logically the example also states that the symbol `id` : `Int` $\to$ `Int` exists in the environment in Figure 2.3b which is incorrect.

Listing 2.4: Identity as lambda abstraction

```
1 (λid.id 4)(λx.x)
```

By using a lambda abstraction instead, the parameter type will be instantiated on the first unification (Listing 2.4) thus programs like Listing 2.5 become impossible to prove.

Listing 2.5: Identity with multiple instantiations

```
1 (λid.(id 4, id id))(λx.x)
```

Therefore a correct proof of Listing 2.3 must be Figure 2.4.

$$\text{Gen}\ \dfrac{\text{Abs}\ \dfrac{\text{Var}\ \dfrac{x : a \in \Gamma}{\Gamma \vdash x : a}}{\Gamma \vdash (\lambda x.x) : a \to a} \qquad a \notin \mathit{free}(\Gamma)}{\text{Inst}\ \dfrac{\Gamma \vdash (\lambda x.x) : \forall a.a \to a \qquad \forall a.a \to a \sqsubseteq \texttt{Int} \to \texttt{Int}}{\Gamma \vdash (\lambda x.x) : \texttt{Int} \to \texttt{Int}}}$$

(a)

$$\text{App}\ \dfrac{\text{Var}\ \dfrac{\text{id} : \texttt{Int} \to \texttt{Int} \in \Gamma}{\Gamma \vdash \text{id} : \texttt{Int} \to \texttt{Int}} \qquad \text{Var}\ \dfrac{4 : \texttt{Int} \in \Gamma}{\Gamma \vdash 4 : \texttt{Int}}}{\Gamma,\ \text{id} : \texttt{Int} \to \texttt{Int} \vdash \text{id}\ 4 : \texttt{Int}} \qquad \text{Let}\ \dfrac{\text{2.3a} \qquad \text{2.3b}}{\Gamma \vdash \texttt{let}\ \text{id} = (\lambda x.x)\ \texttt{in}\ \text{id}\ 4 \text{:}\ \texttt{Int}}$$

(b) (c)

Figure 2.3: Incorrect identity function instantiation proof

2.1.3 Damas-Milner Algorithm W

Typing rules are by themselves not that useful since they need all type information declared ahead of checking, inference attempts to guess types such that the rules are satisfied. Type inference is the technique of automatically deriving types, of which there exist many algorithms. One of the most common Hindley-Milner compatible algorithms is the Damas-Milner Algorithm W inference algorithm that infers correct types [4, 5]. The Hindley-Milner rules will accept any rules inferred and accepted by Algorithm W.

The Damas-Milner Algorithm W rules (Figure 3.2) introduce some new concepts such as *fresh variables*, *most general unifier*, and the *substitution set*. Fresh variables are introduced by picking a variable that has not been picked before from the infinite set $\tau_1, \tau_2 \ldots$. Fresh variables are introduced when unknown types are discovered and later unified. The substitution set is a mapping from type variables to types (Equation 2.11).

$$S = \{a_1 \mapsto \tau_1, a_2 \mapsto \tau_2 \ldots, a_n \mapsto \tau_n\} \tag{2.11}$$

A substitution written ST where T is an arbitrary component of Hindley-Milner like an environment in which all type variables are substituted (Figure 2.5). Substitution sets can also be combined $S_1 \cdot S_2$ with well defined-semantics. The combination of substitution sets is a key component for the correctness of the Damas-Milner inference algorithm.

$$S_1 \cdot S_2 = \{(a \mapsto S_1\tau) \mid (a \mapsto \tau) \in (S_2 \cup_. S_1)\} \tag{2.12}$$

The $\cup_.$ operator is distinct in the type variable and is left-biased (Equation 2.13).

$$S_1 \cup_. S_2 = S_1 \cup \{(a \mapsto \tau) \mid (a \mapsto \tau) \in S_2 \mid (a, *) \notin S_1\} \tag{2.13}$$

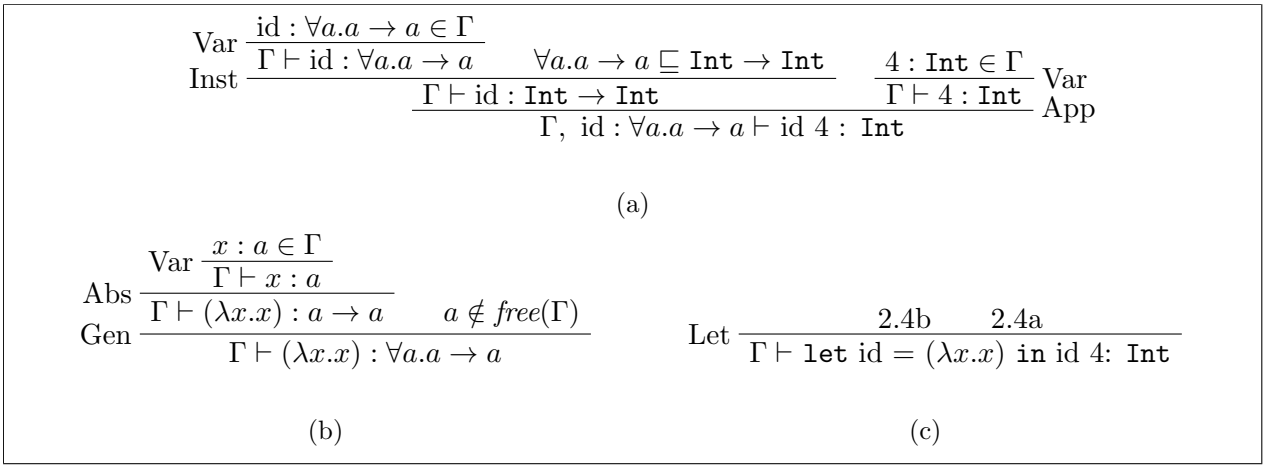

Figure 2.4: Identity function instantiation proof

$$S\Gamma = \{(x, S\sigma) \mid \forall (x, \sigma) \in \Gamma\} \qquad \text{(Environment)}$$

$$S\sigma = \begin{cases} S\tau & \text{if } \sigma \equiv \tau \\ \{a' \mapsto \tau_1 \mid (a', \tau_1) \in S \mid (a, *) \notin S\}\sigma' & \text{if } \sigma \equiv \forall a.\sigma' \end{cases} \qquad \text{(Poly)}$$

$$S(\tau_1 \to \tau_2) = S\tau_1 \to S\tau_2 \qquad \text{(Arrow)}$$

$$Sa = \begin{cases} \tau & \text{if } (a, \tau) \in S \\ a \end{cases} \qquad \text{(Typevariable)}$$

$$SC\tau_1 \ldots \tau_n = CS\tau_1 \ldots S\tau_n \qquad \text{(Type constructor)}$$

Figure 2.5: Substitution semantics

Remark 2.1.3. By the substitution set combination operator transitive and circular substitutions cannot occur since type variables in S_1 will inherit all the mappings from S_2 and substitute ones that exist in both. The properties ensured by the combination semantics also induce the property of idempotence. This property is enforced by the Damas-Milner Algorithm W inference rules.

Unification is performed differently based on the context. Unification is performed on monotypes, each of which can take one of three forms (Equation 2.3). Note that the Var rules for most general unifier outlined in Figure 3.1 are commutative.

Lemma 3. *Var sub and Var empty are commutative.*

Proof. Var empty is trivially true since $\equiv$ is commutative and for any a and

$$\text{Arrow}\ \frac{S, \{(\tau_1 \to \tau_2, \gamma_1 \to \gamma_2)\} \cup T}{S, T \cup \{(\tau_1, \gamma_1), (\tau_2, \gamma_2)\}}$$

$$\text{Intro}\ \frac{\tau_1, \tau_2}{\emptyset, \{(\tau_1, \tau_2)\}} \qquad \text{Var empty}\ \frac{S, \{(a, \tau_1)\} \cup T \qquad a \equiv \tau_1}{S, T}$$

$$\text{Var sub}\ \frac{S, \{(a, \tau_1)\} \cup T \qquad a \notin \mathit{free}(\tau_1)}{S \cup \{a \mapsto S\tau_1\}, \{a \mapsto S\tau_1\} \cup T}$$

$$\text{Atom}\ \frac{S, C_1\tau_1 \ldots \tau_n, C_2\gamma_1 \ldots \gamma_n \cup T \qquad C_1 \equiv C_2}{S, \{(\tau_1, \gamma_1) \ldots, (\tau_n, \gamma_n)\} \cup T}$$

Figure 3.1: Rules for most general unification

τ_1 the rule produces $\emptyset \cup T = T$.

The commutative property of Var sub comes from the realization that $S \cup \{\tau_1 \mapsto S\tau_2\}$ and $S \cup \{\tau_2 \mapsto S\tau_1\}$ lets Algorithm W accept on the same inputs. Furthermor note that Algorithm W substitutes and combines substitution sets at every step of the expression tree such that transitive types never occur because of the combination semantics.

Case 1: The types τ_1 and τ_2 are first introduced and used in unification. Either all future uses of τ_1 will be mapped to τ_2 by the substitution set or all future uses of τ_2 will be mapped to τ_1.

Case 2: τ_1 has been assigned in an earlier unification. If $\tau_2 \mapsto \tau_1$ any existing references to τ_1 need not change since all expressions of type variable τ_2 will be mapped by the rules defined in Figure 3.2. If any future rules need the type variable τ_1 and $\tau_1 \mapsto \tau_2$ is introduced to the substitution set, the substitution rules in Figure 3.2 will substitute the type.

□

$$\text{Var}\ \frac{x : \sigma \in \Gamma \qquad \tau = \mathit{inst}(\sigma)}{\Gamma \vdash x : \tau, \emptyset} \qquad \text{Abs}\ \frac{\tau_1 = \mathit{fresh} \qquad \Gamma, x : \tau_1 \vdash e : \tau_2, S}{\Gamma \vdash \lambda x.e : S\tau_1 \to \tau_2, S}$$

$$\text{App}$$

$$\frac{\Gamma \vdash e_1 : \tau_1, S_1 \quad \tau_3 = \mathit{fresh} \qquad S_1\Gamma \vdash e_2 : \tau_2, S_2 \quad S_2 = \mathit{mgu}(S_2\tau_1, \tau_2 \to \tau_3)}{\Gamma \vdash e_1 e_2 : S_2\tau_3, S_3 \cdot S_2 \cdot S_1}$$

$$\text{Let}\ \frac{\Gamma \vdash e_1 : \tau_1, S_1 \qquad S_1\Gamma, x : S_1\Gamma(\tau_1) \vdash e_2 : \tau_2, S_2}{\Gamma \vdash \texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2 : \tau_2, S_1 \cdot S_2}$$

Figure 3.2: Algorithm W rules

Bibliography

[1] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[2] Alonzo Church. *The calculi of lambda-conversion*. Number 6. Princeton University Press, 1985.

[3] B Jack Copeland. The church-turing thesis. 1997.

[4] Luis Damas. Type assignment in programming languages. 1984.

[5] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, 1982.

[6] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[7] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[8] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Conference on Functional programming languages and computer architecture*, pages 190–203. Springer, 1985.

[9] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[10] Dana Scott. A system of functional abstraction, 1968. lectures delivered at university of california, berkeley. *Cal*, 63:1095, 1962.