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.

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

Equation 1.3 displays a lambda *abstraction* which essentially is a function that states "given some x compute E" where E is another one of the three terms where x may occur. The *variable* (**??**) is a reference to some value introduced by an abstraction. The first term is that of a *variable* (Equation 1.1). 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. In lambda calculus there is also the notion of *context* which simply means where in a lambda expression something is computed. Context is important when discussing *free* and *bound* variables as whether a variable is free or bound is decided by the context. Free varables are determined by Equation 1.4, Equation 1.5 and Equation 1.6.

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

Example 1.1.1.

$$\lambda x.\lambda y.x \tag{1.7}$$

In Equation 1.7 x can appear both free and bound based on the context. If the context is $\lambda y.x$ then x appears free but given the whole expression x appears bound.

In Equation 1.3 the *application* term is displayed. 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 further discussed when introducing typing in **??**.

Example 1.1.2. 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 $E = z$ then 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 *combinator graph rewriting* which is similar and will be introduced in **??**

A remarkable fact about th untyped lambda calculus is that it 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]. Church encoding may prove the power of the untyped lambda calculus but has terrible running time for numerics since to represent some $n \in \mathbb{Z}$ it requires n applications. For the remainder of the dissertation, ordinary arithmetic expressions are written in traditional mathematics. The expressiveness and simplicity of lambda calculus makes it an excellent language to transpile to, which 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 1.8 and Listing 1.1, that both add an a and a b.

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

Listing 1.1: Add function

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

Notice that in Equation 1.8 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 1.9.

$$(\lambda add.add\ 1\ 2)(\lambda a.(\lambda b.a + b)) \tag{1.9}$$

Listing 1.2: Add function applied

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

1.2.1 Scoping

Notice that Equation 1.8, 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 1.10. The definition of *sub*, or rather, the applied lambda abstraction, is missing a reference to the *add* function.

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

lambda lifting is a technique where free variables (section 1.1), are explicitly parameterized [10]. This is exactly the problem in Equation 1.10, which has the lambda lifted solution Equation 1.11.

$$(\lambda sub.(\lambda add.(sub\ add\ (add\ 10\ 20)\ 5))\ (\lambda a.(\lambda b.a + b)))\ (\lambda add.(\lambda a.(\lambda b.add\ a(0 - b)))) \tag{1.11}$$

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

Reductions in mathematics and computer science are one of the principal methods used for 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 1.12.

$$n + (n-1) \cdots + 0 = \sum_{k=0}^{n} k \tag{1.12}$$

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.

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

The naive implementation of a recursive variant, will yield an unsolvable problem, in fact, an infinite problem. In Equation 1.13, 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 1.14, will yield the same problem again, but at one level deeper.

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

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 1.15.

Listing 1.8: Explicitly passing recursive function

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

$$(\lambda f.E)(\lambda f.(\lambda n.\texttt{if}(n == 0)(n)(n + (f\ f\ (n-1))))) \tag{1.15}$$

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 1.16 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.

$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \tag{1.16}$$

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

Listing 1.9: Mutual recursion

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

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

```
type List a =
    | Nil
    | Cons a (List a)
;
```

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 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 constructors `Nil` and `Cons` both create a value of type `List a` once instatiated.

Listing 1.11: List instance and match

```
let l = Cons 1 (Cons 2 (Cons 3 Nil));

match l
    | Nil -> 0;
    | Cons n _ -> n;
;
```

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 1.17) is a technique that describes a general purpose framework to encode algebraic data types into lambda calculus [13]. 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 constructors should now be functions that are each parameterized by the constructor parameters $x_1 \dots x_{A_i}$ where A_i is the arity of the constructor i. Additionally each of the constructor functions return a n arity function, where n is the cardinality

of the set of constructors. Of the n functions, the 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.

$$\lambda x_1 \dots x_{A_i}.\lambda c_1 \dots c_n.c_i x_1 \dots x_{A_i} \tag{1.17}$$

Example 1.3.1. The `List` algebraic data type in Listing 1.10 has two constructors, `Nil` with the constructor type Equation 1.18 and `Cons` with the constructor type Equation 1.19. Equation 1.18 is in fact also the type of `List` once instantiated, effetively treating partially applied functions as data.

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

$$(a \to \texttt{List}\ a \to b) \to b \to (a \to \texttt{List}\ a \to b) \to b \tag{1.19}$$

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 1.20).

$$\beta_{red}((\lambda x.T)E) = T[x := E] \tag{1.20}$$

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; a 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 [9].

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 [8, 12].

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).

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

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 $\tau_1 \dots \tau_n$ 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 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* (not to be confused for constructors like `Cons` or `Nil`) 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 either take the shape of a type variable or introduce a type which all types a adhere to. This does not necessarily include *all* types since the **Gen** rule of Figure 2.2 constrains the domain that a ranges

over to contain only type variables that are not free. Many types may 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 introduce to the type

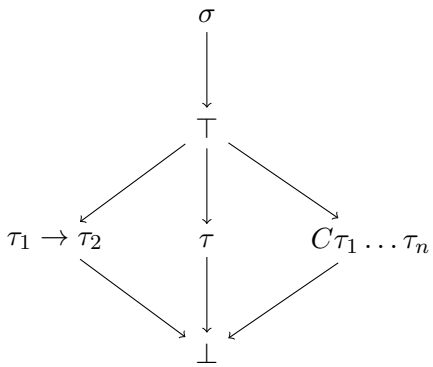


Figure 2.1: The type hierarchy of Hindley-Milner.

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.

Remark 2.1.1. An important implementation detail which should be noted is that of the polymorphic type. Polymorphic types can be regarded as being a pair of bound types and monotype. Instead of keeping track of what types cannot occur, keeping track of the ones than can occur simplifies the implementation. This representation is convenient for the **Gen** rule.

A principal component of typing in Hindley-Milner is the *environment*. The environment Γ is a set of pairs of variable names and polytype (Equation 2.5). $\Gamma \vdash x : \sigma$ implies a *typing judgment*, meaning that given Γ, the variable x can take 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 Example 2.2.2 where `id` may take the type of either $\forall a.a \to a$, `Int` $\to$ `Int` or $\forall a.(a \to a) \to (a \to a)$.

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

Like in the untyped lambda calculus, types also have notions of free and bound type variables. Bound type variables are ones that explicitly

have been introduced to the type system by either let or abstraction in the context of some expression. Type variables are bound when they have been introduced by a quantification or exist in the environment.

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

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

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

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

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

Example 2.1.2. Consider the type for the function `fst` in Listing 2.3.

Listing 2.2: First function

```
1 fun fst a b: ∀A.∀B.A → B → A = a;
```

Listing 2.3: First function in lambda calculus

```
1 let fst = λa.(let f = λb.a in f) in fst
```

The type for `fst` is $\forall A \forall B.A \to B \to A$.

Note that a naive typing could look like $\forall A.A \to (\forall B.B \to A)$ but rank-2 polymorphism is not typable in Hindley-Milner. An important realization is the context from where the type analysis is made. If type analysis is made from within the bounded context of `f` the type of `f` becomes $\forall B.B \to A$ and the type variable A is free.

The variables which may appear in a quantification have an important role in Equation 2.14, since only free variables may be substituted. Free variables are also a core part of generalizing a type for inference algorithms (subsection 2.2.1). When modelling polymorphic types with a technique such as Remark 2.1.1 finding the set of bound variables is trivial.

$$bound(\tau) = free(\tau) - free(\Gamma) \tag{2.11}$$

When generalizing a type τ all types which do not occur in Γ must be quantified.

Example 2.1.3.

$$\Gamma = \{(x, \gamma)\} \tag{2.12}$$

$$\begin{aligned} bound(\tau \to \gamma) &= \{\tau, \gamma\} - free(\Gamma) \\ &= \{\tau, \gamma\} - \{\gamma\} = \{\tau\} \end{aligned} \tag{2.13}$$

Clearly the only bound type variable in the context of $\tau \to \gamma$ is τ such that it may become $\forall\tau.\tau \to \gamma$ in the instance that the type represents a polymorphic let expression. Note that `x:`γ in Equation 2.12 does not contain γ as a quantified type since it has been introduced by an abstraction and **Abs** only introduces monomorphic types (Figure 2.2). An interesting observation is that there can only exist one implementation of the above type system if $\tau \to \gamma$ is to be introduced by a polymorphic let expression which is displayed in Listing 2.4 [14].

Listing 2.4: Implementation of type state

```
1 λx.
2     let z = (λy.x) in
3     ...
```

2.2 Hindley-Milner

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

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

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

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

Figure 2.2: Hindley-Milner type rules

Milner rules outlined in Figure 2.2.

- **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. Observe that **Let** introduces a polymorphic type to the environment while **Abs** introduces a monomorphic one. Note that by Remark 2.1.2 x may be polymorphic in e_2.

- **Inst** specializes some polymorphic type (in regard to the type system implementation) to a more specific polymorphic type. $\sqsubseteq$ is the partial order of types where the binary relation between two types compares the descriptiveness of types.

  Example 2.2.1. In L the smallest element is the top of the type hierarchy (Figure 2.1), the polymorphic type.

- **Gen** generalizes over all bound variables a.

Let polymorphism is exemplified in Example 2.2.2.

Example 2.2.2. Throughout this example (x, z) is the pair of the variables x and z.

The identity function is a common example to illustrate type systems (Listing 2.5).

Listing 2.5: Identity function in L

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

Listing 2.6: 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 Listing 2.6 program correct? By applying the Hindley-Milner rules one can prove or disprove this statement. A correct proof of Listing 2.5 must be Figure 2.3.

Listing 2.7: Identity function in lambda calculus by abstraction

```
1 (λid.id 4)(λx.x)
```

Listing 2.6 and Listing 2.7 are two equivalent programs with slightly different proofs which raises the question of why the let expression is even needed. If Listing 2.6 and Listing 2.7 were to be slightly changed such that two new programs Listing 2.8 and Listing 2.9 were to be proved, Listing 2.9 would not be provable while Listing 2.8 would.

Listing 2.8: Identity function with two applications

```
1 let id = (λx.x) in
2 (id 4, id id)
```

Listing 2.9: Identity function with two applications as abstraction

```
1 (λid.(id 4, id id)(λx.x)
```

In Listing 2.9 `id` cannot adhere to polymorphism by **Abs** in Figure 2.2 whilst **Let** can.

$$\text{Var Inst}\ \frac{\dfrac{\text{id} : \forall a.a \to a \in \Gamma}{\Gamma \vdash \text{id} : \forall a.a \to a} \qquad \forall a.a \to a \sqsubseteq \texttt{Int} \to \texttt{Int}}{\Gamma \vdash \text{id} : \texttt{Int} \to \texttt{Int}} \qquad \frac{4 : \texttt{Int} \in \Gamma}{\Gamma \vdash 4 : \texttt{Int}}\ \text{Var}}{\Gamma,\ \text{id} : \forall a.a \to a \vdash \text{id}\ 4 :\ \texttt{Int}}\ \text{App}$$

(a)

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

(b)

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

(c)

Figure 2.3: Identity function instantiation proof

2.2.1 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 inference algorithms that produce typings which the Hindley-Milner rules accept is the Damas-Milner Algorithm W inference algorithm [4, 5].

The Damas-Milner Algorithm W rules (Figure 2.6) 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 2.14).

$$S = \{a_1 \mapsto \tau_1, a_2 \mapsto \tau_2 \ldots, a_n \mapsto \tau_n\} \tag{2.14}$$

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

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

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

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

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

Figure 2.4: Substitution semantics

A substitution written ST where T is an arbitrary component of Hindley-Milner like an environment in which all type variables are substituted (Figure 2.4). 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.

$$S_1 \cdot S_2 = \{(a \mapsto S_1\tau) \mid (a \mapsto \tau) \in S_2)\} \cup S_1 \qquad (2.15)$$

Remark 2.2.1. 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 by union. Trasitivity is avoided by substituting all instances of type variables values (the mapped to type variables) in S_2 with ones that occur in S_1. 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 2.5 are commutative.

Remark 2.2.2. The Damas-Milner algorithm W is the most popular inference algorithm for Hindley-Milner. Though it remains the most popular, it has some interesting competitors. One of which is that of the constraint solver approach which is also used in OCaml [7]. The constraint solver approach is a two phase type inference algorithm. In the first phase the algorithm inspects the expression tree and generates a set of constraints as it goes. After the set of constraints C has been generated it then traverses the constraints and generates type variable substitutions. It is argued that error reporting is significantly easier in such an approach.

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

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

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

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

Figure 2.5: Rules for most general unification

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

$$\text{App}$$

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

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

Figure 2.6: Algorithm W

Remark 2.2.3. In the language L a function `fun` translates to a let expressions while `let` translates to abstraction and application.

Another interesting addition introduced by algorithm W in Figure 2.6 is *inst*. *inst* naturally follows from the **Inst** rule in Figure 2.2 but has a slightly different behaviour. The *inst* function does not specify types anymore but simply makes unification of polymorphic types possible.

$$inst(\sigma) = \{\, a \mapsto fresh \mid a \notin free(\sigma) \,\}\sigma \tag{2.16}$$

inst (Equation 2.16) maps all bound type variables to fresh type variables in the polytype σ. *inst* is an important component to allow polymorphic types to remain polymorphic since no bound type variables may be substituted.

Example 2.2.3. Performing some type analysis on Listing 2.10 yields a very rich example of why *inst* is necessary.

Listing 2.10: Polymorphic id

```
1 fun id x = x;
2 fun ap x f = f x;
3 fun doubleid x = id (id (x + 1));
```

After inferring `id` and `ap` the environment will contain id $\Gamma = \{(\texttt{id}, \forall a.a \to a), (\texttt{ap}, \forall \gamma, \beta.\gamma \to (\gamma \to \beta) \to \beta)\}$. Typing the function `doubleid` without the use of *inst*, start with the introduced parameter `x` and then the innermost expression `id (x + 1)`.

$$bound(\tau) = free(\tau) - free(\Gamma) = \{\tau\} \qquad (\textbf{Abs} \text{ intro } \texttt{x} : \tau)$$

$$\Gamma = \{(\texttt{id}, \forall a.a \to a), (\texttt{ap}, \forall \gamma, \beta.\gamma \to (\gamma \to \beta) \to \beta), (\texttt{x}, \forall \tau.\tau)\} \tag{2.17}$$

$$unify(\tau, \texttt{Int}) = \{\tau \mapsto \texttt{Int}\} \tag{2.18}$$

$$unify(a \to a, \texttt{Int} \to \mu) = unify(\{a \mapsto \texttt{Int}\}a, \{a \mapsto \texttt{Int}\}\mu) \cdot \{a \mapsto \texttt{Int}\} \tag{2.19}$$

$$= \{\mu \mapsto \texttt{Int}\} \cdot \{a \mapsto \texttt{Int}\}$$

$$= \{\mu \mapsto \texttt{Int}, a \mapsto \texttt{Int}\}$$

This example might not look compromising but a minor change such that the body of `doubleid` becomes `id (ap (id (x + 1)))` yields an interesting problem. In the case of introducing `ap` the two type instances for `id` must be different (`id` must be introduced with different type variables) to retain it's polymorphic properties. The following steps are performed when inferring

this new body.

$$
\begin{aligned}
&\mathit{unify}(\gamma \to (\gamma \to \beta) \to \beta, \texttt{Int} \to \delta) && (\texttt{ap (id (x + 1))})\\
&= \{\gamma \mapsto \texttt{Int}, \delta \mapsto (\texttt{Int} \to \beta) \to \beta\}\\
&\{\gamma \mapsto \texttt{Int}, \delta \mapsto (\texttt{Int} \to \beta) \to \beta\} \cdot \{\mu \mapsto \texttt{Int}, a \mapsto \texttt{Int}\} && (\textbf{App}\ S_3 \cdot S_2)\\
&= \{\gamma \mapsto \texttt{Int}, \delta \mapsto (\texttt{Int} \to \beta) \to \beta, \mu \mapsto \texttt{Int}, a \mapsto \texttt{Int}\}\\
&\mathit{unify}(a \to a, ((\texttt{Int} \to \beta) \to \beta) \to \theta) && (\texttt{id (ap (id (x + 1)))})\\
&= \mathit{unify}(\{a \mapsto (\texttt{Int} \to \beta) \to \beta\}a, \{a \mapsto (\texttt{Int} \to \beta) \to \beta\}\theta) \cdot \{a \mapsto (\texttt{Int} \to \beta) \to \beta\}\\
&= \{a \mapsto (\texttt{Int} \to \beta) \to \beta, \theta \mapsto (\texttt{Int} \to \beta) \to \beta\} \cdot \{a \mapsto (\texttt{Int} \to \beta) \to \beta\}\\
&= \{a \mapsto (\texttt{Int} \to \beta) \to \beta, \theta \mapsto (\texttt{Int} \to \beta) \to \beta\}\\
&\{a \mapsto (\texttt{Int} \to \beta) \to \beta, \theta \mapsto (\texttt{Int} \to \beta) \to \beta\} \cdot && (\textbf{App}\ S_3 \cdot S_2)\\
&\{\gamma \mapsto \texttt{Int}, \delta \mapsto (\texttt{Int} \to \beta) \to \beta, \mu \mapsto \texttt{Int}, a \mapsto \texttt{Int}\}
\end{aligned}
$$

Clearly a cannot map to two types which cannot be unified which is a violation of the type system. The apparent problem is that `id` is specialized within the whole of `doubleid`. By instantiating quantified types when they are needed cases such as this can be avoided (it also makes the algorithm correct).

$$
\begin{aligned}
&\mathit{unify}(\mathit{inst}(\forall a.a \to a), \mathit{inst}(\forall \tau.\tau \to \mu)) && (2.20)\\
&= \mathit{unify}(\gamma \to \gamma, \varphi \to \mu)\\
&= \{\varphi \mapsto \mu, \gamma \mapsto \mu\}
\end{aligned}
$$

2.2.2 Recursion

Recursion is a trivial matter once the primitives of the Hindley-Milner type system have been introduced. Recall that in subsection 1.2.2 recursion (along with mutual recursion) was shown to be implementable by introducing functions to their own scope, the same is true for types. Allowing recursive functions in Hindley-Milner type inference systems is a matter of letting the function be present in the environment when inferring the function's own body.

Example 2.2.4. If the function `f` defined in Listing 2.11 were to be typed it would need to be introduced as an unknown type to the environment before typing the body of `f`.

Listing 2.11: Recursive function

```
1 fun f x = (f x) + 1;
```

Let $\Gamma = \{\texttt{f} : \forall\tau.\tau, \texttt{x} : \forall\mu.\mu\}$. From the application `f x` the unification $\texttt{unify}(\tau, \gamma \to \mu) = \{\tau \mapsto \mu \to \gamma\}$ must be performed, and the resulting type for the expression is γ. The addition operation forces $\texttt{unify}(\texttt{Int}, \gamma) = \{\gamma \mapsto$

`Int`}. Finally the application of the addition function $+ : \texttt{Int} \to \texttt{Int} \to \texttt{Int}$ and the two expressions `f x` and 1 such that the resulting expression type is `Int`.

2.2.3 Additional language features

In addition to the rules in Figure 2.6 many other ergonomic features can easily be modelled once the framework has been understood. One of the most crucial features of languages are that of decision.

Listing 2.12: ADT implementation of decision

```
1 type Boolean =
2     | BFalse
3     | BTrue
4 ;
5 let b = BFalse;
6 fun double x = x + x;
7 b (0) (double 10);
```

Decision can be implemented in a variety of ways such as in Listing 2.12 by the use of algebraic data structures aligning very much with Church Booleans [2]. Rather decision can be implemented by more conventional

$$\Gamma \vdash e_2 : \tau_2, S_2 \;\; \tau_4 = \mathit{fresh} \;\; \Gamma, S_4 = \mathit{mgu}(S_1' \cdot S_2 \cdot S_3\tau_2, \tau_4)$$

(a)

$$\Gamma \vdash e_1 : \tau_1, S_1 \;\; S_1' = \mathit{mgu}(\texttt{Bool}, \tau_1) \qquad \Gamma \vdash e_3 : \tau_3, S_3 \;\; \Gamma, S_5 = \mathit{mgu}(S_4 \cdot S_1' \cdot S_2 \cdot S_3\tau_3, S_4\tau_4)$$

(b) (c)

$$\dfrac{\text{2.7b} \qquad \dfrac{\text{2.7a} \qquad \text{2.7c}}{\Gamma, S_6 \;\; S_6 = S_5 \cdot S_4 \cdot S_1' \cdot S_2 \cdot S_3}}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : S_6\tau_4, S_6}$$

Figure 2.7: Decision

methods than combinator logic by introducing more inference rules as in Figure 2.7. Additional language syntax features can in most cases be implemented as decision can.

2.3 The cost of expressiveness

Modern languages with strong type systems tend to be notoriously slow to type on pathological inputs. In fact many languages with strong type systems provide type systems expressive enough to be turing complete.

In the construction of the compiler for L one target was the C++ language. An instance of a pathological input for the C++ type checker is most definitely the untyped lambda calculus. The lambda terms in C++ must adhere to polymorphism in many cases which has some unknown but large blowup in compilation time. In fact type polymorphism is commonly the root of blowup in typing.

ML which implemented Hindley-Milner was believed to have linear complexity before shown to be exponential along with other problematic complexity findings [11]. As it will turn out, Hindley-Milner also suffers an explosive worst case induced by a pathological input fueled by polymorphism.

Lemma 3. *Hindley-Milner typing is worst case* $O(c^n)$.

Proof. The basis of the blowup stems from the introduced fresh type variables in the polymorphic let inference rule. If the amount of type variables can be shown to be exponential the running time must be at least the same by operations such as subsection set combination and unification.

Listing 3.1: Nested pair

```
1 fun dup a f = f a a;
2 fun deep x = dup (dup (dup (... x)));
```

Listing 3.1 builds a large function signature for `deep`. The innermost `pair` invocation will have it's signature unified to $x \to (x \to x \to \tau) \to \tau$. The second innermost `pair` invocation has the signature $((x \to x \to \tau) \to \tau) \to (((x \to x \to \tau) \to \tau) \to \gamma) \to \gamma$. Naively one might judge Listing 3.1 to run in $O(2^n)$ but an important observation for why Listing 3.1 does not induce exponential blowup is the uniqueness of the type variables. If an efficient representation of `dup` was implemented such that the left and right side were shared such that $\mu \mapsto ((x \to x \to \tau) \to \tau)$, the amount introduced type variables would be $O(n)$.

Listing 3.2: Nested pairs with different type variables

```
1 fun tuple a b f = f a b;
2 fun one = tuple tuple tuple;
3 fun two = tuple one one;
4 fun three = tuple two two;
5 ...
```

The trick to induce an exponential running time is with the pathological program in Listing 3.2. By allowing `tuple` to be polymorphic and have having $c > 1$ polymorphic parameters, every time `tuple` is instantiated it will contain only fresh variables. The type of `tuple` is $a \to b \to (a \to b \to c) \to c$. Clearly this looks very much like Listing 3.1 but has the subtle difference of letting the parameters `a` and `b` be polymorphic and introducing every "step" as a polymorphic let expression. The return type of `one` (the type of `f`) is displayed in Equation 3.1.

$$inst(a \to b \to (a \to b \to c) \to c) \to inst(a \to b \to (a \to b \to c) \to c) \to \gamma \to \gamma. \tag{3.1}$$

The first and second instantiations will contain different type variable such that they are not structurally equivalent Equation 3.2.

$$(\tau \to \mu \to (\tau \to \mu \to \phi) \to \phi) \to (\varphi \to \zeta \to (\varphi \to \zeta \to \delta) \to \delta) \to \gamma \to \gamma. \tag{3.2}$$

□

Running the program Listing 3.2 in L yields a blowup of 2^n (Listing 5.1). Figure 3.1 shows the relationship between the program in L, the theoretical time and an exp fit of data from L.

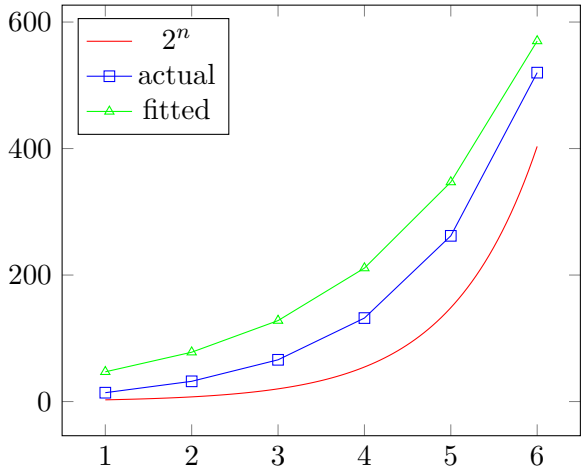


Figure 3.1: Plot of type variables in Hindley-Milner type systems

Chapter 4

Program evaluation

The untyped lambda calculus may provide a simple interface for programming but does not pair very well with the modern computer. *Interpreting* is a common technique for evaluating the untyped lambda calculus. An interpreter is an execution engine usually implemented in a more low-level language.

4.1 Evaluation strategies

When evaluating the untyped lambda calculus one has to choose an evaluation strategy. The choice of evaluation strategy has a large impact on aspects such as complexity guarantees. The names of such strategies are *call by value*, *call by name* and *call by need*. The call by value is most often the simplest and most natural way of assuming program execution.

Listing 4.1: Program that doubles values

```
1 fun double x = x + x;
2 let a = double 10;
3 double (double 10);
```

By the call by value semantics, Listing 4.1 eagerly evaluates every expression. Clearly the variable `a` is never used but under the call by value semantics everything is eagerly evaluated. Every expression is evaluated in logical order in the call by value evaluation strategy. The call by name semantics however does only evaluate expressions once they are needed (also commonly called *lazy evaluation*). By the call by name semantics `a` is never evaluated since it is never used. In Listing 4.2 call by name has been implemented by the use of various functions such as the two constant functions `suspend` and `force`. `susExpensiveOp` ensures that the forcing (evaluation) of `x` never occurs until the caller of `double` forces the result. By the aforementioned semantics of call by name in the context of the program in Listing 4.2 `a` is never forced thus the computation is never performed. The implementation

Listing 4.2: Implementation of call by name

```
1 fun suspend x unit = x;
2 fun force x = x 0;
3 let value = suspend 10;
4 fun double x =
5     fun susExpensiveOp unit =
6         (force x) + (force x);
7     susExpensiveOp;
8 let a = double value;
9 force (double value);
```

of call by name can become quite troublesome and therefore in most cases is a part of the native execution environment which will be discussed in **??**.

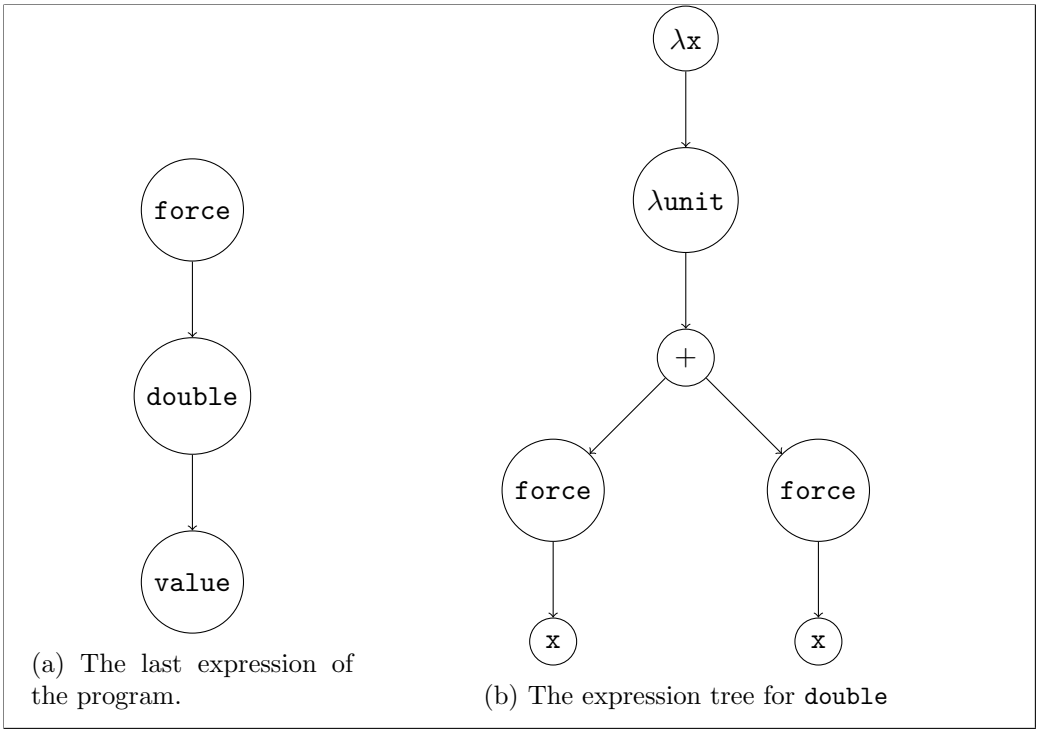
The call by need semantics introduces the same lazy evaluation semantics as the call by name strategy but with one extra detail named *sharing*. In Listing 4.2 `force x` is performed twice which may be an expensive operation. Under call by need all non side-effectful operations' results are saved for later use similar to techniques such as dynamic programming. To understand this better observe the expression tree for Listing 4.2 in Figure 4.1. Clearly the right and left branches in Figure 4.1b are identical thus they may be memoized such that the forcing of `x` only occurs once. More generally if the execution environment supports lazy evaluation, once an expression has been forced it is remembered if that branch is executed again.

4.2 Runtime environments

Now that the untyped lambda calculus has been introduced, implemented and validated efficiently the question of execution naturally follows. There exists many different well understood strategies to implement an execution environment for the untyped lambda calculus. Naively it may seem straightforward to evaluate the untyped lambda calculus mechanically by β-reductions but a problem arises when implementing an interpreter for application. When applying the term fx in $\lambda f.\lambda x.fx$ such that $f[? := x]$ where ? is the parameter name of f it should become clear why a naive strategy is not enough since the parameters of f are anonymous.

4.2.1 Combinator reducers

One of the most prominent techniques for evaluating functional programs is that of *combinator graphs reductions*. Formally a combinator is a function that has no free variables which is convenient since the problem of figuring



(a) The last expression of the program.

(b) The expression tree for `double`

Figure 4.1

out what a parameter name is never arises.

$$x \tag{4.1}$$

$$F \tag{4.2}$$

$$YE \tag{4.3}$$

There are three types of terms in combinator logic; the variable much like the lambda calculus (Equation 4.1), application (Equation 4.3) and the combinator (Equation 4.2). The SKI calculus is a very simple set of combinators which are powerful enough to be turing complete and translate to and from the lambda calculus. In SKI $F ::= S \mid K \mid I$ where the equivalent lambda calculus combinators for $S = \lambda x.\lambda y.\lambda z.xz(yz)$, $K = \lambda x.\lambda y.x$ and $I = \lambda x.x$. Evaluating an SKI program is a straightforward reduction where F'_F denotes combinator F' has been partially applied with combinator F.

Example 4.2.1.

$$\begin{aligned} & SKSI \\ =& KI(SI) \\ =& K_I(SI) \\ =& I \end{aligned} \tag{4.4}$$

The algorithm for converting a lambda calculus program into a SKI combinator program is a straightforward mechanical one. The evaluation context is always an abstraction $\lambda x.E$ thus translation of E will also always be an abstraction.

Case 1: $E = x$ then rewrite $\lambda x.E$ to I.

Case 2: $E = y$ where y is a variable then rewrite $\lambda x.y$ to Ky.

Case 3: $E = YE$ then rewrite $\lambda x.YE$ to $S(\lambda x.Y)(\lambda x.E)$ since applying some y must lambda lift y as a parameter named x to both Y and E such that the lifted expression becomes $((\lambda x.Y)y)((\lambda x.E)y) = S(\lambda x.Y)(\lambda x.E)y$. Then recurse in both branches.

Case 4: $E = \lambda y.E'$ rewrite E' and then rewrite the expression.

When translating the untyped lambda calculus to SKI the "magic" variable names ζ, κ and ι are used as placeholder functions for the SKI combinators since the translation requires a lambda calculus form. When the translation has been completed then replace $\zeta \mapsto S, \kappa \mapsto K, \iota \mapsto I$.

4.2.2 Combinator translation growth

$$
\begin{aligned}
&\lambda f_1.(\lambda f_2.f_1 f_2)(\lambda f_2.f_1 f_2) && (4.5)\\
=&\zeta(\lambda f_1.(\lambda f_2.f_1 f_2))(\lambda f_1.(\lambda f_2.f_1 f_2))\\
=&\zeta(\lambda f_1.\zeta(\lambda f_2.f_1)(\lambda f_2.f_2))(\lambda f_1.\zeta(\lambda f_2.f_1)(\lambda f_2.f_2))\\
=&\zeta(\lambda f_1.\zeta(\kappa f_1)(\iota))(\lambda f_1.\zeta(\kappa f_1)(\iota))\\
=&\zeta(\zeta(\lambda f_1.\zeta(\kappa f_1))(\lambda f_1.\iota))(\zeta(\lambda f_1.\zeta(\kappa f_1))(\lambda f_1.\iota))\\
=&\zeta(\zeta(\zeta(\lambda f_1.\zeta)(\lambda f_1.\kappa f_1))(\kappa\iota))(\zeta(\zeta(\lambda f_1.\zeta)(\lambda f_1.\kappa f_1))(\kappa\iota))\\
=&\zeta(\zeta(\zeta(\kappa\zeta)(\zeta(\lambda f_1.\kappa)(\lambda f_1.f_1)))(\kappa\iota))(\zeta(\zeta(\kappa\zeta)(\zeta(\lambda f_1.\kappa)(\lambda f_1.f_1)))(\kappa\iota))\\
=&\zeta(\zeta(\zeta(\kappa\zeta)(\zeta(\kappa\kappa)(\iota)))(\kappa\iota))(\zeta(\zeta(\kappa\zeta)(\zeta(\kappa\kappa)(\iota)))(\kappa\iota))\\
=&S(S(S(KS)(S(KK)(I)))(KI))(S(S(KS)(S(KK)(I)))(KI))
\end{aligned}
$$

4.2.3 Beta reductions with De Bruijn indices

Beta reductions with De Bruijn indices [6] is another more straightforward method of evaluating the untyped lambda calculus. De Bruijn indices is a representation of lambda calculus which deals with variables based on the scope "distance" instead of variable names.

Listing 4.3: Add as De Bruijn

```
1 let add = (λ(λ 2 1))
```

Consider the De Bruijn form of Listing 1.1 where `2` is the index of `a` and `1` is the index of `b`. More generally all variable occurrences are replaced with the distance to the abstraction which introduced them. The use of De Bruijn indices allow anonymous naming of parameters thus imply a method of solving the application problem when directly interpreting the untyped lambda calculus. A minor but important detail when

$$
\begin{aligned}
&(\lambda x.((\lambda f.\lambda x.fx)(\lambda z.z))x)''0" \rightarrow && (4.6)\\
& && (4.7)
\end{aligned}
$$

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] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. North-Holland, 1972.

[7] BJ Heeren, Jurriaan Hage, and S Doaitse Swierstra. Generalizing hindley-milner type inference algorithms, 2002.

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

[9] 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.

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

[11] Harry G Mairson. Deciding ml typability is complete for deterministic exponential time. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 382–401, 1989.

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

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

[14] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, 1989.

Chapter 5

Appendix

Listing 5.1: The output of an exponential type

```
1  ########## tuple ##########
2  substitution set Map(c0 -> (a0 -> (b0 -> d0)))
3  type (a0 -> (b0 -> ((a0 -> (b0 -> d0)) -> d0)))
4  current env is Map(tuple -> Scheme(Set(a0, b0, d0),(
       a0 -> (b0 -> ((a0 -> (b0 -> d0)) -> d0)))))
5  Type vars = 4
6  Type vars in sub = 4
7  ########## tuple ##########
8  ########## one ##########
9  substitution set Map(c0 -> (a0 -> (b0 -> d0)), e0 ->
       (h0 -> (i0 -> ((h0 -> (i0 -> j0)) -> j0))), f0
       -> (k0 -> (l0 -> ((k0 -> (l0 -> m0)) -> m0))), n0
       -> (((h0 -> (i0 -> ((h0 -> (i0 -> j0)) -> j0)))
       -> ((k0 -> (l0 -> ((k0 -> (l0 -> m0)) -> m0))) ->
       g0)) ->
10 g0))
11 type (((h0 -> (i0 -> ((h0 -> (i0 -> j0)) -> j0))) ->
       ((k0 -> (l0 -> ((k0 -> (l0 -> m0)) -> m0))) ->
       g0)) -> g0)
12 current env is Map(tuple -> Scheme(Set(a0, b0, d0),(
       a0 -> (b0 -> ((a0 -> (b0 -> d0)) -> d0)))), one
       -> Scheme(Set(k0, g0, h0, i0, l0, m0, j0),(((h0
       -> (i0 -> ((h0 -> (i0 -> j0)) -> j0))) -> ((k0 ->
       (l0 -> ((k0 -> (l0 -> m0)) -> m0))) -> g0)) ->
       g0)))
13 Type vars = 14
14 Type vars in sub = 33
15 ########## one ##########
16 ########## two ##########
```

```
substitution set Map(o0 -> (((t0 -> (u0 -> ((t0 -> (u0 -> x0)) -> x0))) -> ((r0 -> (v0 -> ((r0 -> (v0 -> w0)) -> w0))) -> s0)) -> s0), e0 -> (h0 -> (i0 -> ((h0 -> (i0 -> j0)) -> j0))), f0 -> (k0 -> (l0 -> ((k0 -> (l0 -> m0)) -> m0))), p0 -> (((a1 -> (b1 ->
 ((a1 -> (b1 -> e1)) -> e1))) -> ((y0 -> (c1 -> ((y0 -> (c1 -> d1)) -> d1))) -> z0)) -> z0), n0 -> (((h0 -> (i0 -> ((h0 -> (i0 -> j0)) -> j0))) -> ((k0 -> (l0 -> ((k0 -> (l0 -> m0)) -> m0))) -> g0)) -> g0), c0 -> (a0 -> (b0 -> d0)), f1 -> (((((t0 -> (u0 ->
 ((t0 -> (u0 -> x0)) -> x0))) -> ((r0 -> (v0 -> ((r0 -> (v0 -> w0)) -> w0))) -> s0)) -> s0) -> ((((a1 -> (b1 -> ((a1 -> (b1 -> e1)) -> e1))) -> ((y0 -> (c1 -> ((y0 -> (c1 -> d1)) -> d1))) -> z0)) -> z0) -> q0)) -> q0))
type (((((t0 -> (u0 -> ((t0 -> (u0 -> x0)) -> x0))) -> ((r0 -> (v0 -> ((r0 -> (v0 -> w0)) -> w0))) -> s0)) -> s0) -> ((((a1 -> (b1 -> ((a1 -> (b1 -> e1)) -> e1))) -> ((y0 -> (c1 -> ((y0 -> (c1 -> d1)) -> d1))) -> z0)) -> z0) -> q0)) -> q0)
current env is Map(tuple -> Scheme(Set(a0, b0, d0),(a0 -> (b0 -> ((a0 -> (b0 -> d0)) -> d0)))), one -> Scheme(Set(k0, g0, h0, i0, l0, m0, j0),(((h0 -> (i0 -> ((h0 -> (i0 -> j0)) -> j0))) -> ((k0 -> (l0 -> ((k0 -> (l0 -> m0)) -> m0))) -> g0)) -> g0)), two -
> Scheme(Set(u0, x0, q0, a1, b1, s0, e1, d1, z0, w0, y0, v0, t0, c1, r0),(((((t0 -> (u0 -> ((t0 -> (u0 -> x0)) -> x0))) -> ((r0 -> (v0 -> ((r0 -> (v0 -> w0)) -> w0))) -> s0)) -> s0) -> ((((a1 -> (b1 -> ((a1 -> (b1 -> e1)) -> e1))) -> ((y0 -> (c1 -> ((y0 ->
 (c1 -> d1)) -> d1))) -> z0)) -> z0) -> q0)) -> q0))
    )
Type vars = 32
Type vars in sub = 94
########## two ##########
########## three ##########
substitution set Map(o0 -> (((t0 -> (u0 -> ((t0 -> (u0 -> x0)) -> x0))) -> ((r0 -> (v0 -> ((r0 -> (v0 -> w0)) -> w0))) -> s0)) -> s0), e0 -> (h0 -> (i0 -> ((h0 -> (i0 -> j0)) -> j0))), n2 -> (((((((
```

```
   v1 -> (j1 -> ((v1 -> (j1 -> k1)) -> k1))) -> ((x1
    -> (u1 -
> ((x1 -> (u1 -> s1)) -> s1))) -> o1)) -> o1) ->
   ((((m1 -> (n1 -> ((m1 -> (n1 -> p1)) -> p1))) ->
   ((t1 -> (w1 -> ((t1 -> (w1 -> q1)) -> q1))) -> r1
   )) -> r1) -> l1)) -> l1) -> ((((((k2 -> (y1 -> ((
   k2 -> (y1 -> z1)) -> z1))) -> ((m2 -> (j2 -> ((m2
    -> (j2 ->
h2)) -> h2))) -> d2)) -> d2) -> ((((b2 -> (c2 -> ((
   b2 -> (c2 -> e2)) -> e2))) -> ((i2 -> (l2 -> ((i2
    -> (l2 -> f2)) -> f2))) -> g2)) -> g2) -> a2))
   -> a2) -> i1)) -> i1), f0 -> (k0 -> (l0 -> ((k0
   -> (l0 -> m0)) -> m0))), p0 -> (((a1 -> (b1 -> ((
   a1 -> (b1 -
> e1)) -> e1))) -> ((y0 -> (c1 -> ((y0 -> (c1 -> d1)
   ) -> d1))) -> z0)) -> z0), n0 -> (((h0 -> (i0 ->
   ((h0 -> (i0 -> j0)) -> j0))) -> ((k0 -> (l0 -> ((
   k0 -> (l0 -> m0)) -> m0))) -> g0)) -> g0), c0 ->
   (a0 -> (b0 -> d0)), g1 -> (((((v1 -> (j1 -> ((v1
   -> (j1 -
> k1)) -> k1))) -> ((x1 -> (u1 -> ((x1 -> (u1 -> s1)
   ) -> s1))) -> o1)) -> o1) -> ((((m1 -> (n1 -> ((
   m1 -> (n1 -> p1)) -> p1))) -> ((t1 -> (w1 -> ((t1
    -> (w1 -> q1)) -> q1))) -> r1)) -> r1) -> l1))
   -> l1), h1 -> (((((k2 -> (y1 -> ((k2 -> (y1 -> z1
   )) -> z1))
) -> ((m2 -> (j2 -> ((m2 -> (j2 -> h2)) -> h2))) ->
   d2)) -> d2) -> ((((b2 -> (c2 -> ((b2 -> (c2 -> e2
   )) -> e2))) -> ((i2 -> (l2 -> ((i2 -> (l2 -> f2))
    -> f2))) -> g2)) -> g2) -> a2)) -> a2), f1 ->
   (((((t0 -> (u0 -> ((t0 -> (u0 -> x0)) -> x0))) ->
    ((r0 -> (
v0 -> ((r0 -> (v0 -> w0)) -> w0))) -> s0)) -> s0) ->
    ((((a1 -> (b1 -> ((a1 -> (b1 -> e1)) -> e1))) ->
    ((y0 -> (c1 -> ((y0 -> (c1 -> d1)) -> d1))) ->
   z0)) -> z0) -> q0)) -> q0))
type (((((((v1 -> (j1 -> ((v1 -> (j1 -> k1)) -> k1))
   ) -> ((x1 -> (u1 -> ((x1 -> (u1 -> s1)) -> s1)))
   -> o1)) -> o1) -> ((((m1 -> (n1 -> ((m1 -> (n1 ->
    p1)) -> p1))) -> ((t1 -> (w1 -> ((t1 -> (w1 ->
   q1)) -> q1))) -> r1)) -> r1) -> l1)) -> l1) ->
   ((((((k2 ->
 (y1 -> ((k2 -> (y1 -> z1)) -> z1))) -> ((m2 -> (j2
    -> ((m2 -> (j2 -> h2)) -> h2))) -> d2)) -> d2)
```

```
    -> ((((b2 -> (c2 -> ((b2 -> (c2 -> e2)) -> e2)))
     -> ((i2 -> (l2 -> ((i2 -> (l2 -> f2)) -> f2)))
    -> g2)) -> g2) -> a2)) -> a2) -> i1)) -> i1)
current env is Map(tuple -> Scheme(Set(a0, b0, d0),(
   a0 -> (b0 -> ((a0 -> (b0 -> d0)) -> d0)))), one
   -> Scheme(Set(k0, g0, h0, i0, l0, m0, j0),(((h0
   -> (i0 -> ((h0 -> (i0 -> j0)) -> j0))) -> ((k0 ->
    (l0 -> ((k0 -> (l0 -> m0)) -> m0))) -> g0)) ->
   g0)), two -
> Scheme(Set(u0, x0, q0, a1, b1, s0, e1, d1, z0, w0,
    y0, v0, t0, c1, r0),(((((t0 -> (u0 -> ((t0 -> (
   u0 -> x0)) -> x0))) -> ((r0 -> (v0 -> ((r0 -> (v0
    -> w0)) -> w0))) -> s0)) -> s0) -> ((((a1 -> (b1
    -> ((a1 -> (b1 -> e1)) -> e1))) -> ((y0 -> (c1
   -> ((y0 ->
 (c1 -> d1)) -> d1))) -> z0)) -> z0) -> q0)) -> q0))
    , three -> Scheme(Set(j2, i1, m1, c2, n1, r1, z1
    , g2, q1, s1, w1, t1, f2, x1, o1, j1, i2, a2, h2
    , b2, u1, v1, p1, k2, m2, l2, l1, y1, e2, k1, d2
    ),(((((((v1 -> (j1 -> ((v1 -> (j1 -> k1)) -> k1)
    )) -> ((x1 -
> (u1 -> ((x1 -> (u1 -> s1)) -> s1))) -> o1)) -> o1)
    -> ((((m1 -> (n1 -> ((m1 -> (n1 -> p1)) -> p1)))
    -> ((t1 -> (w1 -> ((t1 -> (w1 -> q1)) -> q1)))
   -> r1)) -> r1) -> l1)) -> l1) -> ((((((k2 -> (y1
   -> ((k2 -> (y1 -> z1)) -> z1))) -> ((m2 -> (j2 ->
    ((m2 ->
(j2 -> h2)) -> h2))) -> d2)) -> d2) -> ((((b2 -> (c2
    -> ((b2 -> (c2 -> e2)) -> e2))) -> ((i2 -> (l2
   -> ((i2 -> (l2 -> f2)) -> f2))) -> g2)) -> g2) ->
    a2)) -> a2) -> i1)) -> i1)))
Type vars = 66
Type vars in sub = 219
########## three ##########
########## four ##########
substitution set Map(o0 -> (((t0 -> (u0 -> ((t0 -> (
   u0 -> x0)) -> x0))) -> ((r0 -> (v0 -> ((r0 -> (v0
    -> w0)) -> w0))) -> s0)) -> s0), b5 -> (((((((((
   m3 -> (g3 -> ((m3 -> (g3 -> u3)) -> u3))) -> ((e3
    -> (l3 -> ((e3 -> (l3 -> a3)) -> a3))) -> f3))
   -> f3) ->
 ((((t2 -> (v2 -> ((t2 -> (v2 -> n3)) -> n3))) -> ((
    c3 -> (b3 -> ((c3 -> (b3 -> z2)) -> z2))) -> w2)
    ) -> w2) -> r3)) -> r3) -> ((((((o3 -> (s3 -> ((
```

```
    o3 -> (s3 -> x2)) -> x2))) -> ((p3 -> (r2 -> ((
    p3 -> (r2 -> j3)) -> j3))) -> v3)) -> v3) ->
    ((((k3 -> (u2 ->
 ((k3 -> (u2 -> t3)) -> t3))) -> ((h3 -> (q3 -> ((h3
     -> (q3 -> d3)) -> d3))) -> y2)) -> y2) -> i3))
    -> i3) -> s2)) -> s2) -> (((((((r4 -> (l4 -> ((
    r4 -> (l4 -> z4)) -> z4))) -> ((j4 -> (q4 -> ((
    j4 -> (q4 -> f4)) -> f4))) -> k4)) -> k4) ->
    ((((y3 -> (a4 ->
 ((y3 -> (a4 -> s4)) -> s4))) -> ((h4 -> (g4 -> ((h4
     -> (g4 -> e4)) -> e4))) -> b4)) -> b4) -> w4))
    -> w4) -> ((((((t4 -> (x4 -> ((t4 -> (x4 -> c4))
     -> c4))) -> ((u4 -> (w3 -> ((u4 -> (w3 -> o4))
    -> o4))) -> a5)) -> a5) -> ((((p4 -> (z3 -> ((p4
     -> (z3 -> y
4)) -> y4))) -> ((m4 -> (v4 -> ((m4 -> (v4 -> i4))
   -> i4))) -> d4)) -> d4) -> n4)) -> n4) -> x3)) ->
    x3) -> q2)) -> q2), o2 -> (((((((m3 -> (g3 -> ((
   m3 -> (g3 -> u3)) -> u3))) -> ((e3 -> (l3 -> ((e3
    -> (l3 -> a3)) -> a3))) -> f3)) -> f3) -> ((((t2
    -> (v2 -
> ((t2 -> (v2 -> n3)) -> n3))) -> ((c3 -> (b3 -> ((
   c3 -> (b3 -> z2)) -> z2))) -> w2)) -> w2) -> r3))
    -> r3) -> ((((((o3 -> (s3 -> ((o3 -> (s3 -> x2))
    -> x2))) -> ((p3 -> (r2 -> ((p3 -> (r2 -> j3))
   -> j3))) -> v3)) -> v3) -> ((((k3 -> (u2 -> ((k3
   -> (u2 ->
t3)) -> t3))) -> ((h3 -> (q3 -> ((h3 -> (q3 -> d3))
   -> d3))) -> y2)) -> y2) -> i3)) -> i3) -> s2)) ->
    s2), e0 -> (h0 -> (i0 -> ((h0 -> (i0 -> j0)) ->
   j0))), n2 -> (((((((v1 -> (j1 -> ((v1 -> (j1 ->
   k1)) -> k1))) -> ((x1 -> (u1 -> ((x1 -> (u1 -> s1
   )) -> s1)
)) -> o1)) -> o1) -> ((((m1 -> (n1 -> ((m1 -> (n1 ->
    p1)) -> p1))) -> ((t1 -> (w1 -> ((t1 -> (w1 ->
   q1)) -> q1))) -> r1)) -> r1) -> l1)) -> l1) ->
   ((((((k2 -> (y1 -> ((k2 -> (y1 -> z1)) -> z1)))
   -> ((m2 -> (j2 -> ((m2 -> (j2 -> h2)) -> h2))) ->
    d2)) -> d2)
 -> ((((b2 -> (c2 -> ((b2 -> (c2 -> e2)) -> e2))) ->
     ((i2 -> (l2 -> ((i2 -> (l2 -> f2)) -> f2))) ->
    g2)) -> g2) -> a2)) -> a2) -> i1)) -> i1), f0 ->
     (k0 -> (l0 -> ((k0 -> (l0 -> m0)) -> m0))), p0
    -> (((a1 -> (b1 -> ((a1 -> (b1 -> e1)) -> e1)))
```

```
    -> ((y0 -> (
55 c1 -> ((y0 -> (c1 -> d1)) -> d1))) -> z0)) -> z0),
   n0 -> (((h0 -> (i0 -> ((h0 -> (i0 -> j0)) -> j0))
   ) -> ((k0 -> (l0 -> ((k0 -> (l0 -> m0)) -> m0)))
   -> g0)) -> g0), c0 -> (a0 -> (b0 -> d0)), g1 ->
   (((((v1 -> (j1 -> ((v1 -> (j1 -> k1)) -> k1))) ->
    ((x1 -> (
56 u1 -> ((x1 -> (u1 -> s1)) -> s1))) -> o1)) -> o1) ->
    ((((m1 -> (n1 -> ((m1 -> (n1 -> p1)) -> p1))) ->
    ((t1 -> (w1 -> ((t1 -> (w1 -> q1)) -> q1))) ->
   r1)) -> r1) -> l1)) -> l1), h1 -> (((((k2 -> (y1
   -> ((k2 -> (y1 -> z1)) -> z1))) -> ((m2 -> (j2 ->
    ((m2 ->
57 (j2 -> h2)) -> h2))) -> d2)) -> d2) -> ((((b2 -> (c2
    -> ((b2 -> (c2 -> e2)) -> e2))) -> ((i2 -> (l2
   -> ((i2 -> (l2 -> f2)) -> f2))) -> g2)) -> g2) ->
    a2)) -> a2), p2 -> (((((((r4 -> (l4 -> ((r4 -> (
   l4 -> z4)) -> z4))) -> ((j4 -> (q4 -> ((j4 -> (q4
    -> f4))
58 -> f4))) -> k4)) -> k4) -> ((((y3 -> (a4 -> ((y3 ->
   (a4 -> s4)) -> s4))) -> ((h4 -> (g4 -> ((h4 -> (
   g4 -> e4)) -> e4))) -> b4)) -> b4) -> w4)) -> w4)
    -> ((((((t4 -> (x4 -> ((t4 -> (x4 -> c4)) -> c4)
   )) -> ((u4 -> (w3 -> ((u4 -> (w3 -> o4)) -> o4)))
    -> a5))
59 -> a5) -> ((((p4 -> (z3 -> ((p4 -> (z3 -> y4)) -> y4
   ))) -> ((m4 -> (v4 -> ((m4 -> (v4 -> i4)) -> i4))
   ) -> d4)) -> d4) -> n4)) -> n4) -> x3)) -> x3),
   f1 -> (((((t0 -> (u0 -> ((t0 -> (u0 -> x0)) -> x0
   ))) -> ((r0 -> (v0 -> ((r0 -> (v0 -> w0)) -> w0))
   ) -> s0))
60  -> s0) -> ((((a1 -> (b1 -> ((a1 -> (b1 -> e1)) ->
    e1))) -> ((y0 -> (c1 -> ((y0 -> (c1 -> d1)) ->
    d1))) -> z0)) -> z0) -> q0)) -> q0))
61 type (((((((((m3 -> (g3 -> ((m3 -> (g3 -> u3)) -> u3
   ))) -> ((e3 -> (l3 -> ((e3 -> (l3 -> a3)) -> a3))
   ) -> f3)) -> f3) -> ((((t2 -> (v2 -> ((t2 -> (v2
   -> n3)) -> n3))) -> ((c3 -> (b3 -> ((c3 -> (b3 ->
    z2)) -> z2))) -> w2)) -> w2) -> r3)) -> r3) ->
   ((((((o3
62 -> (s3 -> ((o3 -> (s3 -> x2)) -> x2))) -> ((p3 -> (
   r2 -> ((p3 -> (r2 -> j3)) -> j3))) -> v3)) -> v3)
    -> ((((k3 -> (u2 -> ((k3 -> (u2 -> t3)) -> t3)))
    -> ((h3 -> (q3 -> ((h3 -> (q3 -> d3)) -> d3)))
```

```
   -> y2)) -> y2) -> i3)) -> i3) -> s2)) -> s2) ->
   (((((((((r4
-> (l4 -> ((r4 -> (l4 -> z4)) -> z4))) -> ((j4 -> (
   q4 -> ((j4 -> (q4 -> f4)) -> f4))) -> k4)) -> k4)
    -> ((((y3 -> (a4 -> ((y3 -> (a4 -> s4)) -> s4)))
    -> ((h4 -> (g4 -> ((h4 -> (g4 -> e4)) -> e4)))
   -> b4)) -> b4) -> w4)) -> w4) -> ((((((t4 -> (x4
   -> ((t4 ->
 (x4 -> c4)) -> c4))) -> ((u4 -> (w3 -> ((u4 -> (w3
    -> o4)) -> o4))) -> a5)) -> a5) -> ((((p4 -> (z3
     -> ((p4 -> (z3 -> y4)) -> y4))) -> ((m4 -> (v4
    -> ((m4 -> (v4 -> i4)) -> i4))) -> d4)) -> d4)
    -> n4)) -> n4) -> x3)) -> x3) -> q2)) -> q2)
current env is Map(four -> Scheme(Set(s4, q4, y4, d3
   , t4, g3, r2, k3, w2, x3, y2, r3, c3, m4, i4, w3,
    v4, u4, u3, w4, r4, z2, i3, u2, y3, a5, s2, g4,
   f3, t2, n3, l3, v3, c4, f4, x4, x2, h4, j3, t3,
   z4, e3, m3, n4, h3, v2, s3, e4, o4, z3, k4, b4,
   b3, j4, a3
, q2, l4, q3, a4, p4, p3, d4, o3),(((((((((m3 -> (g3
    -> ((m3 -> (g3 -> u3)) -> u3))) -> ((e3 -> (l3
   -> ((e3 -> (l3 -> a3)) -> a3))) -> f3)) -> f3) ->
    ((((t2 -> (v2 -> ((t2 -> (v2 -> n3)) -> n3))) ->
    ((c3 -> (b3 -> ((c3 -> (b3 -> z2)) -> z2))) ->
   w2)) -> w2
) -> r3)) -> r3) -> ((((((o3 -> (s3 -> ((o3 -> (s3
   -> x2)) -> x2))) -> ((p3 -> (r2 -> ((p3 -> (r2 ->
    j3)) -> j3))) -> v3)) -> v3) -> ((((k3 -> (u2 ->
    ((k3 -> (u2 -> t3)) -> t3))) -> ((h3 -> (q3 ->
   ((h3 -> (q3 -> d3)) -> d3))) -> y2)) -> y2) -> i3
   )) -> i3)
-> s2)) -> s2) -> (((((((((r4 -> (l4 -> ((r4 -> (l4
   -> z4)) -> z4))) -> ((j4 -> (q4 -> ((j4 -> (q4 ->
    f4)) -> f4))) -> k4)) -> k4) -> ((((y3 -> (a4 ->
    ((y3 -> (a4 -> s4)) -> s4))) -> ((h4 -> (g4 ->
   ((h4 -> (g4 -> e4)) -> e4))) -> b4)) -> b4) -> w4
   )) -> w4)
-> ((((((t4 -> (x4 -> ((t4 -> (x4 -> c4)) -> c4)))
   -> ((u4 -> (w3 -> ((u4 -> (w3 -> o4)) -> o4))) ->
    a5)) -> a5) -> ((((p4 -> (z3 -> ((p4 -> (z3 ->
   y4)) -> y4))) -> ((m4 -> (v4 -> ((m4 -> (v4 -> i4
   )) -> i4))) -> d4)) -> d4) -> n4)) -> n4) -> x3))
    -> x3) ->
```

```
70  q2)) -> q2)), three -> Scheme(Set(j2, i1, m1, c2,
       n1, r1, z1, g2, q1, s1, w1, t1, f2, x1, o1, j1,
       i2, a2, h2, b2, u1, v1, p1, k2, m2, l2, l1, y1,
       e2, k1, d2),(((((((v1 -> (j1 -> ((v1 -> (j1 ->
       k1)) -> k1))) -> ((x1 -> (u1 -> ((x1 -> (u1 ->
       s1)) -> s1))) -
71  > o1)) -> o1) -> ((((m1 -> (n1 -> ((m1 -> (n1 -> p1)
       ) -> p1))) -> ((t1 -> (w1 -> ((t1 -> (w1 -> q1))
       -> q1))) -> r1)) -> r1) -> l1)) -> l1) -> ((((((
       k2 -> (y1 -> ((k2 -> (y1 -> z1)) -> z1))) -> ((m2
        -> (j2 -> ((m2 -> (j2 -> h2)) -> h2))) -> d2))
       -> d2) ->
72  ((((b2 -> (c2 -> ((b2 -> (c2 -> e2)) -> e2))) -> ((
       i2 -> (l2 -> ((i2 -> (l2 -> f2)) -> f2))) -> g2))
        -> g2) -> a2)) -> a2) -> i1)) -> i1)), two ->
       Scheme(Set(u0, x0, q0, a1, b1, s0, e1, d1, z0, w0
       , y0, v0, t0, c1, r0),(((((t0 -> (u0 -> ((t0 -> (
       u0 -> x0))
73  -> x0))) -> ((r0 -> (v0 -> ((r0 -> (v0 -> w0)) -> w0
       ))) -> s0)) -> s0) -> ((((a1 -> (b1 -> ((a1 -> (
       b1 -> e1)) -> e1))) -> ((y0 -> (c1 -> ((y0 -> (c1
        -> d1)) -> d1))) -> z0)) -> z0) -> q0)) -> q0)),
        tuple -> Scheme(Set(a0, b0, d0),(a0 -> (b0 -> ((
       a0 -> (b0
74   -> d0)) -> d0)))), one -> Scheme(Set(k0, g0, h0, i0
       , l0, m0, j0),(((h0 -> (i0 -> ((h0 -> (i0 -> j0)
       ) -> j0))) -> ((k0 -> (l0 -> ((k0 -> (l0 -> m0))
        -> m0))) -> g0)) -> g0)))
75  Type vars = 132
76  Type vars in sub = 472
77  ########## four ##########
78  ########## main ##########
79  substitution set Map(o0 -> (((t0 -> (u0 -> ((t0 -> (
       u0 -> x0)) -> x0))) -> ((r0 -> (v0 -> ((r0 -> (v0
        -> w0)) -> w0))) -> s0)) -> s0), b5 -> (((((((((
       m3 -> (g3 -> ((m3 -> (g3 -> u3)) -> u3))) -> ((e3
        -> (l3 -> ((e3 -> (l3 -> a3)) -> a3))) -> f3))
       -> f3) ->
80   ((((t2 -> (v2 -> ((t2 -> (v2 -> n3)) -> n3))) -> ((
       c3 -> (b3 -> ((c3 -> (b3 -> z2)) -> z2))) -> w2)
       ) -> w2) -> r3)) -> r3) -> ((((((o3 -> (s3 -> ((
       o3 -> (s3 -> x2)) -> x2))) -> ((p3 -> (r2 -> ((
       p3 -> (r2 -> j3)) -> j3))) -> v3)) -> v3) ->
       ((((k3 -> (u2 ->
```

```
81  ((k3 -> (u2 -> t3)) -> t3))) -> ((h3 -> (q3 -> ((h3
       -> (q3 -> d3)) -> d3))) -> y2)) -> y2) -> i3))
     -> i3) -> s2)) -> s2) -> (((((((r4 -> (l4 -> ((
     r4 -> (l4 -> z4)) -> z4))) -> ((j4 -> (q4 -> ((
     j4 -> (q4 -> f4)) -> f4))) -> k4)) -> k4) ->
     ((((y3 -> (a4 ->
82  ((y3 -> (a4 -> s4)) -> s4))) -> ((h4 -> (g4 -> ((h4
       -> (g4 -> e4)) -> e4))) -> b4)) -> b4) -> w4))
     -> w4) -> ((((((t4 -> (x4 -> ((t4 -> (x4 -> c4))
       -> c4))) -> ((u4 -> (w3 -> ((u4 -> (w3 -> o4))
     -> o4))) -> a5)) -> a5) -> ((((p4 -> (z3 -> ((p4
       -> (z3 -> y
83  4)) -> y4))) -> ((m4 -> (v4 -> ((m4 -> (v4 -> i4))
    -> i4))) -> d4)) -> d4) -> n4)) -> n4) -> x3)) ->
       x3) -> q2)) -> q2), o2 -> (((((((m3 -> (g3 -> ((
    m3 -> (g3 -> u3)) -> u3))) -> ((e3 -> (l3 -> ((e3
       -> (l3 -> a3)) -> a3))) -> f3)) -> f3) -> ((((t2
       -> (v2 -
84  > ((t2 -> (v2 -> n3)) -> n3))) -> ((c3 -> (b3 -> ((
    c3 -> (b3 -> z2)) -> z2))) -> w2)) -> w2) -> r3))
       -> r3) -> ((((((o3 -> (s3 -> ((o3 -> (s3 -> x2))
       -> x2))) -> ((p3 -> (r2 -> ((p3 -> (r2 -> j3))
    -> j3))) -> v3)) -> v3) -> ((((k3 -> (u2 -> ((k3
    -> (u2 ->
85  t3)) -> t3))) -> ((h3 -> (q3 -> ((h3 -> (q3 -> d3))
    -> d3))) -> y2)) -> y2) -> i3)) -> i3) -> s2)) ->
       s2), e0 -> (h0 -> (i0 -> ((h0 -> (i0 -> j0)) ->
    j0))), n2 -> (((((((v1 -> (j1 -> ((v1 -> (j1 ->
    k1)) -> k1))) -> ((x1 -> (u1 -> ((x1 -> (u1 -> s1
    )) -> s1)
86  )) -> o1)) -> o1) -> ((((m1 -> (n1 -> ((m1 -> (n1 ->
       p1)) -> p1))) -> ((t1 -> (w1 -> ((t1 -> (w1 ->
    q1)) -> q1))) -> r1)) -> r1) -> l1)) -> l1) ->
    ((((((k2 -> (y1 -> ((k2 -> (y1 -> z1)) -> z1)))
    -> ((m2 -> (j2 -> ((m2 -> (j2 -> h2)) -> h2))) ->
       d2)) -> d2)
87   -> ((((b2 -> (c2 -> ((b2 -> (c2 -> e2)) -> e2))) ->
        ((i2 -> (l2 -> ((i2 -> (l2 -> f2)) -> f2))) ->
     g2)) -> g2) -> a2)) -> a2) -> i1)) -> i1), f0 ->
        (k0 -> (l0 -> ((k0 -> (l0 -> m0)) -> m0))), p0
     -> (((a1 -> (b1 -> ((a1 -> (b1 -> e1)) -> e1)))
     -> ((y0 -> (
88  c1 -> ((y0 -> (c1 -> d1)) -> d1))) -> z0)) -> z0),
    n0 -> (((h0 -> (i0 -> ((h0 -> (i0 -> j0)) -> j0))
```

```
   ) -> ((k0 -> (l0 -> ((k0 -> (l0 -> m0)) -> m0)))
   -> g0)) -> g0), c0 -> (a0 -> (b0 -> d0)), g1 ->
   (((((v1 -> (j1 -> ((v1 -> (j1 -> k1)) -> k1))) ->
    ((x1 -> (
u1 -> ((x1 -> (u1 -> s1)) -> s1))) -> o1)) -> o1) ->
    ((((m1 -> (n1 -> ((m1 -> (n1 -> p1)) -> p1))) ->
    ((t1 -> (w1 -> ((t1 -> (w1 -> q1)) -> q1))) ->
   r1)) -> r1) -> l1)) -> l1), h1 -> (((((k2 -> (y1
   -> ((k2 -> (y1 -> z1)) -> z1))) -> ((m2 -> (j2 ->
    ((m2 ->
(j2 -> h2)) -> h2))) -> d2)) -> d2) -> ((((b2 -> (c2
    -> ((b2 -> (c2 -> e2)) -> e2))) -> ((i2 -> (l2
   -> ((i2 -> (l2 -> f2)) -> f2))) -> g2)) -> g2) ->
    a2)) -> a2), p2 -> (((((((r4 -> (l4 -> ((r4 -> (
   l4 -> z4)) -> z4))) -> ((j4 -> (q4 -> ((j4 -> (q4
    -> f4))
-> f4))) -> k4)) -> k4) -> ((((y3 -> (a4 -> ((y3 ->
   (a4 -> s4)) -> s4))) -> ((h4 -> (g4 -> ((h4 -> (
   g4 -> e4)) -> e4))) -> b4)) -> b4) -> w4)) -> w4)
    -> ((((((t4 -> (x4 -> ((t4 -> (x4 -> c4)) -> c4)
   )) -> ((u4 -> (w3 -> ((u4 -> (w3 -> o4)) -> o4)))
    -> a5))
-> a5) -> ((((p4 -> (z3 -> ((p4 -> (z3 -> y4)) -> y4
   ))) -> ((m4 -> (v4 -> ((m4 -> (v4 -> i4)) -> i4))
   ) -> d4)) -> d4) -> n4)) -> n4) -> x3)) -> x3),
   f1 -> (((((t0 -> (u0 -> ((t0 -> (u0 -> x0)) -> x0
   ))) -> ((r0 -> (v0 -> ((r0 -> (v0 -> w0)) -> w0))
   ) -> s0))
 -> s0) -> ((((a1 -> (b1 -> ((a1 -> (b1 -> e1)) ->
    e1))) -> ((y0 -> (c1 -> ((y0 -> (c1 -> d1)) ->
    d1))) -> z0)) -> z0) -> q0)) -> q0))
type Int
current env is Map(four -> Scheme(Set(s4, q4, y4, d3
   , t4, g3, r2, k3, w2, x3, y2, r3, c3, m4, i4, w3,
    v4, u4, u3, w4, r4, z2, i3, u2, y3, a5, s2, g4,
   f3, t2, n3, l3, v3, c4, f4, x4, x2, h4, j3, t3,
   z4, e3, m3, n4, h3, v2, s3, e4, o4, z3, k4, b4,
   b3, j4, a3
, q2, l4, q3, a4, p4, p3, d4, o3),(((((((((m3 -> (g3
    -> ((m3 -> (g3 -> u3)) -> u3))) -> ((e3 -> (l3
   -> ((e3 -> (l3 -> a3)) -> a3))) -> f3)) -> f3) ->
    ((((t2 -> (v2 -> ((t2 -> (v2 -> n3)) -> n3))) ->
    ((c3 -> (b3 -> ((c3 -> (b3 -> z2)) -> z2))) ->
   w2)) -> w2
```

```
) -> r3)) -> r3) -> ((((((o3 -> (s3 -> ((o3 -> (s3
   -> x2)) -> x2))) -> ((p3 -> (r2 -> ((p3 -> (r2 ->
    j3)) -> j3))) -> v3)) -> v3) -> ((((k3 -> (u2 ->
    ((k3 -> (u2 -> t3)) -> t3))) -> ((h3 -> (q3 ->
   ((h3 -> (q3 -> d3)) -> d3))) -> y2)) -> y2) -> i3
   )) -> i3)
-> s2)) -> s2) -> (((((((r4 -> (l4 -> ((r4 -> (l4
   -> z4)) -> z4))) -> ((j4 -> (q4 -> ((j4 -> (q4 ->
    f4)) -> f4))) -> k4)) -> k4) -> ((((y3 -> (a4 ->
    ((y3 -> (a4 -> s4)) -> s4))) -> ((h4 -> (g4 ->
   ((h4 -> (g4 -> e4)) -> e4))) -> b4)) -> b4) -> w4
   )) -> w4)
-> ((((((t4 -> (x4 -> ((t4 -> (x4 -> c4)) -> c4)))
   -> ((u4 -> (w3 -> ((u4 -> (w3 -> o4)) -> o4))) ->
    a5)) -> a5) -> ((((p4 -> (z3 -> ((p4 -> (z3 ->
   y4)) -> y4))) -> ((m4 -> (v4 -> ((m4 -> (v4 -> i4
   )) -> i4))) -> d4)) -> d4) -> n4)) -> n4) -> x3))
    -> x3) ->
 q2)) -> q2)), three -> Scheme(Set(j2, i1, m1, c2,
    n1, r1, z1, g2, q1, s1, w1, t1, f2, x1, o1, j1,
    i2, a2, h2, b2, u1, v1, p1, k2, m2, l2, l1, y1,
    e2, k1, d2),(((((((v1 -> (j1 -> ((v1 -> (j1 ->
    k1)) -> k1))) -> ((x1 -> (u1 -> ((x1 -> (u1 ->
    s1)) -> s1))) -
> o1)) -> o1) -> ((((m1 -> (n1 -> ((m1 -> (n1 -> p1)
   ) -> p1))) -> ((t1 -> (w1 -> ((t1 -> (w1 -> q1))
   -> q1))) -> r1)) -> r1) -> l1)) -> l1) -> ((((((
   k2 -> (y1 -> ((k2 -> (y1 -> z1)) -> z1))) -> ((m2
    -> (j2 -> ((m2 -> (j2 -> h2)) -> h2))) -> d2))
   -> d2) ->
((((b2 -> (c2 -> ((b2 -> (c2 -> e2)) -> e2))) -> ((
   i2 -> (l2 -> ((i2 -> (l2 -> f2)) -> f2))) -> g2))
    -> g2) -> a2)) -> a2) -> i1)) -> i1)), two ->
   Scheme(Set(u0, x0, q0, a1, b1, s0, e1, d1, z0, w0
   , y0, v0, t0, c1, r0),(((((t0 -> (u0 -> ((t0 -> (
   u0 -> x0))
-> x0))) -> ((r0 -> (v0 -> ((r0 -> (v0 -> w0)) -> w0
   ))) -> s0)) -> s0) -> ((((a1 -> (b1 -> ((a1 -> (
   b1 -> e1)) -> e1))) -> ((y0 -> (c1 -> ((y0 -> (c1
    -> d1)) -> d1))) -> z0)) -> z0) -> q0)) -> q0)),
    main -> Scheme(Set(),Int), tuple -> Scheme(Set(
   a0, b0, d0
),(a0 -> (b0 -> ((a0 -> (b0 -> d0)) -> d0)))), one
   -> Scheme(Set(k0, g0, h0, i0, l0, m0, j0),(((h0
```

```
   -> (i0 -> ((h0 -> (i0 -> j0)) -> j0))) -> ((k0 ->
    (l0 -> ((k0 -> (l0 -> m0)) -> m0))) -> g0)) ->
   g0)))
Type vars = 132
Type vars in sub = 472
########## main ##########
```