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

0.1 Notation

0.1.1 Functions and implementations

Some functions are written in pattern matching style. Functions which have parameters that can take different values or shapes can implement a case for each value or shape. For instance, a function which finds the cardinality of a set can be implemented as in Equation 1.

$$
\begin{aligned}
\texttt{card}(\emptyset) &\texttt{ = 0} \\
\texttt{card}(\{\texttt{x}\} \cup \texttt{S}) &\texttt{ = 1 + card(S)}
\end{aligned} \tag{1}
$$

Inevitably, to allow complicated functions, like algorithms, some functions may require subexpressions. Subexpressions are denoted with `where` when some expression is a composite of multiple expressions like in Equation 2.

$$
\begin{aligned}
&\texttt{fib(0) = 0} \\
&\texttt{fib(1) = 1} \\
&\texttt{fib(n) = } \texttt{F}_{\texttt{n-1}} \texttt{ + } \texttt{F}_{\texttt{n-2}} \\
&\qquad \texttt{where } \texttt{F}_{\texttt{n-1}} \texttt{ = fib(n - 1)}, \\
&\qquad\qquad \texttt{F}_{\texttt{n-2}} \texttt{ = fib(n - 2)}
\end{aligned} \tag{2}
$$

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 [Cop97]. 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[Chu36]. 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 1.2 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 in which x may occur. The *variable* (Equation 1.1) is a reference to some value introduced by an abstraction. A variable is a reference to another lambda abstraction. In the untyped 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 the untyped lambda calculus which will be further discussed when introducing typing in section 2.1.

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 section 3.3.

A remarkable fact about the untyped lambda calculus is that it is 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 [Chu85]. 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 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 Listing 1.1 and Listing 1.2 that both add an `a` and a `b`.

Listing 1.1: Add function in lambda calculus

```
(λadd.E)(λa.λb.a + b)
```

Listing 1.2: Add function in L

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

Notice that in Listing 1.1 the term E is left undefined, E is "the rest of the program in this scope". If the program were to apply 1 and 2 to add the resulting program in L would be Listing 1.4 and in the untyped lambda calculus it would be Listing 1.3.

Listing 1.3: Add function in lambda calculus

```
(λadd.add 1 2)(λa.λb.a + b)
```

Listing 1.4: Add function applied

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

1.2.1 Scoping

Notice that Listing 1.1 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 1.5 functions must be explicitly named to translate as in the above example.

Listing 1.5: A traditional program

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

Listing 1.6: An order dependent program

```
fun sub a b = add a (0 - b);
fun add a b = a + b;
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 1.6 would not translate into a valid program, it would translate into Listing 1.7. The definition of `sub` is missing a reference to the `add` function.

Listing 1.7: Add function in lambda calculus

```
(λsub.λadd. sub (add 10 20) 5)
    (λa.λb.a + b)
    (λa.λb.add a (0 - b))
```

lambda lifting is a technique where free variables (section 1.1) are explicitly parameterized [Joh85]. This is exactly the problem in Listing 1.7 which has the lambda lifted solution Listing 1.8.

Listing 1.8: Order dependent

```
(λsub.λadd. sub add (add 10 20) 5)
    (λa.λb.a + b)
    (λadd.λa.λb.add a (0 - b))
```

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. The function `f` in Listing 1.9 yields `12`.

Listing 1.9: Scoping rules in programming languages

```
let x = 22;
let a = 10;
```

```
fun f =
  let x = 2;
  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 1.10: Infinite program

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

Listing 1.10 defines a function `f` that in fact is infinite. In 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 performed on this function such that it can evaluate *any* case of `n`?

Listing 1.11: Recursive program

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

Listing 1.11 defines a recursive variant of `f` it is a product of the reduction in Equation 1.8.

$$n + (n-1) \cdots + 0 = \sum_{k=0}^{n} k \tag{1.8}$$

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).

Listing 1.12: Recursive function

```
(λf.E) (λn.if (n == 0) (n) (n + (f (n - 1))))
```

The naive implementation of a recursive variant will yield an unsolvable problem which in fact is an infinite problem. In Listing 1.12 when `f` is applied recursively it must be referenced while it is "being constructed". Substituting f with its implementation in Listing 1.13 will yield the same problem again but at one level deeper. The `if` function takes a condition, the body in case of the condition being true and the body in case of the condition being false.

Listing 1.13: Recursive function f substituted

```
(λf.E)
(λn.if (n == 0) (n) (n + (
    (λn.if (n == 0) (n) (n + (f (n - 1))))
    (n - 1)
)))
```

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 referencing `f` from within `f`, `f` should be parameterized as in Listing 1.14 such that it translates to Listing 1.15.

Listing 1.14: Explicitly passing recursive function

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

Listing 1.15: Explicitly passing recursive function in the lambda calculus

```
(λf.E)(λf.λn. if (n == 0) (n) (n + (f f (n - 1))))
```

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 1.9 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.

$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \tag{1.9}$$

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

Listing 1.16: Mutual recursion

```
fun g x = f x;
fun f x = g x;
```

Notice in Listing 1.16 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 different 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 1.17: List algebraic data type

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

Listing 1.17 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 1.18: 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 1.18).

Scott encoding

Pattern matching strays far from the simple untyped lambda calculus, but can in fact be encoded into it. The *scott encoding* (Equation 1.10) is a technique that describes a general purpose framework to encode algebraic data types into lambda calculus [Sco62]. 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 $x_1 \ldots x_{A_i}$ 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 $x_1 \ldots x_{A_i}$ 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.

$$\lambda x_1 \ldots x_{A_i}.\lambda c_1 \ldots c_n.c_i x_1 \ldots x_{A_i} \tag{1.10}$$

Example 1.3.1. The `List` algebraic data type in Listing 1.17 has two constructors, `Nil` with the constructor type Equation 1.11 and `Cons` with the constructor type Equation 1.12. Equation 1.11 is in fact also the type of `List` once instantiated, effetively treating partially applied functions as data.

$$b \to (a \to \texttt{List}\ \ a \to b) \to b \tag{1.11}$$

$$(a \to \texttt{List}\ \ a \to b) \to b \to (a \to \texttt{List}\ \ a \to b) \to b \tag{1.12}$$

Listing 1.19: List algebraic data type implementation

```
fun cons x xs =
    fun c _ onCons = onCons x xs;
    c;

fun nil =
    fun c onNil _ = onNil;
    c;
```

Encoding the constructors in L yields the functions defined in Listing 1.19. Pattern matching is but a matter of applying the appropriate handlers. In Listing 1.20.

Listing 1.20: Example of scott encoded list algebraic data type

```
let l = cons 1 (cons 2 (cons 3 nil));

fun head x _ =
    x;

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 1.13).

$$\beta_{red}((\lambda x.T)E) = T[x := E] \tag{1.13}$$

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 can take properties that make them very powerful since types are, in their essence, a set of logical formulas in which we can use natural deduction to prove their validity [How80].

2.1.1 Notation

In the world of type theory we use natural deduction to prove type validity. Natural deduction is expressed by *inference rules*, which consist of one or more premises and a conclusion. For instance the modus ponens rule which states that if "if `a` implies `b` and `a`, then `b`", can easily be written in inference rules, where `a` $\rightarrow$ `b` and `a` are the permises and `b` is the conclusion (Figure 2.1). Conditions can also occur rules which state what conditions must be met for the rule to apply (Figure 2.2). Furthermore, hypotheses are written $\Gamma \vdash p$ which states that under the assumption of Γ then p.

$$\frac{\mathtt{a} \rightarrow \mathtt{b} \qquad \mathtt{a}}{\mathtt{b}}$$

Figure 2.1

$$\frac{\texttt{P} \qquad \texttt{P}\text{ is not dead}}{\texttt{P}\text{ is alive}}$$

Figure 2.2

2.1.2 The language of types

The untyped lambda calculus is exactly that, untyped. For it to become typed, we must introduce how types occur. A very simple type system for the how typed lambda calculus can be proved must introduce a rule for each of the lambda calculus term types Var, Let, App and Abs. Types must occur in programs to prove correctness, as such, program variables and types are the assumption for a proof of such a program's composite expressions such that the assumption for types will become a set the program variables $\{x_1 \ldots x_n\}$ paired with their respective type $\Gamma = \{(x_1 : \tau_1)\}, \ldots (x_n : \tau_n)\}$. Stating "it is assumed that a variable x of type τ occurs in Γ" is written $\Gamma, x : \tau$.

With the aforementioned knowledge, we can develop a simple set of inference rules which can be used to prove programs with types (Figure 2.3).

Example 2.1.1. With the rules for the simply typed lambda calculus, it becomes possible to prove the types for programs. Let $\lambda f.\lambda x.fx$ be a program with the type $(\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)$, the proof to which is seen in Figure 2.4.

The simply typed lambda calculus is straightforward, but is missing some ingredients that most programming language users cannot do without, namely *polymorphism.* For instance, in the simply typed lambda calculus one would have to define an identity function **for each** different type that uses it. If one were to bind an identity function to *id*, say $\texttt{let}\ id = \lambda x.x\ \texttt{in}\ \ldots$ with type $\tau_1 \to \tau_1$, where τ_1 is not determined yet. Clearly some $f : \tau_2 \to \tau_3$ and some $y : \tau_3$ cannot both be applied to *id* since if $\tau_3 \equiv \tau_1$ and $(\tau_2 \to \tau_3) \equiv \tau_1$ then an infinite type must exist $\tau_2 \to (\tau_2 \to (\tau_2 \ldots))$. What we actually want is to say that τ_1 can become **any** type τ_4 if for every application, every instance of τ_1 is replaced by τ_4 in $\tau_1 \to \tau_1$. As such, when applying $id\, f$ then the type of *id* must become $(\tau_2 \to \tau_3) \to (\tau_2 \to \tau_3)$, but only for this application. More generally the type for *id* becomes *generalized* and as such has the universally quantified type $\forall \tau_1.\tau_1 \to \tau_1$.

Generally, introducing polymorphism directly to the simply typed lambda calculus lifts the type system to one called System F. System F in undecidable, and as such, the Hindley-Milner type system will be of interest instead.

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

$$\text{Abs}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

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

- Var states that if x has type τ in Γ then it is assumed that x has type τ.
- App states that if e_1 can be proved to have type $\tau_1 \to \tau_2$ and e_2 can be proved to have type τ_1, then $e_1 e_2$ must be of type τ_2.
- Abs states that if e has type τ_2 under the assumption that x has some type τ_1, then $\lambda x.e$ must be of type $\tau_1 \to \tau_2$.
- Let does not yet have an important role, since let does the exact same as a combination of App and Abs. Once polymorphism is introduced, Let will play an important role. Currently Let states that if e_1 has type τ_1, and e_2 has type τ_2 under the assumption that x has type τ_1 (from the proof that $e_1 : \tau_1$ since $x = e_1$) then $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ must have inhabit the type τ_2.

Figure 2.3: A simple set of rules for the simply typed lambda calculus

$$
\text{Abs}\,\dfrac{\text{Abs}\,\dfrac{\dfrac{\dfrac{f : \tau_1 \to \tau_2 \in \{(x : \tau_1), (f : \tau_1 \to \tau_2)\}}{\{(x : \tau_1), (f : \tau_1 \to \tau_2)\} \vdash f : \tau_1 \to \tau_2}\,\text{Var} \qquad \dfrac{x : \tau_1 \in \{(x : \tau_1), (f : \tau_1 \to \tau_2)\}}{\{(x : \tau_1), (f : \tau_1 \to \tau_2)\} \vdash x : \tau_1}\,\text{Var}}{\{(x : \tau_1), (f : \tau_1 \to \tau_2)\} \vdash fx : \tau_2}}{\{(f : \tau_1 \to \tau_2)\} \vdash (\lambda x.fx) : \tau_1 \to \tau_2}}{\{\} \vdash \lambda f.(\lambda x.fx) : (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)}
$$

Figure 2.4: The proof for $\lambda f.\lambda x.fx$

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 2.1).

$$
\textit{mono } \tau = a \,|\, \tau \to \tau \,|\, C\tau_1 \ldots \tau_n \tag{2.1}
$$

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 constructors `ADT` encapsulates the set of program defined algebraic data structures (**??**).

Example 2.1.2. Let $\texttt{ADT} = \{\texttt{List}\}$ where `List` is defined as in Listing 1.17. The *type constructor* (not to be confused for constructors like `Cons` or `Nil`) for `List` has the signature $\texttt{a} \to \texttt{List a}$ 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 2.2).

$$
\textit{poly } \sigma = \tau \,|\, \forall a.\sigma \tag{2.2}
$$

Polymorphic types either take the shape of a type variable or universally quantify some type, naming the quantifier a. This does not necessarily include *all* types since the **Gen** rule of Figure 2.6 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 type hierarchy of the Hindley-Milner type-system is shown in Figure 2.5. The lack of σ in Figure 2.5 since is controversial to introduce to the

$$\begin{tikzcd}
& \top \arrow[dl] \arrow[d] \arrow[dr] & \\
\tau_1 \to \tau_2 \arrow[dr] & \tau \arrow[d] & C\tau_1 \dots \tau_n \arrow[dl] \\
& \bot &
\end{tikzcd}$$

Figure 2.5: The type hierarchy of Hindley-Milner.

type hierarchy; σ is but a mechanism to prove type systems.

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 2.3). $\Gamma \vdash x : \sigma$ 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 $\forall a.a \to a$, $\texttt{Int} \to \texttt{Int}$ or $\forall a.(a \to a) \to (a \to a)$.

$$\Gamma \ = \epsilon \,|\, \Gamma, x : \sigma \tag{2.3}$$

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.

$$\mathit{free}(a) = \{a\} \tag{2.4}$$

$$\mathit{free}(C\tau_1 \dots \tau_n) = \bigcup_{i=1}^{n} \mathit{free}(\tau_i) \tag{2.5}$$

$$\mathit{free}(\tau_1 \to \tau_2) = \mathit{free}(\tau_1) \cup \mathit{free}(\tau_2) \tag{2.6}$$

$$\mathit{free}(\Gamma) = \bigcup_{x:\sigma\in\Gamma} \mathit{free}(\sigma) \tag{2.7}$$

$$\mathit{free}(\forall a.\sigma) = \mathit{free}(\sigma) - \{a\} \tag{2.8}$$

Example 2.1.3. Consider the type for the function `fst` in Listing 2.2.

Listing 2.1: First function

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

Listing 2.2: First function in lambda calculus

```
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 2.12, 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.

$$\mathit{bound}(\tau) = \mathit{free}(\tau) - \mathit{free}(\Gamma) \tag{2.9}$$

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

Example 2.1.4.

$$\Gamma = \{(x, \gamma)\} \tag{2.10}$$

$$\begin{aligned} \mathit{bound}(\tau \to \gamma) &= \{\tau, \gamma\} - \mathit{free}(\Gamma) \\ &= \{\tau, \gamma\} - \{\gamma\} = \{\tau\} \end{aligned} \tag{2.11}$$

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 2.10 does not contain γ as a quantified type since it has been introduced by an abstraction and **Abs** only introduces monomorphic types (Figure 2.6). 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 2.3 [Wad89].

Listing 2.3: Implementation of type state

```
λx.
    let z = (λy.x) in
    ...
```

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-

$$\text{Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

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

$$\text{Abs}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\text{Let}\ \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$$

$$\text{Inst}\ \frac{\Gamma \vdash e : \sigma_1 \qquad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash e : \sigma_2}$$

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

Figure 2.6: Hindley-Milner type rules

Milner rules outlined in Figure 2.6.

- **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 2.5), 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 the convenient syntax (x, z) is the pair of the variables x and z which can be implemented by algebraic data structures or a combinator.

The identity function is a common example to illustrate type systems (Listing 2.4).

Listing 2.4: Identity function in L

```
fun id x = x;
id 4;
```

Listing 2.5: Identity function in lambda calculus with let

```
let id = (λx.x) in
id 4
```

Stating that id has the type $\forall a.a \to a$ and 4 has the type `Int` is Listing 2.5 program correct? By applying the Hindley-Milner rules one can prove or disprove this statement. A correct proof of Listing 2.4 must be Figure 2.7.

Listing 2.6: Identity function in lambda calculus by abstraction

```
(λid.id 4)(λx.x)
```

Listing 2.5 and Listing 2.6 are two equivalent programs with slightly different proofs which raises the question of why the let expression is even needed. If Listing 2.5 and Listing 2.6 were to be slightly changed such that two new programs Listing 2.7 and Listing 2.8 were to be proved, Listing 2.8 would not be provable while Listing 2.7 would.

Listing 2.7: Identity function with two applications

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

Listing 2.8: Identity function with two applications as abstraction

```
(λid.(id 4, id id)(λx.x)
```

In Listing 2.8 `id` cannot adhere to polymorphism by **Abs** in Figure 2.6 whilst **Let** can.

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

(a)

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

(b)

$$\text{Let}\,\dfrac{2.7\text{b} \qquad 2.7\text{a}}{\Gamma \vdash \texttt{let}\ \text{id} = (\lambda x.x)\ \texttt{in}\ \text{id}\ 4\text{:}\ \texttt{Int}}$$

(c)

Figure 2.7: 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 [Dam84; DM82].

The Damas-Milner Algorithm W rules (Figure 2.10) introduce some new concepts such as *fresh variables*, *most general unifier*, and the *substitution*

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

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

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

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

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

Figure 2.8: Substitution semantics

set. Fresh variables are introduced by picking a variable that has not been picked before from the infinite set $\tau_1, \tau_2 \dots$. Fresh variables are introduced when unknown types are discovered and later unified. The substitution set is a mapping from type variables to types (Equation 2.12).

$$S = \{a_1 \mapsto \tau_1, a_2 \mapsto \tau_2 \dots, a_n \mapsto \tau_n\} \tag{2.12}$$

A substitution written ST where T is an arbitrary component of Hindley-Milner like an environment in which all type variables are substituted (Figure 2.8). 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.

$$S_1 \cdot S_2 = \{(a \mapsto S_1\tau) \mid (a \mapsto \tau) \in S_2)\} \cup S_1 \tag{2.13}$$

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 2.1). Note that the Var rules for most general unifier outlined in Figure 2.9 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

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

$$\text{Intro}\ \frac{\tau_1, \tau_2}{\emptyset, \{(\tau_1, \tau_2)\}}$$

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

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

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

Figure 2.9: Rules for most general unification

solver approach which is also used in OCaml [HHS02]. 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.

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

2.2.2 Instantiation

Another interesting addition introduced by algorithm W in Figure 2.10 is *inst*. *inst* naturally follows from the **Inst** rule in Figure 2.6 but has a slightly different behaviour. The *inst* function does not specify types anymore but simply makes unification of polymorphic types possible.

$$\mathit{inst}(\sigma) = \{\, a \mapsto \mathit{fresh} \mid a \notin \mathit{free}(\sigma) \,\}\sigma \tag{2.14}$$

inst (Equation 2.14) 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 2.9 yields a very rich example of why *inst* is necessary.

$$\text{Var}\ \frac{x : \sigma \in \Gamma \qquad \tau = \mathit{inst}(\sigma)}{\Gamma \vdash x : \tau, \emptyset}$$

$$\text{Abs}\ \frac{\tau_1 = \mathit{fresh} \qquad \Gamma, x : \tau_1 \vdash e : \tau_2, S}{\Gamma \vdash \lambda x.e : S\tau_1 \to \tau_2, S}$$

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

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

Figure 2.10: Algorithm W

Listing 2.9: Polymorphic id

```
fun id x = x;
fun ap x f = f x;
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*; begin by looking at the introduced parameter `x` and then the innermost expression `id (x + 1)`.

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

$$\Gamma = \{(\texttt{id}, \forall a.a \to a), (\texttt{ap}, \forall \gamma, \beta.\gamma \to (\gamma \to \beta) \to \beta), (\texttt{x}, \forall \tau.\tau)\} \tag{2.15}$$

$$\mathit{unify}(\tau, \texttt{Int}) = \{\tau \mapsto \texttt{Int}\} \tag{2.16}$$

$$\mathit{unify}(a \to a, \texttt{Int} \to \mu) = \mathit{unify}(\{a \mapsto \texttt{Int}\}a, \{a \mapsto \texttt{Int}\}\mu) \cdot \{a \mapsto \texttt{Int}\} \tag{2.17}$$

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

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).

$$
\begin{aligned}
&\mathit{unify}(\mathit{inst}(\forall a.a \to a), \mathit{inst}(\forall \tau.\tau \to \mu)) \\
&= \mathit{unify}(\gamma \to \gamma, \varphi \to \mu) \\
&= \{\varphi \mapsto \mu, \gamma \mapsto \mu\}
\end{aligned} \tag{2.18}
$$

2.2.3 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 2.10 were to be typed it would need to be introduced as an unknown type to the environment before typing the body of `f`.

Listing 2.10: Recursive function

```
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`.

2.2.4 Additional language features

In addition to the rules in Figure 2.10 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 2.11: ADT implementation of decision

```
type Boolean =
    | BFalse
    | BTrue
;
let b = BFalse;
fun double x = x + x;
b (0) (double 10);
```

Decision can be implemented in a variety of ways such as in Listing 2.11 by the use of algebraic data structures aligning very much with Church Booleans [Chu85]. Rather decision can be implemented by more conven-

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

(a)

$$\Gamma \vdash e_1 : \tau_1, S_1 \;\; S_1' = \mathit{mgu}(\texttt{Bool}, \tau_1)$$

(b)

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

(c)

$$\frac{2.11\text{b} \qquad \dfrac{2.11\text{a} \qquad 2.11\text{c}}{\Gamma, S_6 \;\; S_6 = S_5 \cdot S_4 \cdot S_1' \cdot S_2 \cdot S_3}}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : S_6\tau_4, S_6}$$

Figure 2.11: Decision

tional methods than combinator logic by introducing more inference rules as in Figure 2.11. Additional language syntax features can in most cases be implemented as decision can.

2.2.5 Algebraic data structures

To implement rules for algebraic data structures one must first decide on what the type of an algebraic data structure is. If algebraic data structures were implemented as in subsection 1.3.1 the type of an algebraic data structure like Boolean in Listing 2.11 would be `a -> a -> a` since `BFalse` and `BTrue` must have a handler each. Implementing algebraic data structures by this method does not introduce anything to the inference algorithm since every algebraic data structure becomes a function. It is more common to introduce algebraic data structures as new type constructor types since such an implementation yields descriptive errors in comparison to generated function types. Fortunately type constructors are trivial to model in Hindley-Milner since a type constructor is a type lambda and rank-1 type lambdas are simply quantified types. Listing 1.17 introduces the type constructor $\texttt{a} \rightarrow \texttt{List a}$ with value constructors $\texttt{Cons:}\ \ \forall\texttt{a.a} \rightarrow \texttt{List a} \rightarrow \texttt{List a}$ and $\texttt{Nil:}\ \ \forall\texttt{a.List a}$. For instance let some program be `Cons 1 (Cons 2 Nil)` such that inference becomes a matter of first unifying the type of `Cons` and $\forall\texttt{a.Int} \rightarrow \texttt{List a} \rightarrow \tau_1$ using the application rule and then unifying $\forall\texttt{a.Int} \rightarrow \tau_1 \rightarrow \tau_2$ with `Cons` again by application.

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 leads to some unknown but large blowup in compilation time. In fact type polymorphism is commonly the root of blowup in typing.

ML, which implements a Hindley-Milner inference system, was believed to have linear complexity before shown to be exponential along with other problematic complexity findings [Mai89]. As it will turn out, Hindley-Milner also suffers an explosive worst case induced by a pathological input fueled by polymorphism.

Lemma 2.3.1. *There exists a family of programs which are typable in Hindley-Milner and produce $\Omega(2^n)$ unique type variables.*

Proof. The basis of the blowup stems from the introduced fresh type variables in the polymorphic `Let` inference rule. If the number 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 2.12: Nested dup

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

Listing 2.12 builds a large function signature for `deep`. The innermost `dup` invocation will have its signature unified to $x \to (x \to x \to \tau) \to \tau$, if `a` has type x and `f` has type $x \to x \to \tau$ for some unknown τ by the App rule in Figure 2.10. The second innermost `dup` 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 2.12 to run in $\Omega(2^n)$ but an important observation for why Listing 2.12 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 number introduced type variables would be $O(n)$.

Listing 2.13: Nested tuples with different type variables

```
fun tuple a b f = f a b;
fun one = tuple tuple tuple;
fun two = tuple one one;
fun three = tuple two two;
...
```

The trick to induce an exponential running time is demonstrated with the pathological program in Listing 2.13. By allowing `tuple` to be polymorphic and have having two 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 2.12, but has the subtle difference of letting the parameters `a` and `b` (within the type instantiation of the let expression `tuple`)be polymorphic and introducing every "step“ as a polymorphic let expression. The return type of `one` (the type of `f`) is displayed in Equation 2.19.

$$inst(a \to b \to (a \to b \to c) \to c) \to inst(a \to b \to (a \to b \to c) \to c) \to \gamma \to \gamma. \tag{2.19}$$

The first and second instantiations will contain different type variable such that they are not structurally equivalent (Equation 2.20).

$$(\tau \to \mu \to (\tau \to \mu \to \phi) \to \phi) \to (\varphi \to \zeta \to (\varphi \to \zeta \to \delta) \to \delta) \to \gamma \to \gamma. \tag{2.20}$$

□

An interesting observation is that by increasing the amount of polymorphic parameters to some c the number of type variables becomes $\Omega(c^n)$. This observation does not have any significant impact since $O(f(n)) \supseteq \Omega(n^n)$ where f is the algorithm for type inference, such that the problem of type inference in Hindley-Milner is at least in EXPTIME which solvable in both $O(2^n)$ and $O(n^n)$. The upper bound which states that type inference in Hindley-Milner is in fact EXPTIME-complete was justified in [KTU90;

Mai89]. Running the program Listing 2.13 in L yields a blowup of 2^n (Listing 5.1). Figure 2.12 shows the relationship between the program typed in L and the theoretical time of 2^n.

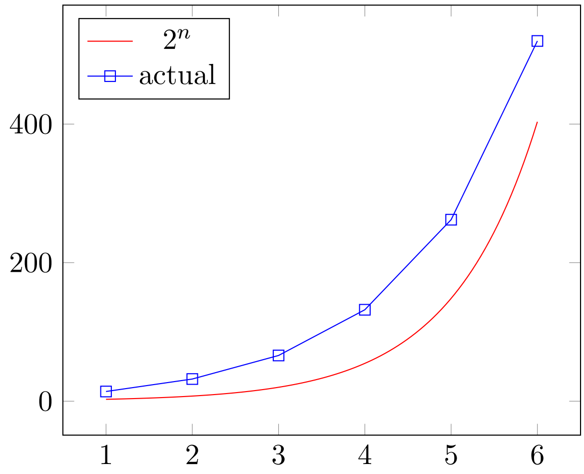


Figure 2.12: Plot of type variables in Hindley-Milner type systems

2.4 Higher level type systems

The Hindley-Milner type system can only express relatively simple programs which robs algorithmic elegance in respect to other type systems. One domain of programs that Hindley-Milner cannot express are those that rely on *rank-n types*. Rank-n types deals with letting abstractions have polymorphic parameters such that a type can be quantified within another type, having its depth bounded by n (rank-**n**). For instance Listing 2.14 is not typable in Hindley-Milner since its type is $\forall\tau.(\forall\gamma.\gamma \rightarrow \texttt{Int}) \rightarrow \tau \rightarrow \texttt{Int}$.

Listing 2.14: Program that requires rank-n types

```
fun f makeNum a =
    ((makeNum a) + (makeNum 0)) + (makeNum (0 == 2))
```

More generally, any type which is quantified on the left side of $\rightarrow$ cannot be moved out thus increases the rank.

Even languages which are typed and inferred by Hindley-Milner like Ocaml have introduced kinds through modules to allow higher-kinded types. Hindley-Milner is in fact a restricted version of another more general type system called *System F* (and System F$\underline{\omega}$). The Hindley-Milner type system introduces abstractions as monomorphic types whereas System F allows any type to be polymorphic. It turns out that allowing higher rank polymorphism makes type inference (type reconstruction in older literature) *undecidable* [Wel99].

Remark 2.4.1. Formal type systems are in their essence deductive systems, which have provable properties such as *decidability*. Decidability in deductive systems is a property which expresses whether a system can be decided by an algorithm (which relates to the encoding of algorithms on theoretic computers). If and only if every valid formula (type) in the deductive system (type system) can have its correctness decided (and reconstructed if necessary) algorithmically.

Another variant of type system is *System F$\underline{\omega}$*. System F$\underline{\omega}$ introduces another feature (System F$\underline{\omega}$ is different to System F, it is not an extension) called type constructors. It is uncommon to use System F$\underline{\omega}$ on its own since it only allows type constructors of monomorphic types (System F introduces polymorphism), which does not yield much expressiveness since only specific types such as `Int` $\rightarrow$ `List Int` would be expressible. Throughout this chapter, type constructors have already been introduced in such a way that they can occur in Hindley-Milner though algebraic data types such as $\forall$`a.a` $\rightarrow$ `List a`. Very commonly, moderately generalized types need both the higher rank polymorphism implied by System F and the type constructors implied by System F$\underline{\omega}$.

Hindley-Milner can only take advantage of System F$\underline{\omega}$ for rank 1 types which significantly constrains the generalization level. A more expressive version of Hindley-Milner is System Fω which in fact, is the basis for the type system of Haskell, which is significantly more expressive than Hindley-Milner.

Remark 2.4.2. Haskell has introduced some additional tweaks to System Fω to avoid the decidability problem among others.

In more expressive functional programming language type systems it has become increasingly popular abstract over implementations by introducing concepts from *category theory*. Naturally many abstractions of category theory require rank-2 polymorphism. More generally the larger the level of polymorphism allowed the larger the possible abstraction level becomes. For instance a general purpose *functor* is implementable and usable with rank-2 polymorphism while a natural transformation becomes a matter of rank-3 polymorphism.

Remark 2.4.3. A functor is a mapping that maps from type constructor instance to another, which for instance can be a functor for lists which provides the algebra $\forall$`a.`$\forall$`b.List a` $\rightarrow$ `List b`.

To generalize functor one must be able to express *kinds* which are the types of type constructors denoted $* \rightarrow *$ for a type constructor that takes some type $*$ and creates some type $*$. $* \rightarrow *$ is a unary type constructor whereas $*$ is an atomic type like `Int` or `List Int`, since these types are fully applied. Kinds allow partial application on type constructors on a general level, since

$$
\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda\Pi\omega \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda\Pi 2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda\Pi\underline{\omega} \arrow[uu] \\
\lambda{\to} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda\Pi \arrow[uu] \arrow[ur] &
\end{tikzcd}
$$

Figure 2.13:

- $\lambda \to$ is the simply typed lambda calculus without polymorphism.
- $\lambda\underline{\omega}$ is System F$\underline{\omega}$.
- $\lambda 2$ is System F.
- $\lambda\omega$ is System Fω.
- Π introduces *dependent types* which is beyond the scope of this thesis.

the only specific constraint is the shape and not where variables appear. The relaxation of type constructions allow various types to be generalized such as `a` $\to$ `b` $\to$ `M a b` which could also have the signature of `a` $\to$ `b` $\to$ `M b a` which kinds abstract over generalizing M to $* \to * \to *$. The kind for `List` is $* \to *$ such that for any τ with kind $* \to *$ the type for functor map is $\forall\tau$`.(`$\forall$`a.`τ`a)` $\to$ `(`$\forall$`b.`τ`b)`.

Remark 2.4.4. Kinds are an abstraction which can exist purely theoretical without robbing the type system of expressiveness. Just as some complications in type systems are resolved with weakening the type system or enriching the syntax, kinds can bu abstracted away into types [WHE13].

λP introduces *Dependent types* which lets types depend on terms in the language. A common example to show what dependent types can do is that of combining two lists v_1 and v_2 of size n_1 and n_2 into a list v_3 of size n_1+n_2, where the size can be expressed in the type system. The signature for such a function in L could be $\forall a$.`List` n_1 `a` $\to$ `List` n_2 `a` $\to$ `List` (n_1+n_2) `a`. Clearly one would have rules for lists such as a base case for the empty list `fun empty = Nil` with type $\forall a$.`List 0 a`, which indicates how the type system is lifted to a logical proofing tool.

Figure 2.13 shows the *lambda cube*, introduced in [Bar91] which encapsulates the family of formal type systems. Complicated type system such as the *calculus of constructions* ($\lambda\Pi\underline{\omega}$) are used in proof assistants since they essentially are deduction systems.

2.5 Concluding remarks

This section should act as an introduction to more general type systems and where Hindley-Milner is placed on the type system map. Hindley-Milner is a small part of a larger more general system which has significant impact on the extensibility of Hindley-Milner. Some very renown functional programming languages began by implementing Hindley-Milner as their type system since it is very fast in practice and relatively simple to implement.

Chapter 3

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.

3.1 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. Such strategies are *call by value*, *call by name* and *call by need*. Call by value is most often the simplest and most natural way of assuming program execution.

Listing 3.1: Program that doubles values

```
fun double x = x + x;
let a = double 10;
double 10;
```

By the call by value semantics, Listing 3.1 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.

Listing 3.2: Implementation of call by name

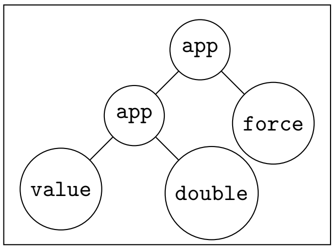
```
fun suspend x unit = x;
fun force x = x 0;
let value = suspend 10;
fun double x =
    fun susExpensiveOp unit =
        (force x) + (force x);
    susExpensiveOp;
let a = double value;
force (double value);
```

The call by name semantics however does only evaluate expressions once they are needed. By the call by name semantics `a` is never evaluated since it is never used. In Listing 3.2 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 3.2 `a` is never forced thus the computation is never performed. The implementation 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 **??**.

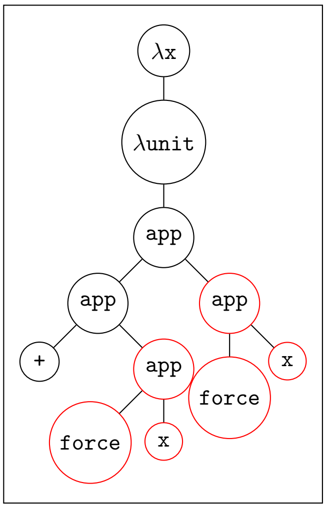
The call by need strategy introduces *lazy evaluation* semantics which is the same as call by name with one extra detail named *sharing*. In Listing 3.2 `force x` is computed twice which may be an expensive operation. Under call by need all results are saved for later use similar to techniques such as dynamic programming. To understand this better observe the expression tree for Listing 3.2 in Figure 3.1. Clearly the two red subtrees in Figure 3.1b 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.

3.2 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 doing so brings upon some problems when implementing an interpreter.



(a) The last expression of the program.



(b) The expression tree for `double`

Figure 3.1

3.3 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 out closures and parameter substitutions in applications never arises.

$$x \tag{3.1}$$

$$F \tag{3.2}$$

$$YE \tag{3.3}$$

There are three types of terms in combinator logic; the variable much like the lambda calculus (Equation 3.1), application (Equation 3.3) and the combinator (Equation 3.2). 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 3.3.1.

$$\begin{aligned} &SKSI \\ =&KI(SI) \\ =&K_I(SI) \\ =&I \end{aligned} \tag{3.4}$$

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 $\lambda x.E$.

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

Case 2: $E = y$ where $y \neq x$ and 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 to $\lambda x.YE'$ 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: $E = \lambda x.E'$ then first rewrite E' with the appropriate cases recursively such that E' becomes either x, y or YE such that Case 1, 2 or 3 can be applied.

The termination of the rewriting to SKI is guaranteed since abstractions are always eliminated and the algorithm never introduce any additional abstractions. When translating the untyped lambda calculus to SKI the "magic" variable names σ, 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 $\sigma \mapsto S, \kappa \mapsto K, \iota \mapsto I$.

3.3.1 Combinator translation growth

Before proving that the SKI translation algorithm produces a program of larger size the notion of size must be established. Size in terms of lambda calculus are the number of lambda terms (Equation 1.2, Equation 1.1 and Equation 1.3) that make up a program. For instance $\lambda x.x$ has a size of two since it is composed of an abstraction and a variable term. The size of an SKI combinator program is in terms of the number of combinators.

Lemma 3.3.1. *There exists a family of lambda calculus programs of size n which are translated into SKI-expressions of size $\Omega(n^2)$.*

Proof.

Case 1: Rewriting $\lambda x.x$ to I is a reduction of one.

Case 2: Rewriting $\lambda x.y$ to Ky is equivalent in terms of size.

Case 3: Rewriting $\lambda x.YE$ to $S(\lambda x.Y)(\lambda x.E)$ is the interesting case. To induce the worst case size Case 1 must be avoided. If $x \notin \mathit{Free}(Y)$ and $x \notin \mathit{Free}(E)$ then for every non-recursive term in Y and E Case 2 is the only applicable rewrite rule which means that an at least equal size is guaranteed. Furthermore observe that by introducing unused parameters one can add one K term to *every* non-recursive case. Observe the instance $\lambda f_1.\lambda f_2.\lambda f_3.(f_1 f_1 f_1)$ where the two unused parameters are used to add K terms to all non-recursive cases in Equation 3.5 such that the amount of extra K terms minus the I becomes variable_references $*$ (unused_abstractions $- 1) = 3 * (3 - 1)$:

$$S(S(KKI)(KKI))(KKI) \tag{3.5}$$

Now let the number of variable references be n and the unused abstractions also be n clearly $\Omega(n * (n-1)) = \Omega(n^2)$

Case 4: Rewriting $\lambda x.E'$ is not a translation rule so the cost is based on what E' becomes.

Notice that the applications $f_1 f_1 \dots f_1$ can in fact be changed to $f_1 f_2 \dots f_n$ since for every f_k where $0 < k \leq n$ there are $n-1$ parameters that induce a K combinator. Let P_n be family of programs with n abstractions and n applications. $\lambda f_1.\lambda f_2.\lambda f_3.(f_1 f_1 f_1) \in P_3$ and in fact for any p where $\forall n \in \mathbb{Z}^+$ and $p \in P_n$, p translates into SKI-expressions of size $\Omega(n^2)$. $\square$

Example 3.3.2. Observe the size of Equation 3.6 in comparison to Equation 5.1.

$$\begin{aligned}
&\lambda f_1.\lambda f_2.f_1 f_2 \\
=&\lambda f_1.\sigma(\lambda f_2.f_1)(\lambda f_2.f_2) \\
=&\lambda f_1.(\sigma(\kappa f_1))(\iota) \\
=&\sigma(\lambda f_1.\sigma(\kappa f_1))(\lambda f_1.\iota) \\
=&\sigma(\sigma(\lambda f_1.\sigma)(\lambda f_1.\kappa f_1))(\kappa\iota) \\
=&\sigma(\sigma(\kappa\sigma)(\sigma(\lambda f_1.\kappa)(\lambda f_1.f_1)))(\kappa\iota) \\
=&\sigma(\sigma(\kappa\sigma)(\sigma(\kappa\kappa)(\iota)))(\kappa\iota) \\
=&S(S(KS)(S(KK)(I)))(KI)
\end{aligned} \tag{3.6}$$

It should become clear that many programs suffer from this consequence such as `let add = (λx.λy.(+ x) y)` $\in P_2$ where the program is written in prefix notation. Translating the lambda calculus into the SKI-expressions does indeed increase the size significantly but does not warrant a write off entirely. More advanced techniques exist to translate the lambda calculus to linearly sized SKI-expressions with the introduction of more complicated combinators [Kis18].

3.4 Reduction strategies

Reductions in the context of the lambda calculus are a small set of well-defined rules for rewriting such that a program is proved or evaluated. The techniques required to correctly prove and evaluate a program are a bit more complicated than the SKI calculus but are rewarding in flexibility and performance. Throughout this section we will explore what difficulties lie within proving and evaluating the untyped lambda calculus via reduction strategies. The first section will interest itself with the semantics of proving the untyped lambda calculus, whilst the second will implement a machine capable of evaluating a result.

3.4.1 Symbols and notation

The following sections will have many variables with different meaning, therefore symbols are constrained to certain types of values as described in Equation 3.7.

$$\begin{aligned} \mathtt{x},\ \mathtt{y},\ \mathtt{z},\ \mathtt{f},\ \mathtt{v},\ \gamma &:= \mathtt{Var} \\ \mathtt{e},\ \mathtt{p},\ \mathtt{l},\ \mathtt{o} &:= \mathtt{Exp} \\ \Gamma,\ \Sigma,\ \Theta,\ \mathtt{S} &:= \mathtt{Heap} \\ \mathtt{E},\ \mathcal{E} &:= \mathtt{Environment} \end{aligned} \tag{3.7}$$

`Exp` is any expression, expressible in both untyped lambda calculus and future extensions.

3.4.2 The abstract evaluation model

The environment is a set of substitutions, that is, a set of variable names to their value denoted $\{\mathtt{x} \mapsto \lambda\mathtt{y}.\mathtt{y}\}$ meaning “the value of variable `x` is λ`y.y`”.

Remark 3.4.1. In the section regarding the semantics of reduction strategies, environments will exist as singleton sets, named *substitutions*.

$$\{\mathtt{x} \mapsto \mathtt{y}\}\mathtt{x} = \mathtt{y} \tag{3.8}$$

Substitutions are performed like shown in Equation 3.9, which states "x is substituted by y".

Evaluation strategies (section 3.1) are a core part of the reduction strategy since the choice of evaluation strategy determines the order in which terms are evaluated. The order of evaluation decides the evaluation strategy and also the final form of expressions [Ses02]. Before delving into more complicated evaluation strategies such as call by need, call by name will be considered.

A reduction strategy would involve substituting variables, once they are applied. When evaluating a term such as $(\lambda x.x)\ y$, x must be substituted by y such that the expression then becomes x with the substitution $\{x \mapsto y\}$ and finally becomes y after the substitution has occurred. The rules in

$$\frac{}{(\lambda x.e) \rightarrow (\lambda x.e)}\ \text{Abs}$$

(a)

$$\frac{}{x \rightarrow x}\ \text{Var}$$

(b)

$$\frac{\{x \mapsto e\}\ p \rightarrow l}{\texttt{let}\ x = e\ \texttt{in}\ p \rightarrow l}\ \text{Let}$$

(c)

$$\frac{l \rightarrow (\lambda x.e) \qquad \{x \mapsto p\}\ e \rightarrow o}{l\ p \rightarrow o}\ \text{App}$$

(d) A simple application rule

Figure 3.2: Simple call by name lambda calculus

Figure 3.2 display a simple set of rules for proving call by name lambda calculus programs.

- The Abs and Var rules (Figure 3.2a and Figure 3.2b) are rules which act as terminal cases of a proof. Abs and Var both state that if either of them occur then the expression must be an axiom by their identity.
- The App rule (Figure 3.2d) states that "$l\ p$ can be proved to evaluate to o if l can be proved to be $(\lambda x.e)$ and e can be proved to evaluate to o, where x has been replaced by p in e".
- The Let rule has the same function as the App rule, but will have an important role in a more refined version of the semantics.

We must introduce rules for how substitutions should act upon encountering lambda calculus terms (Equation 3.9).

$$\begin{aligned}
&\{\texttt{x} \mapsto \texttt{e}\}\texttt{x = e} && (3.9)\\
&\{\texttt{x} \mapsto \texttt{e}\}\texttt{p = p} &&\\
&\texttt{E(l p) = (El)(Ep)} &&\\
&\texttt{E(}\lambda\texttt{x.e) = (}\lambda\texttt{x.Ee)} && (3.10)
\end{aligned}$$

Ambiguous programs

$$\texttt{(}\lambda\texttt{x.(}\lambda\texttt{x.x) 0) 1} \tag{3.11}$$

Proving Equation 3.11 under the rules in Equation 3.9 yields a case for more thorough substitution rules. By inspection one can determine that a simple program like Equation 3.11 yields the symbol `0`, but alas this is not the case. The first step to prove Equation 3.11 is to apply through the App rule, which prompts the application of the Abs rule on the left side for `f`, such that the expression to prove now becomes the lambda abstraction with `x` replaced by `1` (Equation 3.12).

$$\texttt{(}\lambda\texttt{x.1) 0} \tag{3.12}$$

Clearly Equation 3.12 changed the meaning of the program. If we continue the proof which states that the program in Equation 3.12 should evaluate to the symbol `0`, we would not be in luck. Clearly this system is not sound, thus requires some further refinement. Removing the rule Equation 3.10 and adding the two rules in Equation 3.13 solves this type.

$$\begin{aligned}
&\texttt{E(}\lambda\texttt{x.e) = (}\lambda\texttt{x.Ee)} && \texttt{(x} \mapsto \texttt{p)} \notin \texttt{S} && (3.13)\\
&\texttt{E(}\lambda\texttt{x.e) = (}\lambda\texttt{x.(E}\backslash\{\texttt{x} \mapsto \texttt{p}\}\texttt{)e)} && \texttt{(x} \mapsto \texttt{p)} \in \texttt{S} &&
\end{aligned}$$

This is a simple instance of variable ambiguity, a more problematic variant exists which goes by the name of variable capture. This evaluation model is indeed powerful enough to evaluate **most** call by name lambda calculus programs.

Example 3.4.1. With the aforementioned rules, programs can now be proved. Note that the Substitution rule in Figure 3.3 is simply the substitution semantics from Equation 3.13 made clearer. Let the program in Equation 3.14, where `0` is a symbol of any type, be subject to the rules in Figure 3.2, which solves to Figure 3.3.

$$\texttt{((}\lambda\texttt{f.}\lambda\texttt{x.f x) (}\lambda\texttt{x.x)) 0} \tag{3.14}$$

$$\text{Substitution}\ \dfrac{\text{Abs}\ \dfrac{}{\texttt{(}\lambda\texttt{x.(}\lambda\texttt{x.x) x)} \to \texttt{(}\lambda\texttt{x.(}\lambda\texttt{x.x) x)}}}{\texttt{\{f} \mapsto \texttt{(}\lambda\texttt{x.x)\}(}\lambda\texttt{x.f x)} \to \texttt{(}\lambda\texttt{x.(}\lambda\texttt{x.x) x)}}$$

(a)

$$\text{App}\ \dfrac{\text{Abs}\ \dfrac{}{\texttt{(}\lambda\texttt{f.}\lambda\texttt{x.f x)} \to \texttt{(}\lambda\texttt{f.}\lambda\texttt{x.f x)}} \qquad \text{Figure 3.3a}}{\texttt{(}\lambda\texttt{f.}\lambda\texttt{x.f x) (}\lambda\texttt{x.x)} \to \texttt{(}\lambda\texttt{x.(}\lambda\texttt{x.x) x)}}$$

(b)

$$\dfrac{\dfrac{\text{Abs}\ \dfrac{}{\texttt{(}\lambda\texttt{x.x)} \to \texttt{(}\lambda\texttt{x.x)}} \qquad \dfrac{\dfrac{}{\texttt{0} \to \texttt{0}}\ \text{Var}}{\texttt{\{x} \mapsto \texttt{0\}x} \to \texttt{0}}\ \text{Substitution}}{\texttt{(}\lambda\texttt{x.x) 0} \to \texttt{0}}\ \text{App}}{\texttt{\{x} \mapsto \texttt{0\}(}\lambda\texttt{x.x) x} \to \texttt{0}}\ \text{Substitution}$$

(c)

$$\text{App}\ \dfrac{\text{Figure 3.3b} \qquad \text{Figure 3.3c}}{\texttt{((}\lambda\texttt{f.}\lambda\texttt{x.f x) (}\lambda\texttt{x.x)) 0} \to \texttt{0}}$$

Figure 3.3

Variable capture is the basis for some practical difficulties when designing evaluation rules for the untyped lambda calculus. Consider the following sub-program `(λx.y) g` with the following ongoing substitutions $\{\texttt{x} \mapsto \texttt{z}, \texttt{y} \mapsto \texttt{x}, \ldots\}$, which contains `y` as a closure. Substituting by the rules outlined in Equation 3.13 yields `(λx.x) g` which is clearly invalid. The invalid program result is a product of variable capture. To solve ambiguity between variables with the same name, one can perform an *α-conversion*.

Remark 3.4.2. Notice that if variables are renamed before program execution, recursive functions can still suffer from ambiguity since all parameters for that function can occur multiple times.

α-conversions

An α-conversion is a renaming operation which does not modify the meaning of the expression. α-conversions can appear similar to substitutions, for instance renaming `x` to γ appears as $\{\texttt{x} \mapsto \gamma\}$. α-conversions guarantee what is called *α-equivalence* which is the notion of semantic equivalence.

For instance $\lambda\texttt{x.x}$ is α-equivalent with $\lambda\gamma.\gamma$ since both expressions are semantically equivalent. An α-conversion algorithm can be implemented such that when a new variable is introduced through an abstraction, a new name for the variable is given. More formally; Let V_1 be the domain of variables in the program and V_2 be the infinite domain for variable names that satisfies $V_1 \cap V_2 = \emptyset$, such that when a new variable $\texttt{x}$ is discovered, replace it with some $\gamma \in V_2$ and let $V_2 = V_2 \backslash \{\gamma\}$. Working with the previous example $\{\texttt{y} \mapsto \texttt{x}\}(\lambda\texttt{x.y})\ \texttt{g}$ The function `fresh` picks a fresh variable name γ from V_2 and updates V_2 to $V_2 \backslash \{\gamma\}$. α-conversions will be further explored in future refinements of Figure 3.2 in the from of renaming through the Let rule.

The heap

Heaps, like environments in typing (Equation 2.3), define what "state" is required to evaluate some expression. A heap contains mappings from variables to expressions, much like the environment which performs substitutions, except it acts like a store. Heaps are a requirement for call by need semantics. A simple modification to the rules in Figure 3.4, introduces a heap which states that that the semantics must bring a heap along. The

$$\frac{}{\Gamma,\ (\lambda\texttt{x.e}) \rightarrow \Gamma,\ (\lambda\texttt{x.e})}\ \text{Abs}$$

(a)

$$\frac{\Gamma \cup \{\texttt{x} \mapsto \texttt{e}\},\ \texttt{p} \rightarrow \Theta,\ \texttt{l}}{\Gamma,\ \texttt{let x = e in p} \rightarrow \Theta,\ \texttt{l}}\ \text{Let}$$

(b)

$$\frac{\Gamma,\ \texttt{l} \rightarrow \Theta,\ (\lambda\texttt{x.e}) \qquad \Theta,\ \{\texttt{x} \mapsto \texttt{p}\}\texttt{e} \rightarrow \Sigma,\ \texttt{o}}{\Gamma,\ \texttt{l p} \rightarrow \Sigma,\ \texttt{o}}\ \text{App}$$

(c)

$$\frac{\Gamma,\ \texttt{e} \rightarrow \Theta,\ \texttt{p}}{\Gamma \cup \{\texttt{x} \mapsto \texttt{e}\},\ \texttt{x} \rightarrow \Theta \cup \{\texttt{x} \mapsto \texttt{e}\},\ \texttt{p}}\ \text{Var}$$

(d)

Figure 3.4: Call by name lambda calculus with environments

rules in Figure 3.4 are quite different from the rules in Figure 3.2.

- Var is no longer terminal, it now inspects the heap for a replacement value for some x. Notice that Var now removes the mapping from the heap Γ such that recursively defined expressions cannot occur.

- Let now has a role which is distinct from App. Let now introduces values to the heap, but does not induce a substitution.
- App remains the same by eagerly substituting, but now augmented with a heap.
- Abs is now augmented with a heap.

The rules in Figure 3.4 are not any more powerful that the rules in Figure 3.2, but are a basis for lazy evaluation.

Lazy evaluation

With the revised semantics in Figure 3.4, lazy evaluation can now be introduced. The basis for sharing evaluated expressions is rooted in a labelling problem [LI88]. Before delving into a set of rules which use a labelling technique, consider that sharing can be viewed as a dependency graph of expressions. Let Figure 3.5 be a depiction of the dependency graph of Equation 3.15 under the rules in Figure 3.4.

$$\begin{aligned} &\texttt{let k = } (\lambda\texttt{z.z})\ (\lambda\texttt{f.f})\texttt{ in} \\ &\texttt{let x = k in} \\ &\texttt{let y = k in} \\ &\texttt{x + y} \end{aligned} \tag{3.15}$$

A rule which encapsulates "when evaluating a value for a variable, save the

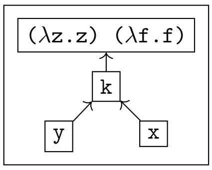


Figure 3.5: Expression dependencies

evaluated value for future use." is required to support sharing computed values. The rule in Figure 3.6 replaces the Var rule, and introduces a sub-

$$\frac{\Gamma,\ \texttt{e} \rightarrow \Theta,\ \texttt{p}}{\Gamma \cup \{\texttt{x} \mapsto \texttt{e}\},\ \texttt{x} \rightarrow \Theta \cup \{\texttt{x} \mapsto \texttt{p}\},\ \texttt{p}}\ \text{Var}$$

Figure 3.6

tle difference; when a variable reference occurs the value which the variable evaluates to is saved as the new reference. Introducing shareable expressions through Let is in it's essence a labelling of an expression. Evaluating Equation 3.15 under the new rules reveals that evaluating `x` forces `k` to be evaluated which then forces `(λz.z) (λf.f)`, which becomes `(λf.f)` and is then saved as the new value of `k` and then as `x`, thus the dependency tree becomes Figure 3.7. One consideration remains, the App rule does not

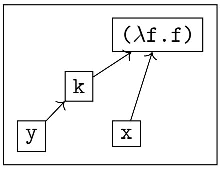


Figure 3.7: Expression dependencies after evaluating `x`

promote lazy evaluation. All non-trivial parameters must be bound to a variable by the Let rule to also allow anonymous expressions to be subject to lazy evaluation. An algorithm for binding anonymous expressions can be found in [Lau93].

Dealing with ambiguity

The rules so far have avoided dealing with variable ambiguity. Notice that ambiguity can only arise in the Let rule, since the Let rule is the only rule of which can introduce new bindings to the heap. Dealing with ambiguity is a matter of ensuring that variables are distinct. Applying the technique from section 3.4.2 properly lets us evaluate programs without ambiguity.

Consider a previous case of variable capture Equation 3.16.

$$\{\mathtt{x} \mapsto \mathtt{z},\ \mathtt{y} \mapsto \mathtt{x},\ \ldots\}(\lambda\mathtt{x}.\mathtt{y})\ \mathtt{g} \tag{3.16}$$

None of the changes so far have any impact on the falsity of the expression. Consider that `x` and `y` must have entered the heap through a Let expression. Consider also that some variable `k` cannot be subject to variable capture if `k` $\notin$ `Bound(λx.e)`. Naturally if `k` is unique, that is, it is introduced through the `fresh` function from section 3.4.2 then `k` can never occur bound. The obvious rule from these considerations must be new Let rule defined in Figure 3.8. The correctness of Figure 3.8 and the aforementioned considerations

$$\frac{\Gamma \cup \{\gamma \mapsto \mathtt{e}\},\ \{\mathtt{x} \mapsto \gamma\}\mathtt{p} \to \Theta,\ \mathtt{l} \qquad \gamma\ \mathtt{=}\ \mathtt{fresh}}{\Gamma,\ \mathtt{let\ x\ =\ e\ in\ p} \to \Theta,\ \mathtt{l}}\ \text{Let}$$

Figure 3.8

are formalised in [Ses97].

Introducing useful functionality

As the set of rules stand currently one can express numbers through church encodings. Church encodings provide a minimal and non-invasive set of combinators which allow the encoding of numbers. Unfortunately it is not as practical as it is minimal to church encode numbers. For instance, to represent the number `100000` one would require `100000` invocations of some successor function. Fortunately dwelling on the representation of numbers is an easy task once one convinces themselves that ordinary numbers and arithmetic operations are friendly.

When discovering an arithmetic operations between two expressions, they must both be forced and then the pending expression must evaluated. Clearly this rule is not encoded into the aforementioned rules, but can be modelled easily as shown in Figure 3.9. Notice that Figure 3.9 also must

$$\frac{\Theta,\ \mathtt{x} \to \Sigma,\ \mathtt{n} \qquad \Sigma,\ \mathtt{y} \to \mathtt{S},\ \mathtt{t} \qquad \oplus \in \{+, -, *, \backslash, =\}}{\Theta,\ \mathtt{x} \oplus \mathtt{y} \to \mathtt{S},\ (\mathtt{n} \oplus \mathtt{t})}\ \text{Bin op}$$

$$\frac{\mathtt{n} \in \mathbb{Z}^{+}}{\mathtt{S},\ \mathtt{n} \to \mathtt{S},\ \mathtt{n}}\ \text{Num}$$

Figure 3.9

accompany a Num rule which introduces integers to the system.

Remark 3.4.3. Notice that the Bin op rule in Figure 3.9 uses `x` and `y` which are in the domain of variables, since all non-trivial expressions must be bound to fresh names through Let.

Example 3.4.2. Now that rules have been established which avoid variable ambiguity through renaming and support lazy evaluation, an example seems natural. An expression which requires the aforementioned properties to resolve as expected is presented in Equation 3.17 and proved in Figure 3.10.

$$\mathtt{let\ y\ =\ (1\ +\ 1)\ in\ (}\lambda\mathtt{x.(}\lambda\mathtt{y.x\ +\ y)\ x)\ y} \tag{3.17}$$

Notice that the left branch and right branch in Figure 3.10c are not identical. The left branch saves the evaluation result such that the right branch only requires a lookup to find γ.

Garbage collection

In functional programming languages unused variables and expressions accumulate during execution. In the context of the rules which have been

$$\text{Lam}\ \frac{}{\{\gamma \mapsto (1 + 1)\},\ (\lambda x.(\lambda y.x + y)\ x) \to \{\gamma \mapsto (1 + 1)\}, (\lambda x.(\lambda y.x + y)\ x)}$$

(a)

$$\text{Lam}\ \frac{}{\{\gamma \mapsto (1 + 1)\},\ (\lambda y.\gamma + y) \to \{\gamma \mapsto (1 + 1)\},\ (\lambda y.\gamma + y)}$$

(b)

$$\text{Bin op}\ \frac{\text{Var}\ \dfrac{\text{Bin op}\ \dfrac{\text{Num}\ \dfrac{}{\{\},\ 1 \to \{\},\ 1} \quad \text{Num}\ \dfrac{}{\{\},\ 1 \to \{\},\ 1}}{\{\},\ 1 + 1 \to \{\},\ 2}}{\{\gamma \mapsto (1 + 1)\},\ \gamma \to \{\gamma \mapsto 2\},\ 2} \qquad \dfrac{\dfrac{}{\{\},\ 2 \to \{\},\ 2}\ \text{Num}}{\{\gamma \mapsto 2\},\ \gamma \to \{\gamma \mapsto 2\},\ 2}\ \text{Var}}{\{\gamma \mapsto (1 + 1)\},\ \gamma + \gamma \to \{\gamma \mapsto 2\},\ 4}$$

(c)

$$\text{Let}\ \frac{\text{App}\ \dfrac{\text{Figure 3.10a} \qquad \text{App}\ \dfrac{\text{Figure 3.10b} \quad \text{Figure 3.10c}}{\{\gamma \mapsto (1 + 1)\},\ (\lambda y.\gamma + y)\ \gamma \to \{\gamma \mapsto 2\},\ 4}}{\{\gamma \mapsto (1 + 1)\},\ (\lambda x.(\lambda y.x + y)\ x)\ \gamma \to \{\gamma \mapsto 2\},\ 4}}{\{\},\ \texttt{let}\ y = (1 + 1)\ \texttt{in}\ (\lambda x.(\lambda y.x + y)\ x)\ y \to \{\gamma \mapsto 2\},\ 4}$$

(d)

Figure 3.10: The proof for the program in Equation 3.17

presented in this section, the heap will inevitably accumulate unused values. It is argued in [Lau93] that garbage collection remains interesting to introduce in the semantics of the system, since it allows reasoning with space usage in an abstract way. In imperative languages which do not make a big deal of side-effects such as C, unused values are managed and released manually. Managing unused variables in a purely lazy functional programming language is not ergonomic, and implies a side-effect, thus the language is no longer pure.

A naive garbage collector could involve letting the Let rule, release references as in Figure 3.11. A garbage collection rule as described in Figure 3.11 works, but is inadequate since it does not allow removal of unused values at *any* time, thus it does not let us reason with recursive programs which run in constant space. Introducing a garbage collection rule which can be placed at any one step of the proof requires some additional work for vari-

$$\frac{\texttt{S} \cup \{\gamma \mapsto \texttt{e}\},\ \{\texttt{x} \mapsto \gamma\}\texttt{y} \rightarrow \Theta,\ \texttt{z} \qquad \gamma\ \texttt{= fresh}}{\texttt{S, let x = e in y} \rightarrow \Theta \setminus \{\gamma \mapsto \texttt{e'}\},\ \texttt{z}}\ \text{Let}$$

Figure 3.11: A Let rule which cleans up after itself

ous reasons. Foremost, introspection of expressions becomes necessary since the rule must determine what gets to stay in the heap. Furthermore, rules which branch such as the App rule and Bin op rule requires the tracing of expressions which are pending. More precisely, when evaluating `l` in the App rule in Figure 3.4c, the expression `p` should not be released since it must be present for the right branch ($\{\texttt{x} \mapsto \texttt{p}\}\texttt{e}$). All branching rules must record expressions which are needed for further branches, this is done by introducing a set of expressions `N`, such that all evaluations are written $\rightarrow_{\texttt{N}}$ (Figure 3.12). In addition to Figure 3.12, there must also be an accompa-

$$\frac{\Gamma,\ \texttt{f} \rightarrow_{(\texttt{N} \cup \{\texttt{z}\})} \Theta,\ (\lambda\texttt{x.e}) \qquad \Theta,\ \{\texttt{x} \mapsto \texttt{z}\}\texttt{e} \rightarrow_{\texttt{N}} \Sigma,\ \texttt{l}}{\Gamma,\ \texttt{f z} \rightarrow_{\texttt{N}} \Sigma,\ \texttt{l}}\ \text{App}$$

$$\frac{\Theta,\ \texttt{x} \rightarrow_{(\texttt{N} \cup \{\texttt{y}\})} \Sigma,\ \texttt{n} \qquad \Sigma,\ \texttt{y} \rightarrow_{\texttt{N}} \texttt{S},\ \texttt{t} \qquad \oplus \in \{+, -, *, \setminus, =\}}{\Theta,\ \texttt{x} \oplus \texttt{y} \rightarrow_{\texttt{N}} \texttt{S},\ (\texttt{n} \oplus \texttt{t})}\ \text{Bin op}$$

Figure 3.12: Branching rules which record needed expressions

nying rule which inspects some current expression `e` and `N`, such that only the `free` variables for these expressions remain (Figure 3.13). `R` is the set of

$$\frac{\Gamma,\ \texttt{e} \rightarrow_{\texttt{N}} \Theta,\ \texttt{p} \qquad \texttt{x} \notin \texttt{R}(\Theta,\ \texttt{N},\ \texttt{e})}{\Gamma \cup \{\texttt{x} \mapsto \texttt{z}\},\ \texttt{e} \rightarrow_{\texttt{N}} \Theta,\ \texttt{p}}\ \text{GC}$$

Figure 3.13: A rule which filters by used values

reachable variables, which inspects the heap, `N` and some current expression `e`. One could also define the garbage collection rule more compactly if the granularity of Figure 3.13 is too fine. Figure 3.14 defines a garbage collection algorithm which prunes all unreachable values, where `Prune` is the minimal set of required values to continue program evaluation. `Prune` can be defined as in Equation 3.18

$$\begin{aligned} &\texttt{Prune}(\Gamma,\ \{\}) = \{\} \\ &\texttt{Prune}(\Gamma,\ \{\texttt{e}\} \cup \texttt{N}) = \{\texttt{x} \mapsto \texttt{y} \mid \texttt{x} \in \texttt{free(e)},\ \texttt{x} \mapsto \texttt{y} \in \Gamma\} \cup \texttt{Prune}(\Gamma,\ \texttt{N}) \end{aligned} \tag{3.18}$$

$$\frac{\Gamma\texttt{, e} \rightarrow_{\texttt{N}} \Theta\texttt{, y} \qquad \Sigma \texttt{ = Prune(}\Gamma\texttt{, N)}}{\Gamma \cup \{\texttt{x} \mapsto \texttt{z}\}\texttt{, e} \rightarrow_{\texttt{N}} \Theta\texttt{, y}}\ \text{GC}$$

Figure 3.14: A rule which prunes unused values

$$\begin{aligned}
\texttt{subst(f, t, }\lambda\texttt{x.e)} &\texttt{ = } \begin{cases} \lambda\texttt{x.e} & \texttt{if x} \equiv \texttt{f} \\ \lambda\texttt{x.subst(f, t, e)} \end{cases} \\
\texttt{subst(f, t, x)} &\texttt{ = } \begin{cases} \texttt{t} & \texttt{if x} \equiv \texttt{f} \\ \texttt{x} \end{cases} \\
\texttt{subst(f, t, x e)} &\texttt{ = subst(f, t, x) subst(f, t, e)} \\
\texttt{subst(f, t, x} \oplus \texttt{e)} &\texttt{ = subst(f, t, x)} \oplus \texttt{subst(f, t, e)}
\end{aligned}$$

Figure 3.15: A function `subst` which states "substitute `f` in with `t` in some expression"

3.4.3 Interpreting programs

Now that the semantics for evaluation of the lambda calculus have been presented, a machine naturally follows. The machine which is presented is originally derived in [Ses97]. One can implement a machine which functions very closely to what the semantics describe. An algorithm which is very alike the natural semantics is presented in Figure 3.16 where the `subst` function is defined as in Figure 3.15. The algorithm in Figure 3.16 is minimal but suffers from some practical issues.

A CPS machine

The algorithm in Figure 3.16 cannot evaluate infinite programs, since it's recursive invocations is not in tail call position. In the machine introduced in [Ses97] (which will be named the *stack machine*), the algorithm is implemented via a stack which is used to record state in recursive invocations that either branch or require sharing. [Ses97] argues that stack testing in the stack machine; testing the top element of the stack to determine the next computation, is a property best eliminated. The *app1* and *app2* rules from the stack machine, displayed in Figure 3.17 respectively, give insight into some properties that we can use to eliminate stack testing. The basis for performing stack testing is the missing information regarding what rule an expression originated from. To eliminate stack testing in the stack machine we can translate the stack machine into a *continuation-passing style*

$$
\begin{aligned}
\texttt{eval}(\Gamma,\ \lambda\texttt{x.e}) &= (\Gamma,\ \lambda\texttt{x.e}) \\[1em]
\texttt{eval}(\Gamma,\ \texttt{f z}) &= \texttt{eval}(\Theta,\ \texttt{subst(x, z, e)}) \\
&\quad \texttt{where } (\Theta,\ \lambda\texttt{x.e}) = \texttt{eval}(\Gamma,\ \texttt{f}) \\[1em]
\texttt{eval}(\Gamma \cup \{\texttt{x} \mapsto \texttt{e}\},\ \texttt{x}) &= (\Theta \cup \{\texttt{x} \mapsto \texttt{y}\},\ \texttt{y}) \\
&\quad \texttt{where } (\Theta,\ \texttt{y}) = \texttt{eval}(\Gamma,\ \texttt{e}) \\[1em]
\texttt{eval}(\Gamma,\ \texttt{let x = e in p}) &= \texttt{eval}(\Gamma \cup \{\gamma \mapsto \texttt{e}\},\ \texttt{l}) \\
&\quad \texttt{where } \gamma = \texttt{fresh}, \\
&\qquad\quad \texttt{l} = \texttt{subst}(\texttt{x},\ \gamma,\ \texttt{p}) \\[1em]
\texttt{eval}(\Gamma,\ \texttt{x} \oplus \texttt{y}) &= (\texttt{S},\ \texttt{n} \oplus \texttt{t}) \\
&\quad \texttt{where } (\Sigma,\ \texttt{n}) = \texttt{eval}(\Gamma,\ \texttt{x}), \\
&\qquad\quad (\texttt{S},\ \texttt{t}) = \texttt{eval}(\Sigma,\ \texttt{y}) \\[1em]
\texttt{eval}(\Gamma,\ \texttt{n} \in \mathbb{Z}^+) &= (\Gamma,\ \texttt{n})
\end{aligned}
$$

Figure 3.16: An algorithm for evaluating the lazy lambda calculus

$$
\begin{aligned}
\texttt{eval}(\Gamma,\ \texttt{S},\ \texttt{l p}) &= \texttt{eval}(\Gamma,\ \texttt{p :S},\ \texttt{l}) \\[1em]
\texttt{eval}(\Gamma,\ \texttt{p :S},\ \lambda\texttt{x.e}) &= \texttt{eval}(\Gamma,\ \texttt{S},\ \texttt{subst(x, p, e)})
\end{aligned}
$$

Figure 3.17: The *app1* and *app2* rules from the stack machine.
`p :S` means that `p` is pushed to the stack `S`, if on the right hand side of `=`, and popped from `S`, if on the left hand side of `=`.

machine, CPS machine for short. A stack will be used for the CPS machine, but the stack will have different role. In the CPS machine the stack holds continuations of the type *cont* : $\Gamma \times S \times \lambda \to \lambda$, that is, there is no returning. There must be catalogue of appropriate continuations for each rule that either branches or requires state and a terminal function `continue` which either continues by popping a continuation from the stack or returns the expression if the stack is empty Figure 3.18. In it's essence, the algorithm in Figure 3.18 evaluates terms in normal order, recording sharing (Var) and branching (App, Bin op), until it reaches a terminal expression. When Figure 3.18 reaches a terminal expression, the most recently pushed continuation must naturally be the expression which is the most recent expression that is subject to the rules Var, App or Bin op.

In the true spirit of suspended computations, a suspended computation should have no impact on the performance characteristics, if not evaluated. The `subst` function in Figure 3.16 is defined to be eager, which naturally does not follow the philosophy of lazy computation.

3.4.4 An invariant on infinite programs

An important problem still remains which is that of infinite programs. Imperative programming languages often solve this by introducing loops, whereas functional programming languages use recursion. Recursion may be equally powerful in terms of expressiveness, but becomes a bit more tricky when considering interpreter details. A prerequisite for an infinitely running program to exist in practice is that the program must not grow it's resource needs as it runs.

The distinction between recursive functions and loops in imperative programming languages is often what makes infinite programs expressible. In a traditional imperative language, a function allocates a *stack frame* and is explicitly parameterized, whereas a loop acts more like an anonymous closure which is always parameterized with itself (a function which is wrapped in a fixed point combinator, like the Y-combinator).

Remark 3.4.4. A call stack is a stack of stack frames. A stack frame is a pointer to a function pointer. Stack frames are used to return execution to the previous function (the calling function). Every time a new function is called, the called-from function places a "resume execution from here" pointer onto the call stack.

Imperative languages are also often evaluated under call by value which further simplifies implementation details. Imperative loops (more interestingly, infinite loops) can safely release all static resources (variables bindings), which were allocated in the iteration, once an iteration has completed. In traditional imperative languages recursive functions can only iterate a finite number of times, more specifically until the call stack is full.

```
        continue(Γ, [], e) = e

   continue(Γ, cont :S, e) = cont(Γ, S, e)

         eval(Γ, S, λx.e) = continue(Γ, S, λx.e)

          eval(Γ, S, l p) = eval(Γ, cont :S, l)
                            where cont(Σ, S', λx.e) =
                                   eval(Σ, S', subst(x, p, e))

  eval(Γ ∪ {x ↦ e}, S, x) = eval(Γ, cont :S, e)
                            where cont(Σ, S', p) =
                                   continue(Σ ∪ {x ↦ p}, S', p)

eval(Γ, S, let x = e in p) = eval(Γ ∪ {γ ↦ e}, S, l)
                            where γ = fresh,
                                  l = subst(x, γ, p)

        eval(Γ, S, x ⊕ y) = eval(Γ, cont :S, x)
                            where cont(Σ, S', n) =
                                   eval(Σ, cont' :S', y)
                               where cont'(Θ, S'', t) =
                                      continue(Θ, S'', n + t)

       eval(Γ, S, n ∈ ℤ⁺) = continue(Γ, S, n)
```

Figure 3.18: A CPS algorithm for a evaluating the lazy lambda calculus

Listing 3.3: Program that implements two functions that fold a `List a` to a `b`

```
type List a =
    | Nil
    | Cons a (List a)
;
fun add a b = a + b;

fun foldl f z l =
    match l
        | Nil -> z;
        | Cons x xs ->
            foldl f (f x z) xs;
    ;

fun foldr f z l =
    match l
        | Nil -> z;
        | Cons x xs ->
            f x (foldr f z xs);
    ;
```

To really understand what happens in a lambda calculus interpreter, we must understand what happens in Listing 3.3. Listing 3.3 implements two variants of a `fold` function which accumulates a list of type `List a` to a `b`. The two variants differ when considering evaluation strategy and *tail call optimization*.

Remark 3.4.5. Tail call optimization is an optimization which can be performed on programs with a particular structure. If the last expression is a function invocation, then the rewritten program does not grow. For instance the expression `let f = (λg.λx.g x) in ... f g' 0` is eventually rewritten to `g' 0`. If for instance the expression awaited a result like in `let f = (λg.λx.x + (g x)) in ... f g' 0`, then it would be rewritten to `x + (g' 0)`, increases the size of the program by `x +`, since the `+` operator requires both expressions to be evaluated. It should become clear that reduction strategies always imply tail call optimization, whenever possible.

The first flavor of `fold`; `foldl`, implements `fold` such that the program expression tree does not grow throughout program interpretation, under a call by value environment. The constraint on evaluation strategy is important for `foldl`, for reasons which will become clear once other evaluation strategies are discussed.

$$\begin{aligned}
&\texttt{foldl 0 add (Cons 1 (Cons 2 }\ldots\texttt{ (Cons n Nil)))} \qquad (3.19)\\
=\ &\texttt{l z (}\lambda\texttt{xs,x.foldl f (f x z) xs)}\\
=\ &\texttt{(Cons 1 (Cons 2 }\ldots\texttt{ (Cons n Nil))) z (}\lambda\texttt{xs,x.foldl f (f x z) xs)}\\
=\ &\texttt{foldl f (f x z) xs } \{\texttt{ xs } \mapsto \texttt{ (Cons 2 (Cons 3 }\ldots\texttt{ (Cons n Nil))), x } \mapsto \texttt{ 1, }\ldots\ \}\\
=\ &\texttt{foldl add (add 1 0) (Cons 2 (Cons 3 }\ldots\texttt{ (Cons n Nil))) }\{\ldots\}\\
=\ &\texttt{foldl add 1 (Cons 2 (Cons 3 }\ldots\texttt{ (Cons n Nil))) }\{\ldots\}\\
&\ldots
\end{aligned}$$

Evaluating `foldl` on a list of size `n` with the addition function showcases how the program only grows by a constant number of terms.

Remark 3.4.6. Note again that the list is always refereed to by reference; the list is not copied.

Part II

Algorithms and Datastructures

Chapter 4

Conventional data structures and terminology

Data structures in traditional contexts are *homogeneous* collections of data, usually with a particular shape represented by an algebraic data type, with an associated set of *morphisms*. A homogeneous collection of data is a collection in which every element is of the same type. Morphisms come in various forms, they essentially encapsulate the operations that can be performed on a data structure (or more generally an object). Algebraic data structures and their associated morphisms come together into an algebra.

Remark 4.0.1. In object oriented programming data structures (an algebra) is most often implemented through a class while functional programming languages often separate the shape and operations.

Conventional data structures encapsulates data structures which are interesting under the call by value (section 3.1) evaluation strategy. Evaluation strategies have many implications on the data structure in question. In call by name or call by need one would have to be careful not to create an unnecessary dependency which may force a computation which could otherwise stay suspended. The choice of evaluation strategy and data structure implementation has a significant impact on complexity analysis, which will be explored.

4.1 Lists and lazy evaluation

An instance of a data structure which has been thoroughly discussed throughout this thesis is `List` (Listing 1.17). `List` is an excellent choice as an introductory data structure since it gives insight into some very universal problems regarding both immutable and mutable data structures. One is free to choose the operations for `List` but a common operation is `map` (Listing 4.1).

Listing 4.1: Mapping from `List a` to `List b`

```
fun map f l =
    match l
     | Cons x xs -> Cons (f x) (map f xs);
     | Nil -> l;
    ;
```

There exists several analytical techniques to justify performance guarantees in call by value data structures, the most straight forward of which is the worst case analysis. Worst case analysis is usually the most straight forward, since it becomes a matter of finding the worst input for any possible state of the data structure.

An interesting observation from map is that it runs differently in a call by need environment compared to a call by value environment. In a call by value environment `map` takes $\Theta(n)$ time since every `Cons`'ed value must be visited. In a call by need environment things become a bit more philosophical. When `map` is evaluated in a call by need environment it is technically suspended thus always requires one operation. When a value which depends on *one* `map` invocation, is forced (from the addition operator for instance), then the computational complexity has the same bounds as if it were call by value. The computational complexity in a call by need (or name) environment for *one* `map` invocation is thus $O(n)$ and $\Omega(1)$, in general *all* call by need algorithms run in $\Omega(1)$. More interestingly, consider n invocations of `map` (Listing 4.2) on some list.

Listing 4.2: n invocation of map

```
fun id x = x;
let xs = Cons 1 (Cons 2 (... (Cons m Nil)));
let m1 = map id xs;
let m2 = map id m1;
...
let m_n = map id m_{n-1};
```

Clearly $\mathtt{m}_n$ in Listing 4.2 requires $n \cdot m$ time if it is forced. Moreover observe that we can "enqueue" an unbounded amount of `map` operations, or rather, the computational complexity is not a function of n (a function of the input size), but rather a function of how much work has been performed on the data structure.

With bounds such as $\Omega(1)$ and a worst case which is unbounded, traditional worst case analysis breaks down.

Bibliography

[Bar91] Henk P Barendregt. “Introduction to generalized type systems”. In: (1991).

[Chu36] Alonzo Church. “An unsolvable problem of elementary number theory”. In: *American journal of mathematics* 58.2 (1936), pp. 345–363.

[Chu85] Alonzo Church. *The calculi of lambda-conversion*. 6. Princeton University Press, 1985.

[Cop97] B Jack Copeland. “The church-turing thesis”. In: (1997).

[Dam84] Luis Damas. “Type assignment in programming languages”. In: (1984).

[DM82] 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*. 1982, pp. 207–212.

[HHS02] BJ Heeren, Jurriaan Hage, and S Doaitse Swierstra. *Generalizing Hindley-Milner type inference algorithms*. 2002.

[How80] William A Howard. “The formulae-as-types notion of construction”. In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[Joh85] Thomas Johnsson. “Lambda lifting: Transforming programs to recursive equations”. In: *Conference on Functional programming languages and computer architecture*. Springer. 1985, pp. 190–203.

[Kis18] Oleg Kiselyov. “λ to SKI, Semantically”. In: *International Symposium on Functional and Logic Programming*. Springer. 2018, pp. 33–50.

[KTU90] Assaf J Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. “ML typability is DEXPTIME-complete”. In: *Colloquium on Trees in Algebra and Programming*. Springer. 1990, pp. 206–220.

[Lau93] John Launchbury. "A natural semantics for lazy evaluation". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1993, pp. 144–154.

[LI88] Jean-Jacques Lévy and AR INRI. "Sharing in the Evaluation of lambda Expressions". In: *Programming of Future Generation Computers II, North Holland* (1988), pp. 183–189.

[Mai89] 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*. 1989, pp. 382–401.

[Sco62] Dana Scott. "A system of functional abstraction, 1968. Lectures delivered at University of California, Berkeley". In: *Cal* 63 (1962), p. 1095.

[Ses02] Peter Sestoft. "Demonstrating lambda calculus reduction". In: *The essence of computation*. Springer, 2002, pp. 420–435.

[Ses97] Peter Sestoft. "Deriving a lazy abstract machine". In: *Journal of Functional Programming* 7.3 (1997), pp. 231–264.

[Wad89] Philip Wadler. "Theorems for free!" In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. 1989, pp. 347–359.

[Wel99] Joe B Wells. "Typability and type checking in System F are equivalent and undecidable". In: *Annals of Pure and Applied Logic* 98.1-3 (1999), pp. 111–156.

[WHE13] Stephanie Weirich, Justin Hsu, and Richard A Eisenberg. "System FC with explicit kind equality". In: *ACM SIGPLAN Notices* 48.9 (2013), pp. 275–286.

Chapter 5

Appendix

Listing 5.1: The output of an exponential type

```
########## tuple ##########
substitution set Map(c0 -> (a0 -> (b0 -> d0)))
type (a0 -> (b0 -> ((a0 -> (b0 -> d0)) -> d0)))
Type vars in sub = 4
########## tuple ##########
########## one ##########
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)) ->
g0))
type (((h0 -> (i0 -> ((h0 -> (i0 -> j0)) -> j0))) -> ((k0 -> (l0 -> ((k0 -> (l0 -> m0)) -> m0))) -> g0)) -> g0)
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)))
Type vars = 14
Type vars in sub = 33
########## one ##########
########## 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 -> (
```

```
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 (((((((((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)
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)),
    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
########## four ##########
########## main ##########
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 -> (
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 = 132
Type vars in sub = 472
########## main ##########
```

$$
\begin{aligned}
&\lambda f_1.\lambda f_2.\lambda f_3.(f_1 f_2) f_3 \qquad (5.1)\\
=&\lambda f_1.\lambda f_2.\sigma(\lambda f_3.f_1 f_2)(\lambda f_3.f_3)\\
=&\lambda f_1.\lambda f_2.\sigma(\sigma(\lambda f_3.f_1)(\lambda f_3.f_2))(\iota)\\
=&\lambda f_1.\lambda f_2.(\sigma(\sigma(\kappa f_1)(\kappa f_2)))\iota\\
=&\lambda f_1.\sigma(\lambda f_2.\sigma(\sigma(\kappa f_1)(\kappa f_2)))(\lambda f_2.\iota)\\
=&\lambda f_1.\sigma(\sigma(\lambda f_2.\sigma)(\lambda f_2.(\sigma(\kappa f_1))(\kappa f_2)))(\lambda f_2.\iota)\\
=&\lambda f_1.\sigma(\sigma(\kappa\sigma)(\sigma(\lambda f_2.\sigma(\kappa f_1))(\lambda f_2.\kappa f_2)))(\lambda f_2.\iota)\\
=&\lambda f_1.\sigma(\sigma(\kappa\sigma)(\sigma(\sigma(\lambda f_2.\sigma)(\lambda f_2.\kappa f_1))(\sigma(\lambda f_2.\kappa)(\lambda f_2.f_2))))(\kappa\iota)\\
=&\lambda f_1.\sigma(\sigma(\kappa\sigma)(\sigma(\sigma(\kappa\sigma)(\sigma(\lambda f_2.\kappa)(\lambda f_2.f_1)))(\sigma(\kappa\kappa)(\iota))))(\kappa\iota)\\
=&\lambda f_1.\sigma(\sigma(\kappa\sigma)(\sigma(\sigma(\kappa\sigma)(\sigma(\kappa\kappa)(\kappa f_1)))(\sigma(\kappa\kappa)(\iota))))(\kappa\iota)\\
=&\lambda f_1.(\sigma(\sigma(\kappa\sigma)(\sigma(\sigma(\kappa\sigma)(\sigma(\kappa\kappa)(\kappa f_1)))(\sigma(\kappa\kappa)(\iota)))))(\kappa\iota)\\
=&\sigma((\lambda f_1.\sigma)(\lambda f_1.(\sigma(\kappa\sigma))(\sigma(\sigma(\kappa\sigma)(\sigma(\kappa\kappa)(\kappa f_1)))(\sigma(\kappa\kappa)(\iota)))))(\lambda f_1.\kappa\iota)\\
=&\sigma((\kappa\sigma)(\sigma(\lambda f_1.\sigma(\kappa\sigma))(\lambda f_1.(\sigma(\sigma(\kappa\sigma)(\sigma(\kappa\kappa)(\kappa f_1))))(\sigma(\kappa\kappa)(\iota)))))(\sigma(\lambda f_1.\kappa)(\lambda f_1.\iota))\\
=&\sigma((\kappa\sigma)(\sigma((\lambda f_1.\sigma)(\lambda f_1.\kappa\sigma))(\sigma(\lambda f_1.\sigma(\sigma(\kappa\sigma)(\sigma(\kappa\kappa)(\kappa f_1))))(\lambda f_1.(\sigma(\kappa\kappa))(\iota)))))(\sigma(\kappa\kappa)(\kappa\iota))\\
=&\sigma\\
&((\kappa\sigma)(\sigma((\kappa\sigma)(\sigma(\lambda f_1.\kappa)(\lambda f_1.\sigma)))(\sigma(\sigma(\lambda f_1.\sigma)(\lambda f_1.(\sigma(\kappa\sigma))(\sigma(\kappa\kappa)(\kappa f_1))))(\sigma(\lambda f_1.\sigma(\kappa\kappa))(\lambda f_1.\iota)))))\\
&(\sigma(\kappa\kappa)(\kappa\iota))\\
=&\sigma\\
&((\kappa\sigma)(\sigma((\kappa\sigma)(\sigma(\kappa\kappa)(\kappa\sigma)))(\sigma(\sigma(\kappa\sigma)(\sigma(\lambda f_1.\sigma(\kappa\sigma))(\lambda f_1.(\sigma(\kappa\kappa))(\kappa f_1))))(\sigma(\sigma(\kappa\sigma)((\kappa\kappa)(\kappa\kappa)))(\kappa\iota)))))\\
&(\sigma(\kappa\kappa)(\kappa\iota))\\
=&\sigma\\
&((\kappa\sigma)(\sigma((\kappa\sigma)(\sigma(\kappa\kappa)(\kappa\sigma)))(\sigma(\sigma(\kappa\sigma)(\sigma(\sigma(\lambda f_1.\sigma)(\lambda f_1.\kappa\sigma))\\
&(\sigma(\lambda f_1.\sigma(\kappa\kappa))(\lambda f_1.\kappa f_1))))(\sigma(\sigma(\kappa\sigma)((\kappa\kappa)(\kappa\kappa)))(\kappa\iota)))))\\
&(\sigma(\kappa\kappa)(\kappa\iota))\\
=&\sigma\\
&((\kappa\sigma)(\sigma((\kappa\sigma)(\sigma(\kappa\kappa)(\kappa\sigma)))(\sigma(\sigma(\kappa\sigma)(\sigma(\sigma(\kappa\sigma)((\kappa\kappa)(\kappa\sigma)))\\
&(\sigma(\sigma(\lambda f_1.\sigma)(\lambda f_1.\kappa\kappa))(\sigma(\lambda f_1.\kappa)(\lambda f_1.f_1)))))(\sigma(\sigma(\kappa\sigma)((\kappa\kappa)(\kappa\kappa)))(\kappa\iota)))))\\
&(\sigma(\kappa\kappa)(\kappa\iota))\\
=&\sigma\\
&((\kappa\sigma)(\sigma((\kappa\sigma)(\sigma(\kappa\kappa)(\kappa\sigma)))(\sigma(\sigma(\kappa\sigma)(\sigma(\sigma(\kappa\sigma)((\kappa\kappa)(\kappa\sigma)))\\
&(\sigma(\sigma(\kappa\sigma)((\kappa\kappa)(\kappa\kappa)))(\sigma(\kappa\kappa)(\iota)))))(\sigma(\sigma(\kappa\sigma)((\kappa\kappa)(\kappa\kappa)))(\kappa\iota)))))\\
&(\sigma(\kappa\kappa)(\kappa\iota))\\
=&S\\
&((KS)(S((KS)(S(KK)(KS)))(S(S(KS)(S(S(KS)((KK)(KS)))\\
&(S(S(KS)((KK)(KK)))(S(KK)(I)))))(S(S(KS)((KK)(KK)))(KI)))))\\
&(S(KK)(KI))
\end{aligned}
$$

(5.2)