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

Abstract

When a programmer picks a language, the choices often falls into two schools, the imperative one and the functional one. The languages that fall into the category of imperative languages often deal with details of how to perform computation. However, functional programming languages deal with problems at higher abstraction levels, that are based in theoretic computer science. Functional programming languages have the benefit of being built on rigorous foundations, some of which guarantee correctness.

In this work, we will demystify the workings of functional programming languages, beginning at definition and ending at evaluation. We will explore the relationship between high level functional programming languages and the minimal lambda calculus. Thereafter, we will consider systems of verification to prove the correctness of programs through type inference and type checking. Once a program has been verified, efficient and minimal methods of practical evaluation of the language is considered, under various evaluation strategies.

Contents

Chapter 1

Introduction and preliminaries

At the time that this thesis was created, imperative programming languages are by far the most dominant in development of software. Imperative languages are the natural materialization of how we act, since they in their essence are encoded in a way similar to how we behave.

1.1 A case for immutability

Making a sandwich, for instance, involves following a recipe, or in computational terms, an algorithm. An imperative algorithm for Figure 1.1 is very much alike the recipe in that, it defines the steps required to make a sandwich. In a traditional imperative programming language, such as C or Java, the sandwich algorithm would involve changing or *mutating* the sandwich at every step. Mutation is the natural way of applying changes to an object, which is where functional programming languages very often differ.

Remark 1.1.1. When functional programming languages are mentioned, it should now be assumed that they are immutable.

In immutable functional programming languages, one cannot mutate an object, but rather create a new one with the desired changes. The naive intuition of this change might indicate that immutability and mutability are opposites, which is necessarily not the case. Let `heatBread` be a function that creates a heated slice of bread and let `butter` be a function that creates a copy of a heated slice of bread, and returns it buttered. The abstraction of immutability might not seem unnatural, if we imagine that every step is performed at a particular point in time. Immutability introduces the abstraction of travelling in time, by never changing objects. In this context, mutability implies the ability to travel back in time and change an action. If one performs a seres of immutable steps in the sandwich algorithm and

1. Heat two slices of bread.
2. Butter the heated bread.
3. Boil eggs.
4. Slice tomatoes.
5. Dice cucumber.
6. Slice eggs.
7. Place tomatoes on the bottom slice of the buttered bread bread.
8. Place the diced cucumber on top of the tomatoes.
9. Place sliced eggs on top of the diced cucumber.
10. Place the top slice of the buttered bread on the eggs.

Figure 1.1: A recipe for sandwiches

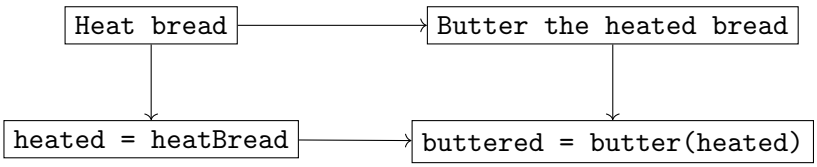


Figure 1.2: An abstract and concrete representation of bread

fails at say, item 4, then one can simply travel back in time and begin from the previous step.

This is the primary philosophy behind immutability, and has other practical benefits that from from the guarantee of immutable data. For instance, many details of concurrency are eliminated in the presence of immutability, since one cannot "go back and change data".

Throughout this work, all algorithms and data structures will be used presented and used immutably.

1.2 Purity of programs

A programming language where all variables can be replaced by their expressions is called *pure*. Purity implies that there cannot be state, mutability and side effect, since all of these concepts can violate the deterministic nature of purity. Purity is a powerful property, since it allows us to reason about our whole program at once. In fact, if a program is pure, any number

of substitutions and rewriting can be performed on the program without altering the meaning, which is not the case when side effects are considered. The concept of purity becomes very helpful once we begin to consider evaluation.

1.3 Preliminaries

1.3.1 Notation

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 1.1.

$$\begin{aligned} \texttt{card}(\emptyset) &= 0 \\ \texttt{card}(\{\texttt{x}\} \cup \texttt{S}) &= 1 + \texttt{card}(\texttt{S}) \end{aligned} \tag{1.1}$$

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 1.2.

$$\begin{aligned} &\texttt{fib(0)} = 0 \\ &\texttt{fib(1)} = 1 \\ &\texttt{fib(n)} = \texttt{F}_{\texttt{n-1}} + \texttt{F}_{\texttt{n-2}} \\ &\quad \texttt{where } \texttt{F}_{\texttt{n-1}} = \texttt{fib(n - 1)}, \\ &\qquad\quad \texttt{F}_{\texttt{n-2}} = \texttt{fib(n - 2)} \end{aligned} \tag{1.2}$$

Functions can also contain nested functions such as in Equation 1.3.

$$\begin{aligned} \texttt{double(n)} &= \texttt{k} + \texttt{id(n)} \\ &\quad \texttt{where id(x)} = \texttt{x}, \\ &\qquad\quad \texttt{k} = \texttt{id(n)} \end{aligned} \tag{1.3}$$

Finally, conditional cases may also occur in functions such that the choice of action is dependent on a predicate. The function in Equation 1.4 increments `n` in the case that it is even, else it returns `n`.

$$\texttt{makeOdd(n)} = \begin{cases} \texttt{n + 1} & \texttt{if n mod 2 = 0} \\ \texttt{n} \end{cases} \tag{1.4}$$

1.3.2 Running and sources

The compiler and interpreter for the programming language implemented in this thesis was written in Scala. To compile the compiler and interpreter, one must install sbt (simple build tool) and bloop (build server). To run a compile and run a program the program bloop must be invoked with the arguments `bloop run root -- /path/to/code/src/L/fib`, where the absolute path is the path to the file containing the code for the implemented programming language. Every resource required for this thesis can be found in the git repository at `https://github.com/valdemargr/masters`, the code can be found in the directory `src`.

Chapter 2

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 thesis, 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 referred to as L.

2.1 The 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.

$$x \tag{2.1}$$
$$\lambda x.E \tag{2.2}$$
$$YE \tag{2.3}$$

Equation 2.2 displays a lambda *abstraction* which 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 term E in the lambda abstraction will be called the *body*. The abstraction will also be called a *function* is some contexts since it is the encoding of a mathematical function. The *variable* (Equation 2.1) is a reference to some value introduced by an abstraction (or to be introduced; let expression). 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 perceived. Context is important when discussing *free* and *bound* variables as whether a variable is free or bound is decided by the context. Free variables are determined by Equation 2.4, Equation 2.5, and Equation 2.6.

$$\mathit{free}(x) = \{x\} \tag{2.4}$$
$$\mathit{free}(\lambda x.E) = \mathit{free}(E) \backslash \{x\} \tag{2.5}$$
$$\mathit{free}(YE) = \mathit{free}(Y) \cup \mathit{free}(E) \tag{2.6}$$

Example 2.1.1.

$$\lambda x.\lambda y.x \tag{2.7}$$

In Equation 2.7 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 2.3 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*, also named the *let expression*, to the untyped lambda calculus, which does not make the untyped lambda calculus any more powerful, but it allows defining different behaviors for introducing bindings through either abstraction or let binding in typing and evaluation. The let binding is written `let` x `= Y in E` where `Y` and `E` are arbitrary lambda calculus terms. When evaluated, the term `E` should have every instance of x replaced by `Y` such that `let` x `= Y in E`

$\equiv \{x \mapsto$ `Y`$\}$`E`. When the let binding does not introduce unique semantics, it can simply be expressed as abstraction and application `let` x `= Y in E` $\equiv$ `(`λ`x.E) Y`.

Example 2.1.2. Let Y be $\lambda x.T$ and E be z then YE is $(\lambda x.T)z$. Furthermore substituting x for E such that Y becomes $\{x \mapsto E\}T$. Since $E = z$ then substitute E for z such that $\{x \mapsto z\}T$ read as "Every instance of x in T should be substituted by z".

Remark 2.1.1. Substituting lambda terms is a popular method of evaluating lambda calculus programs. Other equally small and powerful representations of programming languages, equivalent to the lambda calculus exist and are used in programming languages such as Miranda and Clean, which implement *combinator graph rewriting*, and will be introduced in section 4.3.

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 a fixed-point combinator like 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 mathematical notation, and will be implemented so. The simplicity of lambda calculus makes it an excellent language to transpile to which is a common technique, and is also what will be employed here.

2.2 The high-level language

The programming language L is similar to the lambda calculus, but introduces some additional features. L introduces the keyword `fun` for introducing functions and `let` for introducing program variables. Expressions in L such as `let` are terminated with the symbol`;`.

Moreover, the L language also introduces numerics and arithmetic operations. Numerics are expressions written as constants such as `1` and `42`. Arithmetic operations exist as infix binary operators between expressions such as `x + 2`.

`fun` defines a name for the function and the parameters such as `fun id x = x;`, which translates into `let` id $= (\lambda x.x)$ `in` E. The special function `main` is a function with no parameters that will contain the first expression for the program.

`let` can be defined anywhere and becomes a program variable introduced through abstraction such that `fun add x y = let a = x + y; a`

translates into `let` $id = (\lambda x.\lambda y.(\lambda a.a)(x + y))$ `in` .... It may seem a bit strange that `fun` becomes a let binding and `let` becomes an abstraction, but functions must be introduced as polymorphic and program variables must be introduced as monomorphic, which are concepts that will be introduced in typing.

Furthermore, the language L allows algebraic data structures with type constructors. The algebraic data structure with a arity one type constructor named `List` is written as `type List a = | Nil | Cons a (List a);`. When algebraic data structures occur, one must also be able to match on the case, which in the instance a sum of `List` is written `fun sum l = match l | Nil -> 0; | Cons x xs -> (x + (sum xs));;`. Algebraic data structures and type constructors will be introduced and explored more throughly throughout this work.

Boolean expressions can be introduced through algebraic data structures, but will exist as natural constructs of the language, such as $x == y$, since binary comparison operators and conditionals are more ergonomic. Naturally, a conditional function `if else` which takes the form of a condition, a case for the instance of truth and a case for the instance of false, written `if` (Y) E`; else` T`;`

2.3 A recipe for transpilation

High level languages associated with lambda calculus are often also close to it. The L language requires a minimal amount of rewriting to be translated into the untyped lambda calculus. Though close, the process of transpilation is not trivial. The lambda calculus must have expressions that follow let bindings, which exist as functions denoted `fun` in L. In L the body of the `main` function is the last expression in a program such that the program in Listing 2.1 becomes Equation 2.8.

Listing 2.1: Add function in L

```
1 fun add a b = a + b;
2 fun main = (add 2 4);
```

$$\texttt{let } add = \lambda a.\lambda b.a + b \texttt{ in } (add\ 2\ 4) \tag{2.8}$$

2.3.1 Scoping

Notice that Equation 2.8 must bind the function name *add* "outside the rest of the program" or more formally in an outer scope. Notice in Listing 2.2 that several problems occur, such as the order that functions are defined, may alter whether the program is correct. For instance the program defined

Listing 2.2: An order dependent program

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

in Listing 2.2 would not translate into valid lambda calculus, it would translate into Equation 2.9. The definition of `sub` is missing a reference to the `add` function.

$$\begin{aligned} &\texttt{let } sub \ = \lambda a.\lambda b.add\ a\ (0-b) \ \texttt{in} \\ &\texttt{let } add \ = \lambda a.\lambda b.a+b \ \texttt{in} \\ &sub\ (add\ 10\ 20)\ 5 \end{aligned} \tag{2.9}$$

lambda lifting is a technique where free variables (section 2.1) are explicitly parameterized [Joh85]. This what is required in Equation 2.9, which has the lambda lifted solution seen in Equation 2.10.

$$\begin{aligned} &\texttt{let } sub \ = \lambda add.\lambda a.\lambda b.add\ a\ (0-b) \ \texttt{in} \\ &\texttt{let } add \ = \lambda a.\lambda b.a+b \ \texttt{in} \\ &sub\ add\ (add\ 10\ 20)\ 5 \end{aligned} \tag{2.10}$$

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

Moreover lambda lifting must also conform to "traditional" scoping rules. *Variable shadowing* occurs when there exists more than one reachable variables of the same name. In the lambda calculus, the "nearest" in regard to scope distance is chosen, which is the desired semantics. Effectively other variables than the one chosen are *shadowed*. For instance, the function `f` in Listing 2.3 yields `12`.

Listing 2.3: Scoping rules in programming languages

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

2.3.2 Recursion

Reductions in mathematics and computer science are one of the principal methods for development of solutions. Listing 2.4 defines a function `f` that in fact is infinite. In the untyped lambda calculus there are not any of

Listing 2.4: Infinite program

```
1 fun f n =
2   if (n == 0) n;
3   else
4     if (n == 1) n + (n - 1);
5     else
6       if (n == 2) n + ((n - 1) + (n - 2));
7       ...
```

Listing 2.5: Recursive program

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

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 2.5 defines a recursive variant of `f` it is a product of the reduction in Equation 2.11.

$$n + (n-1) \cdots + 0 = \sum_{k=0}^{n} k \tag{2.11}$$

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).

Now that the case for recursive functions has been introduced, we will interest ourselves with simple, albeit rather unresourceful, recursive lambda calculus programs, such as Equation 2.12.

$$\texttt{let } f = \lambda x.fx \texttt{ in } E \tag{2.12}$$

For Equation 2.12 to be in a valid lambda calculus form, f must be reachable in $\lambda x.fx$. A naive attempt of solving this could involve substituting f by it's implementation (Equation 2.13), which yields another issue.

$$\texttt{let } f = \lambda x.(\lambda x.fx)x \texttt{ in } E \tag{2.13}$$

The program in Equation 2.13 still suffers from the problem in Equation 2.12, now at one level deeper.

One could say that the problem is now recursive. Recall that lambda lifting (subsection 2.3.1) is the technique of explicitly parameterizing outside

references. Assuming that f lives in the scope above it's body lets us to lambda lift f into it's own body by explicitly parameterizing f, such that the program in Equation 2.12 is reshaped to the program in Equation 2.14.

$$\texttt{let } f = \lambda f.\lambda x.fffx \texttt{ in } E \tag{2.14}$$

The invocation of f must involve f such that it becomes ffn. The *Y-combinator*; an implementation of a fixed-point combinator, displayed in Equation 2.15, is the key to realize that the lambda calculus can implement recursion. Languages with functions and support binding functions to parameters can implement recursion with the Y-combinator.

$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \tag{2.15}$$

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

$$\begin{aligned}&\texttt{let } g = \lambda x.fx \texttt{ in} \\ &\texttt{let } f = \lambda x.gx \texttt{ in } \ldots\end{aligned} \tag{2.16}$$

Notice that in Equation 2.16 that g requires f to be lifted and f requires g to be lifted.

$$\begin{aligned}&\texttt{let } g = \lambda f.\lambda g.(\lambda x.ffgx) \texttt{ in} \\ &\texttt{let } f = \lambda f.\lambda g.(\lambda x.gfgx) \texttt{ in } \ldots\end{aligned} \tag{2.17}$$

If a transpilation algorithm "pessimistically" lambda lifts all definitions from the above scope, then all required references are bound thus the program becomes valid.

Notice that in Equation 2.14 and Equation 2.17, the invocation of the (mutually-)recursive function requires the explicit parameterization of itself and all other functions. Figuring out what lifted parameters a function requires, requires much program analysis. A method which can be used to hide lambda lifted parameters of abstractions, called *partial application*, is very useful. A partial application is a technique that encapsulates only delivering a subset of all the required parameters to an abstraction, such as all the lambda lifted parameters.

$$\begin{aligned}&\texttt{let } g' = \lambda f.\lambda g.(\lambda x.fffgx) \texttt{ in} \\ &\texttt{let } f' = \lambda f.\lambda g.(\lambda x.gfgx) \texttt{ in} \\ &\texttt{let } g = g'f'g' \texttt{ in} \\ &\texttt{let } f = f'f'g' \texttt{ in } \ldots\end{aligned} \tag{2.18}$$

Equation 2.18 solves the problem of referring to f and g outside of their definitions. Unfortunately Equation 2.18 still requires analysis of f and g

inside the bodies of g' and f', such that the references are replaced. Fortunately the lifted functions g and f can be bound to their partially applied variants, inside of their bodies, such as in Equation 2.19.

$$\begin{aligned}
&\texttt{let } g\text{'} = \lambda f''.\lambda g''.(\texttt{let } f = f''f''g'' \texttt{ in } \lambda x.fx) \texttt{ in} \\
&\texttt{let } f\text{'} = \lambda f''.\lambda g''.(\texttt{let } g = g''f''g'' \texttt{ in } \lambda x.gx) \texttt{ in} \\
&\texttt{let } g = g'f'g' \texttt{ in} \\
&\texttt{let } f = f'f'g' \texttt{ in } \ldots
\end{aligned} \tag{2.19}$$

Now we have developed a technique which implements (mutual) recursion, and requires no rewriting other than binding names and lifting parameters.

Another method of allowing recursive let bindings can be realized in the evaluation process, thus this leaves the lambda calculus at the mercy of interpreter. The evaluation method explored in section 4.4 introduces let bindings into their own bodies.

Remark 2.3.1. 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".

2.4 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.

2.4.1 Algebraic data types

Algebraic data types, in their essence, are tagged unions of tuples. Algebraic data types become much more powerful once enhanced with *type constructors*, as such, they are closely related to types thus require some type theory to fully grasp. Types are explored more in depth in section 3.1.

An algebraic data type is a name A for a tagged union of tuples $T_1, T_2 \ldots, T_n$ that states that any tuple T_k with tag t_k for some $1 \leq k \leq n$ is of type A. This construct allows any of the tuples $T_1, T_2 \ldots, T_n$ to be unified and inhabit A under the names $t_1, t_2 \ldots, t_n$.

In L, algebraic data types give rise to tangible implementations of abstract types such as lists (Listing 2.6). Listing 2.6 displays an algebraic data type with name `IntList` which is the tagged union of the nullary tuple `Nil` and the binary tuple `Cons`. `IntList` states that if a value that inhabits the

Listing 2.6: List algebraic data type

```
type IntList =
    | Nil
    | Cons Int IntList
;
```

Listing 2.7: List instance and match

```
fun main =
  let l = Cons 1 (Cons 2 (Cons 3 Nil));
  match l
      | Nil -> 0;
      | Cons x xs -> x;
  ;
```

type `IntList` occurs, it must either be the tag `Nil` or the tag `Cons` which carries a value of type `Int` and another value of type `IntList`. Once a tuple of values are embedded by a tag 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 matching comes in many forms, notably it may allow one to define a computation based on the type an algebraic data type instance realizes (Listing 2.7), which is the type of pattern matching of L.

Scott encoding

Pattern matching strays far from the simple untyped lambda calculus, but can in fact be encoded into it. *scott encoding* (Equation 2.20) is a technique that describes a general purpose framework to encode algebraic data types into the lambda calculus [Sco62]. Considering an algebraic data type instance as a function which accepts a set of "handlers", paves the way for the encoding into the lambda calculus. The scott encoding specifies that constructors should now be functions that are each parameterized by the constructor parameters $x_1 \dots x_{A_i}$ where A_i is the arity of the constructor i. Additionally, each of the constructor functions return a n arity function, where n is the number of tagged tuples $T_1, T_2 \dots, T_n$. Of the n functions, the constructor parameters $x_1 \dots x_{A_i}$ are applied to the i'th "handler" c_i. These encoding rules ensure that the "handler" functions are provided uniformly to all instances of the algebraic data type.

$$\lambda x_1 \dots x_{A_i}.\lambda c_1 \dots c_n.c_i x_1 \dots x_{A_i} \tag{2.20}$$

Listing 2.8: List algebraic data type implementation

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

Example 2.4.1. The `IntList` algebraic data type in Listing 2.6 has two constructors, the `Nil` constructor and the `Cons` constructor. The construction of a value of type `Cons` or `Nil` effectively partially applies an abstraction and returns an abstraction that is uniform for both `Nil` and `Cons`, such as in (Listing 2.8).

Types have not been introduced yet, but seeing the types of these functions might help understanding scott encoding. Equation 2.21 is the constructor type for `Nil` and Equation 2.22 is the constructor type for `Cons`.

$$b \to (a \to \texttt{List}\ a \to b) \to b \tag{2.21}$$

$$(a \to \texttt{List}\ a \to b) \to b \to (a \to \texttt{List}\ a \to b) \to b \tag{2.22}$$

Encoding the constructors in L yields the functions defined in Listing 2.8. Pattern matching is but a matter of applying the appropriate handlers. In Listing 2.9.

Listing 2.9: Example of scott encoded list algebraic data type

```
1 fun main =
2   let l = cons 1 (cons 2 (cons 3 nil));
3   fun consCase x xs = x;
4   fun nilCase = 0;
5   l consCase nilCase
```

2.4.2 Remark on scott encoded algebraic data types

Scott encoding provides a method of encoding algebraic data types into the lambda calculus without expanding the language of lambda calculus. Scott encoding has a big drawback, once one considers expanding pattern matching. Nested pattern matching is a commonly used feature, in intricate algorithms (for instance in [Oka99]). If one was to transform pattern nested pattern matching for Listing 2.10, one would have to consider the implications. `l` requires a handler for `Cons` and `Nil`, so what are the correct handlers? If one were to write it out in handle form, the program would

Listing 2.10: Nested pattern match

```
1 fun sum2 l =
2   match l
3     | Cons x (Cons y ys) -> x + y;
4     | _ -> 0;
5   ;
```

Listing 2.11: Nested pattern match in handle form

```
1 fun sum2 l =
2   fun onNil = 0;
3   fun onConsfirst x xs =
4     fun onConssecond y ys =
5       x + y;
6     xs onConssecond onNil;
7   l onConsfirst onNil;
```

come out as in Listing 2.11. One would have to perform quite a bit of non-trivial introspection to figure out what handler to use.

Chapter 3

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. The spell checking equivalent for computer programs could be type checking; a sub-problem 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]. In fact, type systems are children of findings in the field of logic [Wad15].

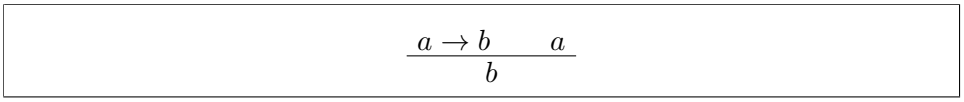


Figure 3.1

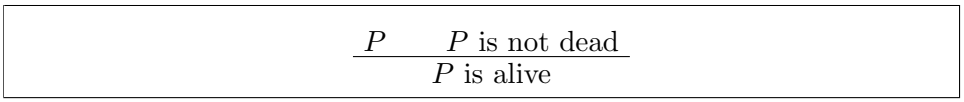


Figure 3.2

3.1 Types and validation

A type system and algorithm which shortly will become the main interest is the Hindley-Milner type system and the Damas-Milner Algorithm W. The Hindley-Milner type system are the natural semantics which the Damas-Milner Algoritm W (or any other inference algorithm that infers types that the Hindley-Milner type system accepts) must adhere to. The Damas-Milner Algorithm W is a child of an observation, that one can traverse the typing rules of Hindley-Milner backwards. The Damas-Milner Algorithm W reconstructs types by first introducing types as unknowns, and then constraining these unknown types as more information is gathered by an operation named *unification*.

3.1.1 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 $a \to b$ and a are the premises and b is the conclusion (Figure 3.1). Conditions can also occur rules which state what conditions must be met for the rule to apply (Figure 3.2). Furthermore, hypotheses are written $\Gamma \vdash p$ which states that under the assumption of Γ then p.

3.1.2 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

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

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

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

- 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 3.3: A simple set of rules for the simply typed lambda calculus

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 3.3).

Example 3.1.1. 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 3.4.

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

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

Figure 3.4: The proof for $\lambda f.\lambda x.fx$

`let` $id = \lambda x.x$ `in` ... 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, since it introduces polymorphism like System F, but in a restricted way such that it becomes decidable.

3.1.3 Polymorphism and Hindley-Milner

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 3.1).

$$mono\ \tau = a \mid \tau \to \tau \mid C\tau_1 \ldots \tau_n \tag{3.1}$$

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 type.

Example 3.1.2. Let $\texttt{ADT} = \{\texttt{List}\}$ where `List` is defined as in Listing 2.7. The *type constructor* (not to be confused for constructors like `Cons` or `Nil`) for `List` has the signature `a` $\to$ `List a` stating that if supplied with some

type `a` it constructs a type of `List a` (effectively containing the provided type). The type `List` is a type constructor with one type parameter `a`.

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

A polytype is a polymorphic type (Equation 3.2).

$$poly\ \sigma = \tau \mid \forall a.\sigma \tag{3.2}$$

Polymorphic types either take the shape of a type variable or universally quantify some type, naming the quantifier a. All type variables of σ, are not necessarily quantified, since the constraint imposed by the **Gen** rule of Figure 3.6 constrains the domain that a ranges over to contain only type variables that are not free in Γ. The notion of free type variables and polymorphism will be explored further once the definition of free has been introduced.

The type hierarchy of the Hindley-Milner type-system is shown in Figure 3.5. Notice the lack of σ in Figure 3.5 since σ is but a mechanism to

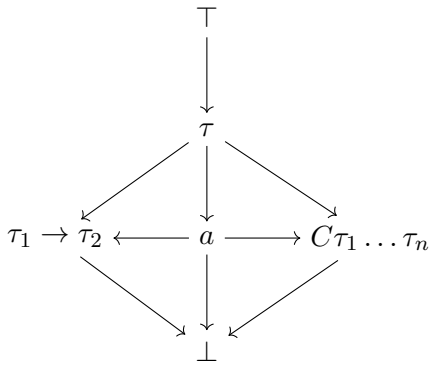


Figure 3.5: The type hierarchy of Hindley-Milner.

prove type systems.

A principal component of typing in Hindley-Milner is the *environment*, the environment is nothing more than a more context aware name for the hypotheses. The environment Γ is now a set of pairs of program variable and polytype (Equation 3.3), in contrast to a set of pairs of program variables and monotypes (Figure 3.4). Now $\vdash$ is enhanced such that it can also judge

polymorphic types; $\Gamma \vdash x : \sigma$ signifies a *typing judgment*, meaning that under the assumption of Γ, the variable x can take the **poly**type σ.

Remark 3.1.1. Notice that judging a type now, 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 3.2.1 where `id` may take the type of either $\forall a.a \to a$, `Int` $\to$ `Int` or $\forall a.(a \to a) \to (a \to a)$.

$$\Gamma \ = \epsilon \,|\, \Gamma, x : \sigma \tag{3.3}$$

Like in the untyped lambda calculus, types also have notions of free and bound type variables. Type variables occur bound when they occur in the environment (from being bound by either a let expression or abstraction) or when they occur quantified. Type variables occur free when they are not introduced by a quantification and they do not occur in the environment.

$$\begin{aligned}
&\mathit{free}(a) = \{a\} \\
&\mathit{free}(C\tau_1 \ldots \tau_n) = \bigcup_{i=1}^{n} \mathit{free}(\tau_i) \\
&\mathit{free}(\tau_1 \to \tau_2) = \mathit{free}(\tau_1) \cup \mathit{free}(\tau_2) \\
&\mathit{free}(\Gamma) = \bigcup_{x:\sigma \in \Gamma} \mathit{free}(\sigma) \\
&\mathit{free}(\forall a.\sigma) = \mathit{free}(\sigma) \backslash \{a\}
\end{aligned}$$

Understanding how monotypes occur in the environment is of importance when generalizing types in practice. Monotypes are just polytypes without any quantifier, such that they always occur free in Γ. Generalization of a monotype into a polytype should quantify variables that **only** occur free in that monotype, e.g. they must not occur free in any type in Γ, such that the generalisable variables of a monotype are $\mathit{free}(\tau) \backslash \mathit{free}(\Gamma)$.

If when generalizing monotypes, one quantifies all variables $\mathit{free}(\tau)$ arbitrary monotypes that occur various places in a program may occur both polymorphic and monomorphic. If such an event occurs, then values that have **any** type can occur ($\forall a.a$), which should not validate. For instance, a value id af type $\forall a.a$ would both be valid to use in the expression $id\ 5$ and $id + 5$, which clearly do different things.

Remark 3.1.2. The type $\forall a.a$ is clearly an absurd type (called absurd in Haskell), since it allows one to bypass the type-system.

3.2 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-Milner rules outlined in Figure 3.6. Notice that **Let** introduces types to the environment as polymorphic, which is called *let polymorphism*. In contrast, **Abs** introduces types to the environment as monomorphic.

Additionally, rules which introduce numbers and arithmetic operations can easily be introduced by rules such as Figure 3.7. Let polymorphism is exemplified in Example 3.2.1.

Example 3.2.1. Now that polymorphism has been introduced through the Hindley-Milner type system, an example naturally follows. Let Figure 3.8 be the proof that the program `let` $id = (\lambda x.x)$ `in let` $id_2 = (id\ id)$ `in` $id_2\, 0$ has type `Int`.

3.2.1 Damas-Milner Algorithm W

Typing rules are by themselves not that useful since they need all type information declared ahead of checking and one must deduct what types are necessary to complete a proof. Luckily type inference is a well studied technique. Type inference is the technique of automatically deriving types from minimal information, of which there exist many algorithms. One of the most common inference algorithms that produce typings which adhere to the Hindley-Milner rules, is the Damas-Milner Algorithm W inference algorithm [Dam84; DM82]. When the Damas-Milner Algorithm W discovers a new variable, it assumes that it is the most general type. When the Damas-Milner Algorithm W discovers constraints on types, they are unified into a substitution such that the type is specified further.

For instance, in an abstraction such as $\lambda x.x + 1$ the algorithm will introduce x as some type variable τ_1. When the algorithm discovers $x + 1$, it must constrain τ to have a type which must be the unification of τ and `Int`, since $+$ requires both arguments to be of type `Int`, such that τ must be substituted with the type `Int` written $\tau \mapsto$ `Int`.

The Damas-Milner Algorithm W rules (Figure 3.11) introduce some new concepts such as *fresh variables*, *most general unifier*, and the *substitution set*. Fresh variables are introduced by picking a variable that has not been picked before from the infinite set $\tau_1, \tau_2 \ldots$. In the earlier example where x had type τ, τ was a fresh variable. The substitution set is a mapping from type variables to types (Equation 3.4), which in the earlier example had the instance $\tau \mapsto$ `Int`.

$$S = \{a_1 \mapsto \tau_1, a_2 \mapsto \tau_2 \ldots, a_n \mapsto \tau_n\} \tag{3.4}$$

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

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

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

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

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

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

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

Figure 3.6: Hindley-Milner type rules

- **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. Under the assumption that $x : \tau_1$ exists in the environment, if there is a proof of e having type τ_2, then the abstraction of x must take the type of x to create the type of the body e; $\tau_1 \to \tau_2$.
- **Let** states that if e_1 can be proven to have type σ under the environment Γ, and if e_2 can be proven to have the type τ under the environment $\Gamma, x : \sigma$, then $\texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2$ must have type τ.
- **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 of how "specific" types are. In Hindley-Milner there are only universally quantified types and monomorphic types, thus $\sqsubseteq$ specifies a polytype to a monotype or a polytype to a polytype. For instance $\forall a.a \to a \sqsubseteq b \to b$ or $\forall a.a \to a \sqsubseteq \texttt{Int} \to \texttt{Int}$.
- **Gen** generalizes σ over the type variable a, where a must not occur free in Γ, that is, a must not occur as a monomorphic type variable.

$$\text{Bin op}\ \frac{\Gamma \vdash x : \texttt{Int} \qquad \Gamma \vdash y : \texttt{Int}}{\Gamma \vdash x + y : \texttt{Int}} \qquad \text{Num}\ \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \texttt{Int}}$$

Figure 3.7

ure 3.9). 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.

$$S_1 \cdot S_2 = \{(a \mapsto S_1\tau) \mid (a \mapsto \tau) \in S_2)\} \cup S_1 \tag{3.5}$$

Remark 3.2.1. 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. Transitivity 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 3.1). Unification in the context of the Hindley-Milner type system are outlined in Figure 3.10.

Remark 3.2.2. The Damas-Milner algorithm W is the most popular inference algorithm for Hindley-Milner. Though it remains the most popular, it has some interesting competitors. One of which is that of the constraint solver approach which is also used in OCaml [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.

3.2.2 Instantiation

Another interesting addition introduced by algorithm W in Figure 3.11 is *inst*. *inst* naturally follows from the **Inst** rule in Figure 3.6 but has a slightly different behaviour. The *inst* function does not specify types anymore but simply makes unification of polymorphic types possible.

$$\mathit{inst}(\sigma) = \{\, a \mapsto \mathit{fresh} \mid a \notin \mathit{free}(\sigma) \,\}\sigma \tag{3.6}$$

$$\text{Num}\ \frac{0 \in \mathbb{Z}}{\{id\ : (\forall\tau_1.\tau_1 \to \tau_1), id_2 : \forall\tau_2.\tau_2 \to \tau_2\} \vdash 0 : \texttt{Int}}$$

(a)

$$\text{Gen}\ \frac{\text{Var}\ \dfrac{id_2 : \forall\tau_2.\tau_2 \to \tau_2 \in \{id\ : (\forall\tau_1.\tau_1 \to \tau_1), id_2 : \forall\tau_2.\tau_2 \to \tau_2\}}{\{id\ : (\forall\tau_1.\tau_1 \to \tau_1), id_2 : \forall\tau_2.\tau_2 \to \tau_2\} \vdash id_2 : \forall\tau_2.\tau_2 \to \tau_2} \qquad \forall\tau_2.\tau_2 \to \tau_2 \sqsubseteq \texttt{Int} \to \texttt{Int}}{\{id\ : (\forall\tau_1.\tau_1 \to \tau_1), id_2 : \forall\tau_2.\tau_2 \to \tau_2\} \vdash id_2\ : \texttt{Int} \to \texttt{Int}}$$

(b)

$$\text{App}\ \frac{\text{Figure 3.8b} \qquad \text{Figure 3.8a}}{\{id\ : (\forall\tau_1.\tau_1 \to \tau_1), id_2 : \forall\tau_2.\tau_2 \to \tau_2\} \vdash id_2\,0 : \texttt{Int}}$$

(c)

$$\text{Gen}\ \frac{\text{Var}\ \dfrac{id\ : \forall\tau_1.\tau_1 \to \tau_1 \in \{id\ : (\forall\tau_1.\tau_1 \to \tau_1)\}}{\{id\ : (\forall\tau_1.\tau_1 \to \tau_1)\} \vdash id\ : \forall\tau_1.\tau_1 \to \tau_1} \qquad \forall\tau_1.\tau_1 \to \tau_1 \sqsubseteq \tau_2 \to \tau_2}{\{id\ : (\forall\tau_1.\tau_1 \to \tau_1)\} \vdash id\ : \tau_2 \to \tau_2}$$

(d)

$$\text{Inst}\ \frac{\text{Var}\ \dfrac{id\ : (\forall\tau_1.\tau_1 \to \tau_1) \in \{id\ : (\forall\tau_1.\tau_1 \to \tau_1)\}}{\{id\ : (\forall\tau_1.\tau_1 \to \tau_1)\} \vdash id\ : (\forall\tau_1.\tau_1 \to \tau_1)} \qquad \forall\tau_1.\tau_1 \to \tau_1 \sqsubseteq (\tau_2 \to \tau_2) \to (\tau_2 \to \tau_2)}{\{id\ : (\forall\tau_1.\tau_1 \to \tau_1)\} \vdash id : (\tau_2 \to \tau_2) \to (\tau_2 \to \tau_2)}$$

(e)

$$\text{App}\ \frac{\text{Figure 3.8e} \qquad \text{Figure 3.8d}}{\{id\ : (\forall\tau_1.\tau_1 \to \tau_1)\} \vdash (id\ id) : \tau_2 \to \tau_2}$$

(f)

$$\text{Let}\ \frac{\text{Gen}\ \dfrac{\text{Figure 3.8f} \qquad \tau_2 \notin \mathit{free}(\{id\ : (\forall\tau_1.\tau_1 \to \tau_1)\})}{\{id\ : (\forall\tau_1.\tau_1 \to \tau_1)\} \vdash (id\ id) : \forall\tau_2.\tau_2 \to \tau_2} \qquad \text{Figure 3.8c}}{\{id\ : (\forall\tau_1.\tau_1 \to \tau_1)\} \vdash \texttt{let}\ \ id_2 = (id\ id)\ \ \texttt{in}\ \ id_2\,0 : \texttt{Int}}$$

(g)

$$\text{Let}\ \frac{\text{Gen}\ \dfrac{\text{Abs}\ \dfrac{\text{Var}\ \dfrac{x : \tau_1 \in \{x : \tau_1\}}{\{x : \tau_1\} \vdash x : \tau_1}}{\{\} \vdash (\lambda x.x) : \tau_1 \to \tau_1} \qquad \tau_1 \notin \mathit{free}(\{\})}{\{\} \vdash (\lambda x.x) : \forall\tau_1.\tau_1 \to \tau_1} \qquad \text{Figure 3.8g}}{\{\} \vdash \texttt{let}\ \ id\ = (\lambda x.x)\ \ \texttt{in let}\ \ id_2 = (id\ id)\ \ \texttt{in}\ \ id_2\,0 : \texttt{Int}}$$

Figure 3.8

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

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

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

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

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

Figure 3.9: Substitutions

$$\text{Var equiv } \frac{}{\{\}, a = a}$$

$$\text{Right } \frac{a \notin \mathit{free}(\tau)}{\{a \mapsto \tau\}, \tau = a}$$

$$\text{Left } \frac{a \notin \mathit{free}(\tau)}{\{a \mapsto \tau\}, a = \tau}$$

$$\text{Arrow } \frac{S_1, \tau_1 = \gamma_1 \qquad S_2, S_1\tau_2 = S_1\gamma_2}{S_2 \cdot S_1, \tau_1 \to \tau_2 = \gamma_1 \to \gamma_2}$$

$$\text{Constructor } \frac{S, \tau_1 \to (\tau_2 \cdots \to \tau_n) = \gamma_1 \to (\gamma_2 \cdots \to \gamma_n) \qquad C_1 \equiv C_2}{S, C_1\tau_1, \tau_2 \ldots, \tau_n = C_2\gamma_1, \gamma_2 \ldots, \gamma_n}$$

Figure 3.10: Rules for most general unification

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

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

$$\text{App}$$

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

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

Figure 3.11: Algorithm W

inst (Equation 3.6) 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 3.2.2. Performing some type analysis on Equation 3.7 yields a very rich example of why *inst* is necessary.

$$\begin{aligned} &\texttt{let}\ \ id\ = (\lambda x.x)\ \ \texttt{in} \\ &\texttt{let}\ \ ap\ = (\lambda x.\lambda f.fx)\ \ \texttt{in} \\ &\texttt{let}\ \ doubleid\ = (\lambda x.id(id(x+1)))\ \ \texttt{in}\ \ doubleid \end{aligned} \tag{3.7}$$

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

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

$$\Gamma = \{(id, \forall a.a \to a), (ap, \forall\gamma, \beta.\gamma \to (\gamma \to \beta) \to \beta), (\texttt{x}, \forall\tau.\tau)\} \tag{3.8}$$

$$\mathit{unify}(\tau, \texttt{Int}) = \{\tau \mapsto \texttt{Int}\} \tag{3.9}$$

$$\mathit{unify}(a \to a, \texttt{Int} \to \mu) = \mathit{unify}(\{a \mapsto \texttt{Int}\}a, \{a \mapsto \texttt{Int}\}\mu) \cdot \{a \mapsto \texttt{Int}\} \tag{3.10}$$

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

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

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.

$$
\begin{aligned}
&\mathit{unify}(\mathit{inst}(\forall a.a \to a), \mathit{inst}(\forall \tau.\tau \to \mu)) && (3.11)\\
&= \mathit{unify}(\gamma \to \gamma, \varphi \to \mu)\\
&= \{\varphi \mapsto \mu, \gamma \mapsto \mu\}
\end{aligned}
$$

3.2.3 Recursion

Recursion is a trivial matter once the primitives of the Hindley-Milner type system have been introduced. Recall that in subsection 2.3.2 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. Note, however, that the type must be introduced monomorphically, or else one introduces polymorphic recursion which is undecidable.

Example 3.2.3. If the function `f` defined in Equation 3.12 were to be typed it would need to be introduced as an unknown type to the environment before typing the body of `f`.

$$\texttt{let}\ \ f = (\lambda x.(fx) + 1) \tag{3.12}$$

Let $\Gamma = \{\texttt{f} : \tau, \texttt{x} : \mu\}$. From the application `f x` the unification $\texttt{unify}(\tau, \mu \to \gamma) = \{\tau \mapsto \mu \to \gamma\}$ must be performed, and the resulting type for the

expression is γ. The addition operation forces $\texttt{unify}(\texttt{Int}, \gamma) = \{\gamma \mapsto \texttt{Int}\}$. 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`.

3.2.4 Additional language features

In addition to the rules in Figure 3.11 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 3.1: ADT implementation of decision

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

Decision can be implemented in a variety of ways such as in Listing 3.1 by the use of algebraic data type aligning very much with Church Booleans [Chu85]. Rather decision can be implemented by more conventional methods than

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

(a)

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

(b) (c)

$$\dfrac{\text{3.12b} \qquad \dfrac{\text{3.12a} \qquad \text{3.12c}}{\Gamma, S_6 \;\; S_6 = S_5 \cdot S_4 \cdot S_1' \cdot S_2 \cdot S_3}}{\Gamma \vdash \texttt{if} \;\; e_1 \;\; \texttt{then} \;\; e_2 \;\; \texttt{else} \;\; e_3 : S_6\tau_4, S_6}$$

Figure 3.12: Decision

combinator logic by introducing more inference rules as in Figure 3.12. Additional language syntax features can in most cases be implemented as decision can.

3.2.5 Algebraic data types and type constructors

To implement rules for algebraic data type one must first decide on what the type of an algebraic data type is. If algebraic data types were implemented as in subsection 2.4.1 the type of an algebraic data type like Boolean in Listing 3.1 would be `a -> a -> a` since `BFalse` and `BTrue` must have a handler each. Implementing algebraic data type by this method does not introduce anything to the inference algorithm since every algebraic data type becomes a function. It is common to introduce algebraic data type along type constructors, since such an implementation yields descriptive errors in comparison to generated function types.

Before delving into what type constructors and algebraic data types are, we must understand what a *sum-type* and a *product-type* is. Product-types are in simply terms n-arity tuples expressed by the conjunction operator $\times$ such that a two-arity tuple of type `Int` becomes `Int` $\times$ `Int`. Sum-types are in informal terms a unifying umbrella, in which product-types can have a unified name. Sum-types lets us reason with product types of different shapes, as if they were the same type. A sum-type which we have seen earlier is the `Boolean` sum-type that can take the value of the product-type `BFalse` and `BTrue`. The sum-type `Boolean` combines two nullary (nullary becomes the singleton 1 in type theory) product-types by the disjunction operator $+$ `1 + 1`. More exotic types, such as `List`, also requires *type-constructor*s to become generalized. A type constructor in type theory is denoted as abstraction in lambda calculus, such that `List a` becomes λa (note this is not the full type for `List`, that comes later). Recursion in type theory is denoted by the special abstraction operator μ. Now we have all ingredient to represent a type such as `List a`; $\lambda a.\mu rec.1 + (a \times rec)$.

Now it only becomes a matter of introducing sum-types and product-types to the system. Sum-types will exist purely on the type level and product-types will exist purely on the value level.

The `List` algebraic data type has the constructor `Cons` which requires a value of type `a` and a value of type `List a`. To infer `Cons`, we must acquire a fresh variable γ for each type constructor parameter of `List`, that is, `a` such that the type becomes `List` γ. Now all instances of `a` in the constructor declaration of `Cons` must be substituted by γ, such that `Cons` γ `(List` γ`)`.

In the case of construction, the product type `Cons` γ `(List` γ`)` must be unified with the inferred type of the constructor. When the subsequent `List` sum-type appears in the constructor, the appropriate case is handled by substituting the type parameters in the product type declaration like above.

In the case of matching, the program variables are introduced into Γ, such that for a case of `Cons x xs` the types are introduced monomorphically into Γ as $\Gamma, x : \gamma, xs :$ (`List` γ).

3.3 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 3.3.1. *There exists a family of programs which are typeable 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.

$$\texttt{let } dup = (\lambda a.\lambda f.faa) \texttt{ in} \tag{3.13}$$

$$\texttt{let } deep = (\lambda x.dup\ (dup\ (dup\ (\ldots)))) \texttt{ in } \ldots \tag{3.14}$$

Equation 3.13 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 3.11. 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 Equation 3.13 to run in $\Omega(2^n)$ but an important observation for why Equation 3.13 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 $O(n)$.

$$\begin{aligned} &\texttt{let } tuple = (\lambda a.\lambda b.\lambda f.fab) \texttt{ in} \\ &\texttt{let } one = tuple\ tuple\ tuple \texttt{ in} \\ &\texttt{let } two = tuple\ one\ one \texttt{ in} \\ &\texttt{let } three = tuple\ two\ two \texttt{ in} \end{aligned} \tag{3.15}$$

The trick to induce an exponential running time is demonstrated with the pathological program in Equation 3.15. 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 Equation 3.15, 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 3.16.

$$inst(a \to b \to (a \to b \to c) \to c) \to inst(a \to b \to (a \to b \to c) \to c) \to \gamma \to \gamma. \tag{3.16}$$

The first and second instantiations will contain different type variable such that they are not structurally equivalent (Equation 3.17).

$$(\tau \to \mu \to (\tau \to \mu \to \phi) \to \phi) \to (\varphi \to \zeta \to (\varphi \to \zeta \to \delta) \to \delta) \to \gamma \to \gamma. \tag{3.17}$$

□

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 contains problems 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 3.2 in L yields a blowup of 2^n (Listing 7.1). Figure 3.13 shows the relationship between the program typed in L and the theoretical time of 2^n.

Listing 3.2: Nested tuples with different type variables

```
1 fun tuple a b f = f a b;
2 fun one = tuple tuple tuple;
3 fun two = tuple one one;
4 fun three = tuple two two;
5 fun four = tuple three three;
6 fun main = 0;
```

3.4 Higher level type systems

The Hindley-Milner type system is a constrained variant of more general type systems. The typed lambda caculus can come in various forms, the simplest of which is the simply typed lambda calculus (Figure 3.3). Throughout

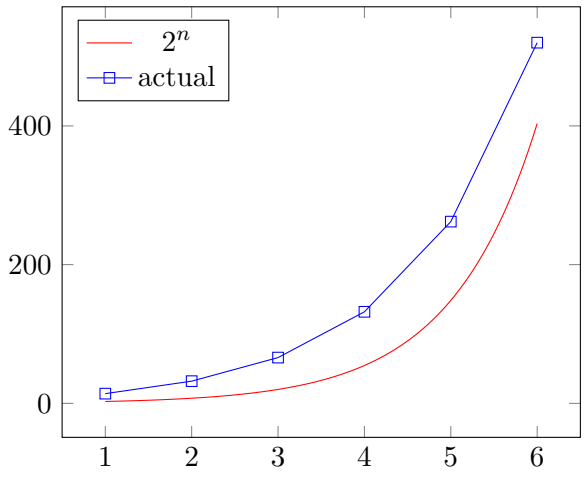


Figure 3.13: Plot of type variables in Hindley-Milner type systems

this section we have explored a type system that introduces polymorphism and type constructors.

3.4.1 System F

System F generalizes the notion of polymorphism by letting terms become polymorphic within other polymorphic types. System F allows exotic types such as the signature $\forall a.\forall c.(\forall b.b \to \texttt{Int}) \to a \to c \to \texttt{Int}$, which could be a function that takes two potentially different typed values a and c and turns them into an integer and adds them (Listing 3.3).

Listing 3.3: Rank 2 type in System F

```
1 fun f makeNum a c =
2     (makeNum a) + (makeNum c);
```

We say that a type has a rank when interesting ourselves with nested quantification. The type of by the function in Listing 3.3 requires rank-2 types to be expressible. Generally, when quantifiers occur on the left side of $\to$ they must be locally quantified thus they increase the rank. Unfortunately checking System F is undecidable, thus sound inference is not possible [Wel99].

3.4.2 System F$\underline{\omega}$

System F$\underline{\omega}$ is orthogonal with System F, despite the similar name. System F$\underline{\omega}$ introduces type constructors, which have already been explored. Generally, System F$\underline{\omega}$ introduces a rule that allows types to become parameterized, like abstraction in the lambda calculus, such that a type that depends on a parameter a may occur as $\lambda a.\tau$. For instance, a generalized tuple may be

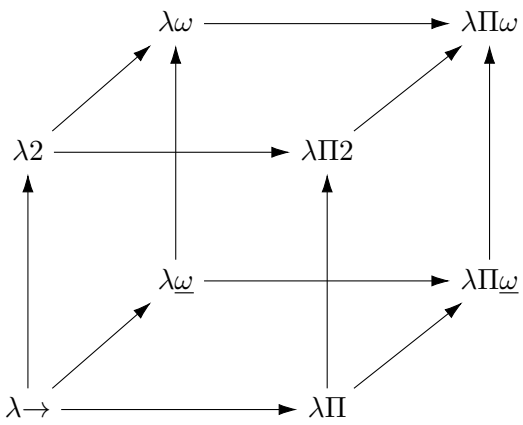


Figure 3.14:

- $\lambda \rightarrow$ is the simply typed lambda calculus without polymorphism.
- $\lambda\underline{\omega}$ is System F$\underline{\omega}$.
- $\lambda 2$ is System F.
- Π introduces dependent types.
- Various combinations of the aforementioned type systems can occur as they are combined by the diagram. For instance, $\lambda\omega$ is System Fω, the combination of $\lambda 2$ and $\lambda\underline{\omega}$.

have the type $\lambda a.\lambda b.a \times b$. System F$\underline{\omega}$ by itself is not of particular interest, since one cannot introduce generalized types without System F.

3.4.3 Dependent types

System F$\underline{\omega}$ and System F are orthogonal, but compliment each other well and can represent types similar to types that can occur in traditional programming languages. Dependent types on the other hand, introduce something quite different. Dependent types introduce the notion of letting types be a product of an expression. Allowing expressions to become a part of the type system, significantly increases the preciseness of types. Dependent types are the foundation of many theorem provers, since one can encode various problems into programs. For instance, one can precisely define the type for a matrix matrix multiplication. Let two matrices m_1 with the dimensions i and j encoded into m_1's type, and m_2 with the dimensions j and j encoded into m_2's type, be multiplied, then the resulting matrix m_3 has the dimensions i and k.

Figure 3.14 shows the *lambda cube*, introduced in [Bar91] which encapsulates the family of formal type systems.

3.5 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 by itself, is a retrofitted version of System F. 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 4

Program evaluation

The untyped lambda calculus may provide a simple interface for programming, but does not pair very well with the modern computer. *Interpreting* is a common technique for evaluating the untyped lambda calculus. An interpreter is an algorithm, usually implemented in a more low-level language, which interprets a program; essentially a program that runs a program.

Throughout this section, we will first consider various evaluation strategies and then consider how one evaluates the lambda calculus.

4.1 Evaluation strategies

When evaluating the untyped lambda calculus one has to choose an evaluation strategy. The choice of evaluation strategy has a large impact on aspects such as complexity guarantees. 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 4.1: Program that doubles values

```
1  fun main =
2    fun double x = x + x;
3    let a = double 10;
4    double 10;
```

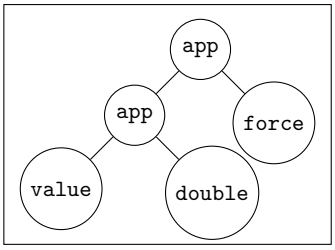
By the call by value semantics, Listing 4.1 eagerly evaluates every expression. Clearly the variable `a` is never used but under the call by value semantics everything is eagerly evaluated. Every expression is evaluated in logical order in the call by value evaluation strategy.

Listing 4.2: Implementation of call by name

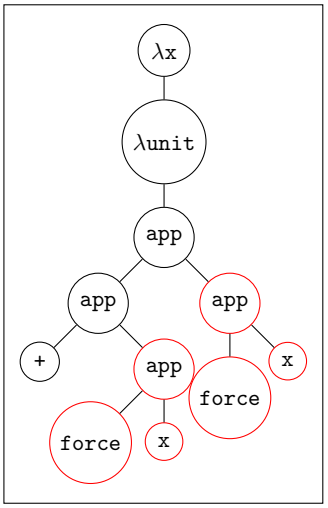
```
1  fun main =
2    fun suspend x unit = x;
3    fun force x = x 0;
4    let value = suspend 10;
5    fun double x =
6        fun susExpensiveOp unit =
7            (force x) + (force x);
8        susExpensiveOp;
9    let a = double value;
10   force (double value);
```

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 4.2 call by name has been implemented by the use of various functions such as the two constant functions `suspend` and `force`. `susExpensiveOp` ensures that the forcing (evaluation) of `x` never occurs until the caller of `double` forces the result. By the aforementioned semantics of call by name in the context of the program in Listing 4.2 `a` is never forced thus the computation is never performed. The implementation 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 4.2 `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



(a) The last expression of the program.



(b) The expression tree for `double`

Figure 4.1

as dynamic programming. To understand this better observe the expression tree for Listing 4.2 in Figure 4.1. Clearly the two red subtrees in Figure 4.1b are identical thus they may be memoized such that the forcing of `x` only occurs once. More generally if the execution environment supports lazy evaluation, once an expression has been forced it is remembered.

4.2 Runtime environments

Now that the untyped lambda calculus has been introduced, implemented and validated efficiently the question of execution naturally follows. There exists many different well understood strategies to implement an execution environment for the untyped lambda calculus. Naively it may seem straightforward to evaluate the untyped lambda calculus mechanically by β-reductions, but doing so brings upon some problems when implementing an interpreter.

4.3 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.

$$x \tag{4.1}$$
$$F \tag{4.2}$$
$$YE \tag{4.3}$$

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

Example 4.3.1.

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

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

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 $\sigma \mapsto S, \kappa \mapsto K, \iota \mapsto I$.

4.3.1 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 2.2, Equation 2.1 and Equation 2.3) 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 4.3.1. *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 4.5 such that the amount of extra K terms minus the I becomes $\text{variable\_references} * (\text{unused\_abstractions} - 1) = 3 * (3 - 1)$:

$$S(S(KKI)(KKI))(KKI) \tag{4.5}$$

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 $f_1 f_1 \ldots f_1$ can in fact be changed to $f_1 f_2 \ldots f_n$ 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 $\Omega(n^2)$. □

Example 4.3.2. Observe the size of Equation 4.6 in comparison to Equation 7.1.

$$
\begin{aligned}
& \lambda f_1.\lambda f_2.f_1 f_2 \qquad (4.6)\\
=&\lambda f_1.\sigma(\lambda f_2.f_1)(\lambda f_2.f_2)\\
=&\lambda f_1.(\sigma(\kappa f_1))(\iota)\\
=&\sigma(\lambda f_1.\sigma(\kappa f_1))(\lambda f_1.\iota)\\
=&\sigma(\sigma(\lambda f_1.\sigma)(\lambda f_1.\kappa f_1))(\kappa\iota)\\
=&\sigma(\sigma(\kappa\sigma)(\sigma(\lambda f_1.\kappa)(\lambda f_1.f_1)))(\kappa\iota)\\
=&\sigma(\sigma(\kappa\sigma)(\sigma(\kappa\kappa)(\iota)))(\kappa\iota)\\
=&S(S(KS)(S(KK)(I)))(KI)
\end{aligned}
$$

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].

4.4 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.

4.4.1 Symbols and notation

In this section, expressions will be written in a different typesetting since some expressions and symbols mean something different than in other sections. For instance `x` $\rightarrow$ `y` means the evaluation of `x` results in `y`, while $x \rightarrow y$ means the type of a function that takes a value of type x and returns a value of type y.

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

Equation 4.7.

$$\begin{aligned} \texttt{x, y, z, f, v, } \gamma \texttt{ := Var} \\ \texttt{e, p, l, o := Exp} \\ \Gamma\texttt{, } \Sigma\texttt{, } \Theta \texttt{ := Heap} \\ \texttt{E := Environment} \end{aligned} \tag{4.7}$$

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

4.4.2 The abstract evaluation model

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

Remark 4.4.1. In the section regarding the semantics of reduction strategies, environments will exist as singleton sets, named *substitutions*.

$$\{\texttt{x} \mapsto \texttt{y}\}\texttt{x = y} \tag{4.8}$$

Substitutions are performed like shown in Equation 4.9, which states "x is substituted by y".

Evaluation strategies (section 4.1) 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 `(`λ`x.x) y`, `x` must be substituted by `y` such that the expression then becomes `x` with the substitution $\{\texttt{x} \mapsto \texttt{y}\}$ and finally becomes `y` after the substitution has occurred. The rules in Figure 4.2 display a simple set of rules for proving call by name lambda calculus programs.

- The Abs and Var rules (Figure 4.2a and Figure 4.2b) 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 4.2d) states that "`l p` can be proved to evaluate to `o` if `l` can be proved to be `(`λ`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.

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

(a)

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

(b)

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

(c)

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

(d) A simple application rule

Figure 4.2: Simple call by name lambda calculus

We must introduce rules for how substitutions should act upon encountering lambda calculus terms (Equation 4.9).

$$\begin{aligned} &\{\texttt{x} \mapsto \texttt{e}\}\texttt{x} = \texttt{e} \qquad (4.9)\\ &\{\texttt{x} \mapsto \texttt{e}\}\texttt{p} = \texttt{p}\\ &\texttt{E(l p)} = \texttt{(El)(Ep)}\\ &\texttt{E(}\lambda\texttt{x.e)} = \texttt{(}\lambda\texttt{x.Ee)} \qquad (4.10) \end{aligned}$$

Ambiguous programs

$$\texttt{(}\lambda\texttt{x.(}\lambda\texttt{x.x) 0) 1} \qquad (4.11)$$

Proving Equation 4.11 under the rules in Equation 4.9 yields a case for more thorough substitution rules. By inspection one can determine that a simple program like Equation 4.11 yields the symbol `0`, but alas this is not the case. The first step to prove Equation 4.11 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 4.12).

$$\texttt{(}\lambda\texttt{x.1) 0} \qquad (4.12)$$

Clearly Equation 4.12 changed the meaning of the program. If we continue the proof which states that the program in Equation 4.12 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 4.10 and adding the two rules in Equation 4.13 solves this type.

$$\begin{aligned} &\texttt{E}(\lambda\texttt{x.e}) = (\lambda\texttt{x.Ee}) && (\texttt{x} \mapsto \texttt{p}) \notin \texttt{S} \\ &\texttt{E}(\lambda\texttt{x.e}) = (\lambda\texttt{x.}(\texttt{E}\backslash\{\texttt{x} \mapsto \texttt{p}\})\texttt{e}) && (\texttt{x} \mapsto \texttt{p}) \in \texttt{S} \end{aligned} \tag{4.13}$$

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 4.4.1. With the aforementioned rules, programs can now be proved. Note that the Substitution rule in Figure 4.3 is simply the substitution semantics from Equation 4.13 made clearer. Let the program in Equation 4.14, where `0` is a symbol of any type, be subject to the rules in Figure 4.2, which solves to Figure 4.3.

$$((\lambda\texttt{f.}\lambda\texttt{x.f x})\ (\lambda\texttt{x.x}))\ \texttt{0} \tag{4.14}$$

Variable capture is the basis for some practical difficulties when designing evaluation rules for the untyped lambda calculus. Consider the following sub-program $(\lambda\texttt{x.y})\ \texttt{g}$ 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 4.13 yields $(\lambda\texttt{x.x})\ \texttt{g}$ 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 4.4.2. 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 be written 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. 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 `x` is discovered, replace it with some $\gamma \in V_2$ and let $V_2 = V_2\backslash\{\gamma\}$. The function `fresh` picks a fresh variable name γ from V_2 and updates V_2 to $V_2\backslash\{\gamma\}$. Letting the previous example $\{\texttt{y} \mapsto \texttt{x}\}(\lambda\texttt{x.y})\ \texttt{g}$, be

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

(a)

$$\begin{array}{l}\text{Abs}\\ \text{App}\end{array}\ \frac{\dfrac{}{\texttt{(}\lambda\texttt{f.}\lambda\texttt{x.f x) }\rightarrow\texttt{ (}\lambda\texttt{f.}\lambda\texttt{x.f x)}}\quad\text{Figure 4.3a}}{\texttt{(}\lambda\texttt{f.}\lambda\texttt{x.f x) (}\lambda\texttt{x.x) }\rightarrow\texttt{ (}\lambda\texttt{x.(}\lambda\texttt{x.x) x)}}$$

(b)

$$\text{Abs}\ \frac{\dfrac{}{\texttt{(}\lambda\texttt{x.x) }\rightarrow\texttt{ (}\lambda\texttt{x.x)}}\quad\dfrac{\dfrac{}{\texttt{0 }\rightarrow\texttt{ 0}}\ \text{Var}}{\texttt{\{x }\mapsto\texttt{ 0\}x }\rightarrow\texttt{ 0}}\ \text{Substitution}}{\dfrac{\texttt{(}\lambda\texttt{x.x) 0 }\rightarrow\texttt{ 0}}{\texttt{\{x }\mapsto\texttt{ 0\}(}\lambda\texttt{x.x) x }\rightarrow\texttt{ 0}}\ \text{Substitution}}\ \text{App}$$

(c)

$$\text{App}\ \frac{\text{Figure 4.3b}\qquad\text{Figure 4.3c}}{\texttt{((}\lambda\texttt{f.}\lambda\texttt{x.f x) (}\lambda\texttt{x.x)) 0 }\rightarrow\texttt{ 0}}$$

Figure 4.3

subject to α-conversion solves the problem of ambiguity ($\texttt{\{y }\mapsto\texttt{ }\tau\texttt{\}(}\lambda\texttt{x.}\tau\texttt{) g}$), since `y` could be renamed to a variable introduced through `fresh`, thus no ambiguity occurs. α-conversions will be further explored in future refinements of Figure 4.2 in the from of renaming through the Let rule.

The heap

Heaps, like environments in typing (Equation 3.3), 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 4.4, introduces a heap which states that that the semantics must bring a heap along. The rules in Figure 4.4 are quite different from the rules in Figure 4.2.

- 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.

$$\frac{}{\Gamma,\ (\lambda \mathtt{x.e}) \rightarrow \Gamma,\ (\lambda \mathtt{x.e})}\ \text{Abs} \qquad \frac{\Gamma \cup \{\mathtt{x} \mapsto \mathtt{e}\},\ \mathtt{p} \rightarrow \Theta,\ \mathtt{l}}{\Gamma,\ \mathtt{let\ x = e\ in\ p} \rightarrow \Theta,\ \mathtt{l}}\ \text{Let}$$

(a) (b)

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

(c)

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

(d)

Figure 4.4: Call by name lambda calculus with environments

- 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 4.4 are not any more powerful that the rules in Figure 4.2, but are a basis for lazy evaluation.

Lazy evaluation

With the revised semantics in Figure 4.4, 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 4.5 be a depiction of the dependency graph of Equation 4.15 under the rules in Figure 4.4.

```
let k = (λz.z) (λf.f) in          (4.15)
let x = k in
let y = k in
x + y
```

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

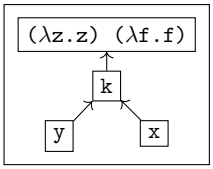


Figure 4.5: Expression dependencies

evaluated value for future use." is required to support sharing computed values. The rule in Figure 4.6 replaces the Var rule, and introduces a sub-

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

Figure 4.6

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 4.15 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 4.7. One consideration remains, the App rule does not

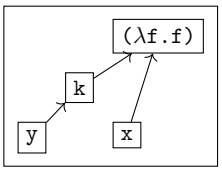


Figure 4.7: Expression dependencies after evaluating `x`

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 variable capture can only occur in the Let rule, since the Let rule is the only rule of which can introduce name bindings. Dealing with ambiguity is a matter of ensuring that variables are distinct. Applying the technique from section 4.4.2 properly lets us evaluate programs without ambiguity.

Consider a previous case of variable capture Equation 4.16.

$$\{\texttt{x} \mapsto \texttt{z},\ \texttt{y} \mapsto \texttt{x},\ \ldots\}(\lambda \texttt{x.y})\ \texttt{g} \tag{4.16}$$

None of the changes so far have any impact on the falsity of the expression. Consider that `x` and `y` must have been bound through a Let expression. Consider also that some variable `k` cannot be subject to variable capture if $\texttt{k} \notin \texttt{Bound}(\lambda \texttt{x.e})$. Naturally if `k` is unique, that is, it is introduced through the `fresh` function from section 4.4.2 then `k` can never occur bound. The obvious rule from these considerations must be new Let rule defined in Figure 4.8. The correctness of Figure 4.8 and the aforementioned considerations

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

Figure 4.8

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 4.9. Notice that Figure 4.9 also must

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

$$\frac{\texttt{n} \in \mathbb{Z}^{+}}{\Gamma,\ \texttt{n} \rightarrow \Gamma,\ \texttt{n}}\ \text{Num}$$

Figure 4.9

accompany a Num rule which introduces integers to the system.

Remark 4.4.3. Notice that the Bin op rule in Figure 4.9 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 4.4.2. 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 4.17 and proved in Figure 4.10.

$$\texttt{let y = (1 + 1) in } (\lambda \texttt{x.}(\lambda \texttt{y.x + y) x) y} \tag{4.17}$$

Notice that the left branch and right branch in Figure 4.10c are not identical.

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

(a)

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

(b)

$$\dfrac{\text{Num}\ \dfrac{}{\{\},\ \texttt{1} \rightarrow \{\},\ \texttt{1}} \quad \text{Num}\ \dfrac{}{\{\},\ \texttt{1} \rightarrow \{\},\ \texttt{1}}}{\text{Bin op}\ \dfrac{}{\{\},\ \texttt{1 + 1} \rightarrow \{\},\ \texttt{2}}} \qquad \dfrac{}{\{\},\ \texttt{2} \rightarrow \{\},\ \texttt{2}}\ \text{Num}$$

$$\dfrac{\text{Var}\ \dfrac{}{\{\gamma \mapsto \texttt{(1 + 1)}\},\ \gamma \rightarrow \{\gamma \mapsto \texttt{2}\},\ \texttt{2}} \qquad \dfrac{}{\{\gamma \mapsto \texttt{2}\},\ \gamma \rightarrow \{\gamma \mapsto \texttt{2}\},\ \texttt{2}}\ \text{Var}}{\text{Bin op}\ \dfrac{}{\{\gamma \mapsto \texttt{(1 + 1)}\},\ \gamma \texttt{ + } \gamma \rightarrow \{\gamma \mapsto \texttt{2}\},\ \texttt{4}}}$$

(c)

$$\dfrac{\text{Figure 4.10a} \qquad \text{App}\ \dfrac{\text{Figure 4.10b} \qquad \text{Figure 4.10c}}{\{\gamma \mapsto \texttt{(1 + 1)}\},\ (\lambda \texttt{y.}\gamma \texttt{ + y})\ \gamma \rightarrow \{\gamma \mapsto \texttt{2}\},\ \texttt{4}}}{\text{App}\ \dfrac{}{\{\gamma \mapsto \texttt{(1 + 1)}\},\ (\lambda \texttt{x.}(\lambda \texttt{y.x + y) x})\ \gamma \rightarrow \{\gamma \mapsto \texttt{2}\},\ \texttt{4}}}$$

$$\text{Let}\ \frac{}{\{\},\ \texttt{let y = (1 + 1) in } (\lambda \texttt{x.}(\lambda \texttt{y.x + y) x) y} \rightarrow \{\gamma \mapsto \texttt{2}\},\ \texttt{4}}$$

(d)

Figure 4.10: The proof for the program in Equation 4.17

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 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 4.11. A garbage collection rule as described in Figure 4.11

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

Figure 4.11: A Let rule which cleans up after itself

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 various 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 4.4c, 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 accomplished by introducing a set of expressions `N`, such that all evaluations are written $\to_{\texttt{N}}$ (Figure 4.12). In addition to Figure 4.12, there must also be an accompa-

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

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

Figure 4.12: 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 4.13). `R` is the set of

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

Figure 4.13: 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 4.13 is too fine. Figure 4.14 defines a garbage collection

$$\frac{\Sigma,\ \mathtt{e} \rightarrow_{\mathtt{N}} \Theta,\ \mathtt{y} \qquad \Sigma = \mathtt{Prune}(\Gamma,\ \{\mathtt{e}\} \cup \mathtt{N})}{\Gamma,\ \mathtt{e} \rightarrow_{\mathtt{N}} \Theta,\ \mathtt{y}}\ \mathrm{GC}$$

Figure 4.14: A rule which prunes unused values

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 4.18

$$\mathtt{Prune}(\Gamma,\ \{\}) = \{\} \tag{4.18}$$

$$\begin{aligned}\mathtt{Prune}(\Gamma,\ \{\mathtt{e}\} \cup \mathtt{N}) &= \mathtt{P} \cup \mathtt{Prune}(\Gamma \setminus \mathtt{P},\ \mathtt{R} \cup \mathtt{N}) \\ &\quad \mathtt{where}\ \mathtt{P} = \{\mathtt{x} \mapsto \mathtt{y} \mid \mathtt{x} \in \mathtt{free(e)} \wedge \mathtt{x} \mapsto \mathtt{y} \in \Gamma\} \\ &\qquad \mathtt{R} = \{\mathtt{y} \mid \mathtt{x} \mapsto \mathtt{y} \in \mathtt{P}\}\end{aligned} \tag{4.19}$$

4.4.3 Interpreting programs

Now that the semantics for evaluation of the lambda calculus have been presented, a machine naturally follows. A machine which is very alike the natural semantics is presented in Figure 4.16 where the `subst` function is defined as in Figure 4.15. An evaluation is written as an evaluation `eval(Γ, e)`, a transition $\Rightarrow$ and a ending state `(Θ, y)`. Furthermore, a transition written `eval(Γ, e)` $\Rightarrow^*$ `(Θ, y)` depicts that through some sequence of transitions, the evaluation `eval(Γ, e)` reaches the state `(Θ, y)`.

Theorem 4.4.1. *If* $\Gamma,\ e \rightarrow \Theta,\ y$ *is provable in the natural semantics by repeated derivations, the machine in Figure 4.16 arrives at the same conclusion.*

Proof. All non-terminal cases will be proved by induction. Since only one machine case is applicable for every shape that a term can take, it suffices

```
          subst(f ≡ x, t, λx.e) = λx.e
          subst(f ≢ x, t, λx.e) = λx.subst(f, t, e)
             subst(f ≡ x, t, x) = t
             subst(f ≢ x, t, x) = x
subst(f ≡ x, t, let x = e in p) = let x = e in p
subst(f ≢ x, t, let x = e in p) = let x = e' in p'
                                  where e' = subst(f, t, e),
                                        p' = subst(f, t, p)
               subst(f, t, l p) = l' p'
                                  where l' = subst(f, t, l),
                                        p' = subst(f, t, p)
             subst(f, t, x ⊕ y) = x' ⊕ y'
                                  where x' = subst(f, t, x),
                                        y' = subst(f, t, y)
            subst(f, t, n ∈ ℤ⁺) = n
```

Figure 4.15: A function `subst` which states "substitute `f` in with `t` in some expression".

```
              eval(Γ, λx.e) = (Γ, λx.e)
               eval(Γ, l p) = eval(Θ, subst(x, p, e))
                              where (Θ, λx.e) = eval(Γ, l)
      eval(Γ ∪ {x ↦ e}, x)= (Θ ∪ {x ↦ y}, y)
                              where (Θ, y) = eval(Γ, e)
  eval(Γ, let x = e in p) = eval(Γ ∪ {γ ↦ e}, l)
                              where γ = fresh,
                                    l = subst(x, γ, p)
             eval(Γ, x ⊕ y) = (Σ, n ⊕ t)
                              where (Θ, n) = eval(Γ, x),
                                    (Σ, t) = eval(Θ, y)
            eval(Γ, n ∈ ℤ⁺) = (Γ, n)
```

Figure 4.16: An algorithm for evaluating the lazy lambda calculus

to prove that for any one shape, the respective machine case will behave like the natural semantics, if all other cases also behave like the natural semantics.

Case 1: Abs states that Γ, $(\lambda x.e) \to \Gamma$, $(\lambda x.e)$, such that it must trivially be true that $\texttt{eval}(\Gamma, \lambda x.e) \Rightarrow (\Gamma, \lambda x.e)$.

Case 2: Num is trivially true, by the same reasoning as Abs.

Case 3: App states that for Γ, $l\ p \to \Sigma$, o to be derivable, then Γ, $l \to \Theta$, $(\lambda x.e)$ and Θ, $\{x \mapsto p\}e \to \Sigma$, o must both be derivable. If the premises are derivable, then the conclusion is derivable.If the left premise is derivable then Equation 4.20 holds.

$$\begin{aligned} & \texttt{eval}(\Gamma, l) && \text{machine instruction for left premise} \\ \Rightarrow^* \ & (\Theta, \lambda x.e) && \text{ind. hyp.} \end{aligned} \tag{4.20}$$

Furthermore, if the second premise is derivable then Equation 4.21.

$$\begin{aligned} & \texttt{eval}(\Theta, \texttt{subst}(x, p, e)) && \text{machine instruction for right premise} \\ \Rightarrow^* \ & (\Sigma, o) && \text{ind. hyp.} \end{aligned} \tag{4.21}$$

Thus the conclusion is derivable by Equation 4.21.

$$\begin{aligned} & \texttt{eval}(\Gamma, l\ p) && \text{conclusion} \\ \Rightarrow^* \ & (\Theta, \lambda x.e) && \text{Equation 4.20} \\ \Rightarrow \ & \texttt{eval}(\Theta, \texttt{subst}(x, p, e)) && \text{machine instruction} \\ \Rightarrow^* \ & (\Sigma, o) && \text{Equation 4.21} \end{aligned} \tag{4.22}$$

$$\tag{4.23}$$

Case 4: Var states that for $\Gamma \cup \{x \mapsto e\}$, $x \to \Theta \cup \{x \mapsto y\}$, y to be derivable then Γ, $e \to \Theta$, y must be derivable. If Γ, $e \to \Theta$, y is derivable, then Equation 4.24.

$$\begin{aligned} & \texttt{eval}(\Gamma \cup \{x \mapsto e\}, x) && \text{conclusion} \\ \Rightarrow \ & \texttt{eval}(\Gamma, e) && \text{machine instruction for premise} \\ \Rightarrow^* \ & (\Theta, y) && \text{ind. hyp.} \\ \Rightarrow \ & (\Theta \cup \{x \mapsto y\}, y) && \text{conclusion} \end{aligned} \tag{4.24}$$

Case 5: For Let to be derivable then $\Gamma \cup \{\gamma \mapsto \texttt{e}\}, \{\texttt{x} \mapsto \gamma\}\texttt{p} \rightarrow \Theta, \texttt{l}$ for some γ that does not occur in `e` or `p`, must be derivable. The primary concern is that the freshness of `fresh` must hold, which is a constraint easily realized in the machine. If the derivation holds, then Equation 4.25.

$$\begin{aligned} & \texttt{eval}(\Gamma, \texttt{let x = e in p}) && \text{conclusion} \\ & && (4.25) \\ \Rightarrow\ & \texttt{eval}(\Gamma \cup \{\gamma \mapsto \texttt{e}\}, \texttt{subst}(\texttt{x}, \gamma, \texttt{p})) && \text{machine instruction for premise} \\ \Rightarrow^*\ & (\Theta, \texttt{l}) && \text{ind. hyp.} \end{aligned}$$

Case 6: Bin op must show that $\Gamma, \texttt{x} \rightarrow \Theta, \texttt{n}$ and $\Theta, \texttt{y} \rightarrow \Sigma, \texttt{t}$ are derivable for $\Gamma, \texttt{x} \oplus \texttt{y} \rightarrow \Sigma\ \texttt{n} \oplus \texttt{t}$ to be derivable. If $\Gamma, \texttt{x} \rightarrow \Theta, \texttt{n}$ is derivable then Equation 4.26.

$$\begin{aligned} & \texttt{eval}(\Gamma, \texttt{x}) && \text{machine instruction for left premise} && (4.26) \\ \Rightarrow^*\ & (\Theta, \texttt{n}) && \text{ind. hyp.} \end{aligned}$$

Moreover, if $\Theta, \texttt{y} \rightarrow \Sigma, \texttt{t}$ holds then Equation 4.27.

$$\begin{aligned} & \texttt{eval}(\Theta, \texttt{y}) && \text{machine instruction for right premise} && (4.27) \\ \Rightarrow^*\ & (\Sigma, \texttt{t}) && \text{ind. hyp.} \\ \Rightarrow\ & (\Sigma, \texttt{n} \oplus \texttt{t}) && \text{conclusion} \end{aligned}$$

Finally, if Equation 4.26 and Equation 4.27 both hold then

$$\begin{aligned} & \texttt{eval}(\Gamma, \texttt{x} \oplus \texttt{y}) && \text{conclusion} && (4.28) \\ \Rightarrow^*\ & (\Theta, \texttt{n}) && \text{Equation 4.26} \\ \Rightarrow\ & \texttt{eval}(\Theta, \texttt{y}) && \text{machine instruction} \\ \Rightarrow^*\ & (\Sigma, \texttt{t}) && \text{Equation 4.27} \\ \Rightarrow\ & (\Theta, \texttt{n} \oplus \texttt{t}) && \text{machine instruction} \end{aligned}$$

□

The inverse case; showing that any program that the machine evaluates, is provable in the natural semantics, is also theoretically interesting, but does yield much practical value. The algorithm Figure 4.16 will shortly be superseded, but will acts as a steppingstone.

A CPS machine

The algorithm in Figure 4.16 is minimal but suffers from some practical issues. Primarily, Figure 4.16 cannot evaluate infinite programs in finite space, since it's recursive invocations are not in tail call position. In the machine introduced in [Ses97] (which will be named the *stack machine*), the primary component is 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 4.17 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 machine from Figure 4.16 into a *continuation-passing style* 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 $\texttt{cont} : \Gamma \times S \times \lambda \to \lambda$, that is, there is no returning. The algorithm for the CPS conversion is as follows:

If an instruction induces a recursive invocation, bound to a name `x`, the instruction should be converted to a continuation `cont` where the continuation binds the result to `x`. The continuation `cont`'s body should now be the rest of the instructions that follow the converted recursive invocation. The only instruction, other than trivial operations, that the converted functions should perform, is pushing `cont` to the continuation stack `S` and then recursively invoke the function for the first continuation. The algorithm is recursively invoked on the body of `cont`, such that other recursive invocations are also converted to continuations.

The textual explanation of the conversion can be difficult to follow, so an example has been prepared in Example 4.4.3.

Example 4.4.3. The recursive Fibonacci function in Equation 4.29 can be converted by converting the $\texttt{F}_{\texttt{n - 1}}$ and $\texttt{F}_{\texttt{n - 2}}$ invocations to continuations as in Equation 4.30.

$$
\begin{aligned}
&\texttt{fib(0) = 0} \\
&\texttt{fib(1) = 1} \\
&\texttt{fib(n) = } \texttt{F}_{\texttt{n-1}} \texttt{ + } \texttt{F}_{\texttt{n-2}} \\
&\quad\texttt{where } \texttt{F}_{\texttt{n-1}} \texttt{ = fib(n - 1),} \\
&\qquad\quad \texttt{F}_{\texttt{n-2}} \texttt{ = fib(n - 2)}
\end{aligned}
\tag{4.29}
$$

```
      continue([], n) = n                                    (4.30)
continue(cont :S, n) = cont(S, n)
           fib(S, 0) = continue(S, 0)
           fib(S, 1) = continue(S, 1)
           fib(S, n) = fib(cont:  S, n - 1)
                  where cont(S', F_{n-1}) =
                          fib(cont' :S', n - 2)
                     where cont'(S'', F_{n-2}) =
                             continue(S'', F_{n-1} + F_{n-2})
```

To prepare the CPS machine, 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 4.18. In it's essence, the algorithm in Figure 4.18 evaluates terms in normal order, recording sharing (Var) and branching (App, Bin op), until it reaches a terminal expression. When Figure 4.18 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.

Since this CPS variant of a lazy evaluation machine is not equivalent to the stack machine, a proof of the machine's expressiveness in respect to the natural semantics is required. Fortunately, the CPS machine is a direct conversion from the machine in Figure 4.16, thus Theorem 4.4.1 holds for the CPS machine also.

Lazy renaming

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 4.16 is defined to be eager, which naturally does not follow the philosophy of lazy computation.

Lazy renaming can be regarded as allocating closures, that is, pairing expressions with metadata that allows sound evaluation. The previous introduction of environments in subsection 4.4.2, in a from which allows them to exist as not only singletons, is part of the recipe for lazy renaming. The other component of lazy renaming is modifying the representation of values stored in the heap, such that the heap now contains mappings from variables to tuples of expressions and environments (Equation 4.31).

$$\Gamma = \{x_1 \mapsto (e_1, E_1), x_2 \mapsto (e_2, E_2) \ldots x_n \mapsto (e_n, E_n)\} \quad (4.31)$$

To implement this into the CPS machine, notice once again that the only case which introduces new bindings is the let expressions. The machine

```
        eval(Γ, S, l p) = eval(Γ, p :S, l)
    eval(Γ, p :S, λx.e) = eval(Γ, S, subst(x, p, e))
```

Figure 4.17: 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 `=`.

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

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

Figure 4.18: A CPS algorithm for a evaluating the lazy lambda calculus

must bring an environment along when evaluating such that the "current" environment can be bound with let introduced heap bindings. Furthermore, renaming is resolved when a variable is referenced.

Before considering a machine with lazy renaming, consider a practical implication of an environment that maps *all* variable names to fresh ones. If an environment contains mappings that exceed the domain of free variables for a paired expression, one can impose *space leaks*. Generally, space leaks have unfortunate consequences, such as not letting a garbage collector free variables. More precisely, for some expression `e` and environment `E`, the superfluous variables that survive garbage collection are at least $\texttt{E} \setminus \mathit{free}(\texttt{e})$. The superfluous variables that survive can indeed point to other variables in the heap, which also contain superfluous variables, thus space leaks can become quite apparent on pathological inputs. In practice, recursively introduced superfluous variables and environments can compromise programs that otherwise run in constant space.

Environments and heaps

An interesting problem which we have not explored yet, is the representation of heaps. Intuitively, heaps take the appearance of a data structure that resembles *hash tables* very much.

If one does not care much for space usage and immutability, arrays can be used for heap representations to provide constant time access and updates. Notice that in traditional data structures, one cannot ask "the rest of the world" to treat a name or key as a different one. Fortunately, in the case of picking fresh variables, we can exploit the property of determining names. Let the heap be represented by an array A, where each cell in the array contains a tuple of an expression and it's paired environment. When the machine begins, an array A of size n and a stack S_A which contains the numbers $0, 1 \ldots n$, are initialized. When a let expression introduces a new name `x` with the expression `e` under the environment `E` to the heap, a fresh number k is popped from S_A such that $A_k = \texttt{(e, E)}$ and $\texttt{E'} = \texttt{E} \cup \{\texttt{x} \mapsto k\}$. Now garbage collection becomes a matter of simply refilling S_A with unreachable indices of A.

Unfortunately, an immutable heap can become of importance once more exotic features are considered, such as section 4.4.3. Fortunately, very efficient persistent hash map implementations exist [Bag01].

[Ses97] introduces *environment trimming* to solve space leaks. Environment trimming is the technique of maintaining the invariant that all mappings that are not free in a paired expression should be "trimmed away". The simplest implementation of environment trimming is a trimmer that eagerly trims the environment such that when an environment `E'` is bound to an expression `e` under some current environment `E`, then Equation 4.32

must hold.

$$\texttt{E'} = \{\texttt{x} \mapsto \texttt{y} \mid (\texttt{x} \mapsto \texttt{y}) \in \texttt{E} \wedge \texttt{x} \in \mathit{free}(\texttt{e})\} \tag{4.32}$$

Equation 4.32 requires runtime introspection of the expression `e`, thus still breaks suspension. Fortunately, computing the free variables of an expression which has not been renamed can be performed during compilation, thus the trimmer should now be assumed to co-exist with every expression.

Remark 4.4.4. When trimmers occur with expressions they are written as a tuple `(e, t)` where `e` is the expression and `t` is the trimmer. In certain circumstances, the tuple may be written as a single variable such as `y`, such that $\texttt{y} \equiv \texttt{(e, t)}$, which will ease readability in some cases.

A parallel garbage collector

In [Lau93] it is mentioned that a garbage collection strategy that implements parallelism is possible. In immutable environments, parallelism becomes trivial since concurrent writing cannot occur. Naturally, if we keep the heaps and various environments immutable, parallel garbage collection becomes feasible.

The CPS machine must be augmented with the stack `N`, that contains expressions that mustn't be garbage collected (Figure 4.12). The stack `N` will later be introduced into the system explicitly, but for now, it remains implicit.

Remark 4.4.5. The stack `N` is exactly $ap(S)$ in the stack machine.

The heap Γ will now be split into two heaps, such that Γ becomes a tuple of a live heap Γ_L and an idle heap Γ_I, $\Gamma = (\Gamma_L, \Gamma_I)$. The live heap Γ_L, is the only heap that any case of let or var may update. The idle heap Γ_I will act as the heap that the garbage collector may work on, but may still be searched in.

When garbage collection begins, the heaps are swapped, such that $\Gamma = (\Gamma_I, \Gamma_L)$, which is denoted `swap(`$\Gamma$`)`, e.g. `swap` can occur between any two instructions. Variables can occur in both Γ_L and Γ_I, which can simply be proven by the sequence of instructions in Equation 4.33.

$$\begin{aligned}
& \texttt{eval((}\Gamma_L\texttt{, }\Gamma_I\texttt{), S, let x = e in p)} \\
\Rightarrow\ & \texttt{eval((}\Gamma_L \cup \{\gamma \mapsto \texttt{e}\}\texttt{, }\Gamma_I\texttt{), S, subst(x, }\gamma\texttt{, p))} \\
\Rightarrow^*\ & \texttt{eval(swap(}\Gamma\texttt{)} \subseteq \texttt{(}\Theta_L\texttt{ , }\Theta_I \cup \{\gamma \mapsto \texttt{e}\}\texttt{), S', }\gamma\texttt{)} \\
\Rightarrow\ & \texttt{eval((}\Theta_L\texttt{, }\Theta_I\texttt{), cont :S', e)} \\
\Rightarrow^*\ & \texttt{continue((}\Sigma_L\texttt{, }\Sigma_I\texttt{), cont :S'', l)} \\
\Rightarrow\ & \texttt{continue((}\Sigma_L \cup \{\gamma \mapsto \texttt{l}\}\texttt{, }\Sigma_I\ \{\gamma \mapsto \texttt{e}\}\texttt{), S'', l)}
\end{aligned} \tag{4.33}$$

Note that the environments have been omitted for readability, they should exist, but are not interesting in this segment. In the evaluation in Equation 4.33 a case of ambiguity occurs; is $\{\gamma \mapsto \texttt{e}\}$ or $\{\gamma \mapsto \texttt{l}\}$ the newest mapping? The solution is not simply a case of consulting the live heap, since another instance of swap can induce that the idle heap now contains the "newest" mapping. To track the newest variable, we employ the works of [Lam78]. Timestamps are logical numbers assigned to events, such that we can impose an ordering on when events occur. In more contextual terms, we assign a unique timestamp to every expression. If an expression $\texttt{e}_1$ with timestamp $\texttt{T}_1$ is created before an event $\texttt{e}_2$ with timestamp $\texttt{T}_2$, then $\texttt{T}_1 < \texttt{T}_2$. The implementation is simple, we extend the tuple of the heap with increasing lamport timestamps T_k such that a heap becomes a mapping from variables to triples as depicted in Equation 4.34.

$$\Gamma = \{\texttt{x}_1 \mapsto (\texttt{e}_1, \texttt{E}_1, \texttt{T}_1), \texttt{x}_2 \mapsto (\texttt{e}_2, \texttt{E}_2, \texttt{T}_2) \ldots \texttt{x}_n \mapsto (\texttt{e}_n, \texttt{E}_n, \texttt{T}_n)\} \tag{4.34}$$

When a case of let or var occurs, the expression is also paired with a timestamp. Now the newest entry of $\{\gamma \mapsto (\texttt{e, E, T})\}$ and $\{\gamma \mapsto (\texttt{l, E', T'})\}$ must be the one that contains the newest timestamp such that either $\texttt{T} > \texttt{T'}$ or $\texttt{T'} > \texttt{T}$. The function `timestamp` generates a new timestamp that is greater than all previous timestamps.

With the introduction of two heaps, environments and trimmers, the algorithm presented in Equation 4.18 is no longer adequate. Fortunately, trimmers are very helpful in simplifying the garbage collection algorithm. The garbage collection algorithm in Figure 4.19 makes use of the trimmers in the environments. The garbage collection in Figure 4.19 prunes all values in Γ_I that either occur in Γ_L with a newer timestamp, or do not occur free in `N`. Once the garbage collection algorithm in Figure 4.19 has completed in parallel, the resulting heap Θ where $\Theta \subseteq \Gamma_I$, becomes the new idle heap.

Remark 4.4.6. The parallel garbage collection algorithm in Figure 4.19 introduces complicated behaviour to maintain immutability in the runtime environment. If one relaxes the constraint of immutability, the horizon widens; A derivative of the persistent hash map data structure [Bag01], ensures efficient and concurrent operations [Pro+12], if one wishes to simplify garbage collection, whilst maintaining the potential of parallelism.

4.4.4 The enhanced CPS machine

Now every ingredient has been prepared for the practical CPS machine to take shape. The type of continuations is lifted such that timestamps, environments, and pending expressions are introduced $\texttt{cont}:\Gamma \times S \times E \times N \times \lambda \rightarrow \lambda$. Trimmers have been omitted for readability in the enhanced CPS machine in Figure 4.20. The machine in Figure 4.20 is daunting, but is only

$$\texttt{trim(E, t)} = \{\gamma \mid \texttt{x} \in \texttt{t} \wedge (\texttt{x} \mapsto \gamma) \in \texttt{E}\}$$

$$\texttt{new}(\Gamma, \{\}) = \{\}$$

$$\texttt{new}(\{\texttt{x} \mapsto (\texttt{e'}, \texttt{E'}, \texttt{T'})\} \cup \Gamma,$$

$$\{\texttt{x} \mapsto (\texttt{e}, \texttt{E}, \texttt{T})\} \cup \Theta) = \texttt{new}(\Gamma, \Theta) \cup \begin{cases} \{\} & \texttt{if T'} > \texttt{T}, \\ \{\texttt{x} \mapsto (\texttt{e}, \texttt{E}, \texttt{T})\} & \end{cases}$$

$$\texttt{new}(\Gamma, \{\texttt{x} \mapsto (\texttt{e}, \texttt{E}, \texttt{T})\} \cup \Theta) =$$

$$\{\texttt{x} \mapsto (\texttt{e}, \texttt{E}, \texttt{T})\} \cup \texttt{new}(\Gamma, \Theta)$$

$$\texttt{Prune}(\Gamma, \{\}) = \{\}$$

$$\texttt{Prune}((\Gamma_L, \Gamma_I), \{((\texttt{e}, \texttt{t}), \texttt{E})\} \cup \texttt{N}) = \texttt{I}_n \cup \texttt{Prune}((\Gamma_L \setminus \texttt{K}, \Gamma_I \setminus \texttt{K}), \texttt{R} \cup \texttt{N})$$

$$\texttt{where A} = \texttt{trim(E, t)},$$

$$\texttt{L} = \{\texttt{x} \mapsto \texttt{y} \mid \texttt{x} \in \texttt{A} \wedge (\texttt{x} \mapsto \texttt{y}) \in \Gamma_L\},$$

$$\texttt{I} = \{\texttt{x} \mapsto \texttt{y} \mid \texttt{x} \in \texttt{A} \wedge (\texttt{x} \mapsto \texttt{y}) \in \Gamma_I\},$$

$$\texttt{L}_n = \texttt{new(I, L)},$$

$$\texttt{I}_n = \texttt{new(L, I)},$$

$$\texttt{K} = \texttt{L}_n \cup \texttt{I}_n,$$

$$\texttt{R} = \{((\texttt{e}, \texttt{t}), \texttt{E}) \mid \texttt{x} \mapsto ((\texttt{e}, \texttt{t}), \texttt{E}, \texttt{T}) \in \texttt{K}\}$$

Figure 4.19: Garbage collection algorithm, note that all pointed to values in any Γ are in the from $\{((\texttt{e}, \texttt{t}), \texttt{E}, \texttt{T})\}$. Unpacking a triple `((e, t), E, T)` can be omitted by naming the triple `y`.

```
                              eval(Γ, S, E, N, λx.e) = continue(Γ, S, E, N, λx.e)

                    eval(Γ, S, {p ↦ z} ∪ E, N, l p) = eval(Γ, cont :S, E, (p, E) :N, l)
                                                      where cont(Σ, S', E', _ :N', λx.e) =
                                                             eval(Σ, S', E' ∪ {x ↦ z}, N', e)

eval(Γ ∪ {γ ↦ (y, E', T)}, S, {x ↦ γ} ∪ E, N, x) = eval(Γ, cont :S, E', N, e)
                                                      where cont(Σ, S', E'', N', p) =
                                                             T = timestamp,
                                                             Θ = Σ ∪ {x ↦ (p, E'', T)},
                                                             continue(Θ, S', E'', N', p)

                    eval(Γ, S, E, N, let x = e in p) = eval(Θ, S, E', N, p)
                                                      where γ = fresh,
                                                            E' = E ∪ {x ↦ γ},
                                                            T = timestamp,
                                                            Θ = Γ ∪ {γ ↦ (e, E', T)}

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

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

Figure 4.20: The CPS machine with environments, timestamps and pending expressions.

a product of the application of the aforementioned enhancements onto the previously explored CPS machine (Figure 4.18).

Naturally, now that the machine has reached fruition, a test of practicality finds it's way into this work. Two programs (Listing 7.2 and Listing 7.3) have been implemented to test the CPS machine with various garbage collection strategies and the effect of trimmers and environments. The result of various observations can be seen in Figure 4.21, where P is the probability that any garbage collection operations are performed on a reduction step.

Interpreting the garbage collection data

The data in Figure 4.21 show that the parallel garbage collector in practice, performs quite a bit better than the synchronous one in terms of speed. The parallel garbage collector, in fact, performs just as well as no garbage collector. The parallel garbage collector suffers from an apparent consequence, which is that it potentially only collects half of the garbage on each collection cycle. The data for heap usage displays this well; When the garbage collector runs on the idle heap, then the live heap keeps accumulating new references in parallel, which in some cases causes the live heap to accumulate more references that was otherwise collected. The heap usage and wall clock running time of a synchronous versus a parallel displays the age old trade of memory versus performance.

4.4.5 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 generally becomes a bit more tricky when considering interpreter details. Fortunately, we are get a property, often named *tali call optimizaiton*, for free, when implementing lambda calculus evaluation machines. A prerequisite for an infinitely running program to run in finite space 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 4.4.7. 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

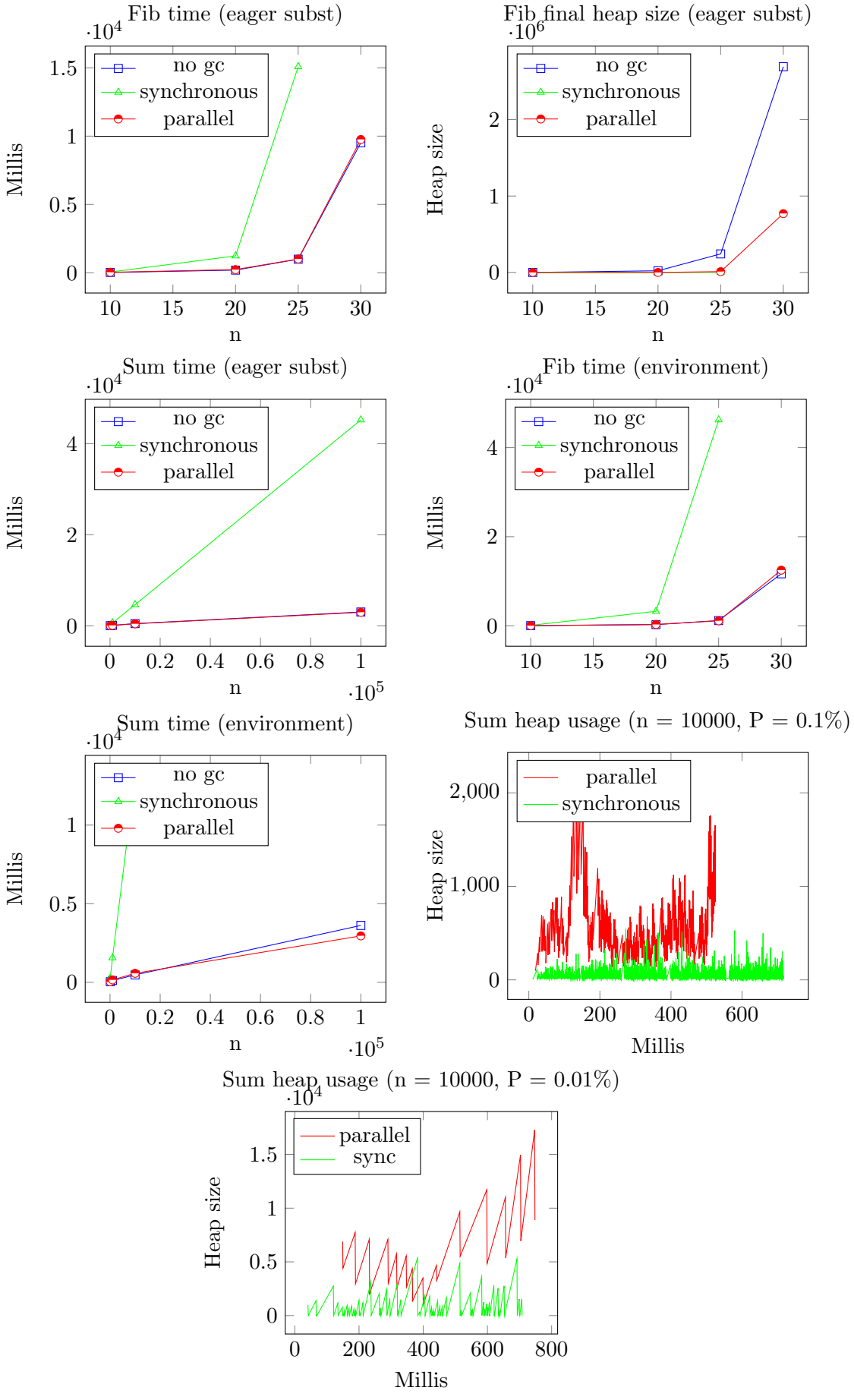

Fib time (eager subst)
·10^4
Millis
n
no gc
synchronous
parallel
Fib final heap size (eager subst)
·10^6
Heap size
Sum time (eager subst)
Fib time (environment)
Sum time (environment)
Sum heap usage (n = 10000, P = 0.1%)
parallel
synchronous
Millis
Sum heap usage (n = 10000, P = 0.01%)
sync

Figure 4.21

is called, the called-from function places a "resume execution from here" pointer onto the call stack (like the CPS machine).

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 some imperative languages, recursive functions can only iterate a finite number of times, more specifically until the call stack is full.

To really understand what happens in a lambda calculus interpreter, we can observe what happens in Listing 7.5 and Listing 7.4. The two implemented variants of a `fold` function accumulate a list of type `List a` to a `b`. The two variants differ when considering evaluation strategy and tail call optimization.

Remark 4.4.8. 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)`, which increases the size of the program by `x +`, since the `+` operator requires both expressions to be evaluated. Reduction strategies always imply tail call optimization, whenever possible.

The first flavor of `fold`; `foldl`, implements `fold` such that the program expression tree does grow throughout program interpretation.

$$
\begin{aligned}
&\texttt{foldl add l (Cons 1 (Cons 2 ... (Cons n Nil)))} && (4.35)\\
&= \texttt{l z (}\lambda\texttt{xs,x.foldl f (f x z) xs)}\\
&= \texttt{(Cons 1 (Cons 2 ... (Cons n Nil))) z (}\lambda\texttt{xs.}\lambda\texttt{x.foldl f (f x z) xs)}\\
&= \texttt{foldl f (f x z) xs \{ xs} \mapsto \texttt{ (Cons 2 (Cons 3 ... (Cons n Nil))), x} \mapsto \texttt{ 1, ... \}}\\
&= \texttt{foldl add (add 1 0) (Cons 2 (Cons 3 ... (Cons n Nil))) \{...\}}\\
&\ \ \ldots
\end{aligned}
$$

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

`foldr` on the other hand, does not necessarily increase in heap size, since the invocation of `f` immediately forces the evaluation of `x` and pushes `x` to the continuation stack by the CPS machine. Clearly the stack of continuations must be where the payment of space is made.

Remark 4.4.9. In practice, `foldr` is very convenient since it can be lazy in it's second argument. For instance, early stopping when performing a linear search, does not require evaluating the rest of the list.

By running the two programs, we can observe something interesting regarding infinite programs in general (Figure 4.22). `foldl` runs slightly faster than `foldr`, on smaller inputs. When garbage collection and space concerns become involved, `foldl` creates a giant un-collectable expression, while `foldr` remains in the territory of constant space. `foldl` passes an accumulator to the next recursive invocation such that the expression of the accumulator becomes `(n + ((n - 1) + ((n - 2) + ... + 0)))`.

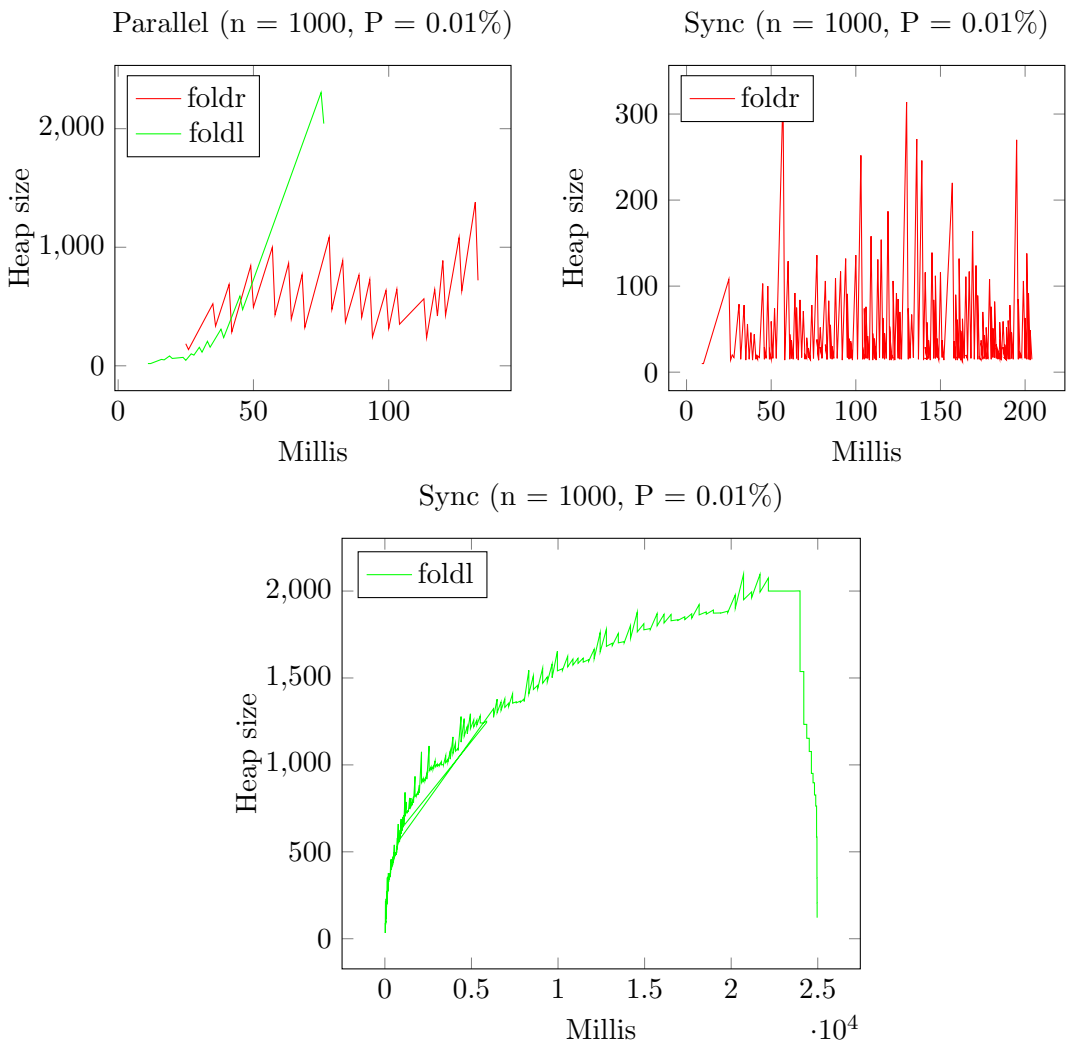

Parallel (n = 1000, P = 0.01%)
foldr
foldl
Heap size
2,000
1,000
0
0
50
100
Millis
Sync (n = 1000, P = 0.01%)
foldr
Heap size
300
200
100
0
0
50
100
150
200
Millis
Sync (n = 1000, P = 0.01%)
foldl
Heap size
2,000
1,500
1,000
500
0
0
0.5
1
1.5
2
2.5
Millis
·10^4

Figure 4.22

Chapter 5

Practical data structures

Throughout the previous sections we have used abstract representations for various collections such as environments in typing (section 3.1) and heaps in reduction strategies (section 4.4). To map the abstract representations to practical algorithms, we must materialize various collections as data structures. Fortunately, the field of data structures and in particular, persistent data structures, is well researched thus it becomes a matter of picking a good fit. Data structures, like evaluation strategies (section 4.1), come in both lazy and eager variants, of which we will interest ourselves with eager variants. All data structures are to be considered subject to eager evaluation.

This section should act like an appendix which introduces the data structures used during the practicalities of this work; implementing the algorithms.

Listing 5.1: Stack implementation

```
1  type Maybe a =
2    | Nothing
3    | Just a
4  ;
5  type TupleAttr a b = Tuple a b;
6  type List a =
7    | Nil
8    | Cons a (List a)
9  ;
10 type StackAttributes a = Stack (Int) (List a);
11
12 fun push s x =
13   match s
14     | Stack n xs -> Stack (n + 1) (Cons x xs);
15   ;
16
17 fun pop s =
18   match s
19     | Stack n xs ->
20       match xs
21         | Nil -> Tuple Nothing s;
22         | Cons x rest ->
23           Tuple (Just x) (Stack (n - 1) rest);
24       ;
25   ;
```

5.1 Lists and stacks

The first and least interesting data structure we will explore, is the stack. The persistent stack in functional programming languages naturally comes from the `List` algebraic data structure. In imperative programming languages a stack can be implemented via a linked list, which is similar to a `List`. Listing 5.1 is an implementation of a stack similar to how stacks can be implemented in imperative languages. The `pop` and `push` functions clearly have worst case constant time since neither of them perform any operations that depend on any input size. A stack that can `pop` and `push` is all that is necessary in the previous usages of the stack data structure.

Remark 5.1.1. The stack in Listing 5.1 can be refined further, allowing worst case constant time `multipush` operations by treating the internal `List a` as a `List (List a)`, but this is not necessary in this case.

5.2 Tables, hashes, colors and tries

A mapping of variables to some other domain are a very common occurrence throughout the previous sections. Hash tables are data structures that provide very good probabilistic bounds for such mappings. Unfortunately, hash tables build upon mutability, thus do not provide the property of persistence that we require. Fortunately, some data structures are better fitted for persistence than others.

In this section we will explore two types persistent types of associative data structures, the red-black tree and the hash array mapped trie.

5.2.1 Red-black trees

Trees are very common data-structures because of their inherit guarantees. A *nodes* is a containers of some key and value, as depicted in Figure 5.1a. The key of a node is it's identity, such that one can query the tree for the associated value. Nodes occur in trees such as in Figure 5.1b, where the *root* of the tree is the topmost node (the root of Figure 5.1b is 2). A node can have children, which are the nodes directly below it. A node with no children, is called a leaf. A sub-tree is the tree that is formed by letting a node that is otherwise a child, be a root. A parent of a node c is the node p that has c as a child. A grandparent of a node is the node's parent's parent. A path p from the root l to o is a sequence of nodes $l \to \cdots \to o$, such that the length of p, denoted $\#p$, is the number of nodes that occur in the path. There are various orderings one can impose on a tree, but we will maintain the invariants that all sub-trees left of a node will have keys that are smaller than the key of that node and all sub-trees right of a node will have keys that are larger than the key of that node. A balanced tree, is tree of size n where no one node is further than $\lceil \log_2 n \rceil$ from the root. Trees can be a very useful representation of stored data, since finding a value in a balanced tree requires one to visit at most $\lceil \log_2 n \rceil$ nodes since that is the furthest (the number of nodes between) a node can be from the root. Searching for a key k in a tree which maintains the previously established ordering invariant, is a matter of beginning at the root and determining if the root's key r has the same key, if not then the left tree is visited if $r > k$ else the right tree is visited, this process is repeated until a node with a key equal to k is found.

Trees by themselves are very open for interpretation, since they specify nothing of how keys are inserted and deleted. An very popular variant, which we will interest ourselves in is the red-black tree [Bay72]. A variant of the red-black tree which we will interest ourselves in is the Okasaki variant [Oka99]. In a red-black tree, every leaf node is a node with no key or value, called an empty node. Furthermore, in a red-black tree every node is enhanced with a color, which is either black or red, hence the name. A

(a) A node (b) A tree

Figure 5.1

red-black tree has two invariants that must be upheld.

- No red node has a red parent.
- Every path from the root to an empty node contains the same number of black nodes.

Maintaining these two invariants, the worst case for searching becomes worse than a balanced tree, but only by a constant factor.

Lemma 5.2.1. *A red-black tree of size n has height at most $2\log_2(n+1)$. [Cor+09].*

Since $O(2\log_2(n+1)) = O(\log_2 n)$ then search can be performed in $O(\log_2 n)$ by the same algorithm as a balanced binary tree.

Insertion

Insertion is the first practical problem we will tackle in the red-black tree. In [Oka99] presents an elegant algorithm for insertion. Observe that by inserting a node naively, one might violate one of the two invariants. When we insert a node, we perform an act of *rebalancing*. Rebalancing is an action that guarantees that the tree will maintain the two aforementioned invariants after action has occurred on the tree. When inserting a node, the node must colored red and always be placed in the bottom of the tree by searching for a position, while maintaining the ordering of the tree. When the node has been placed, rebalancing is performed on the way back up such that violations of the two invariants, are floated up throughout the tree until the root is encountered, and then the root is colored black. Consider that inserting a node at the leaf, may only cause a violation if the parent node of the one inserted is also red; if the parent node is black, then the invariants hold. When considering insertion (and rebalancing), the property that the red-black tree was in a state that upheld the invariants is always maintained. There are two types of *rotations* (and two reflections for each rotation) we must perform to balance the tree after a violation. Rotations are the act of moving nodes around such that they maintain the invariants. Rotations are considered only for nodes that have grandparents (or conversely, nodes

that have grandchildren). For a node that has grandchildren there are four cases that may violate the invariant (again, two reflections for each rotation), which is depicted in Figure 5.2. These rotations satisfy the second invariant, since for all rotations the same number of black nodes are present above `a, b, c` and `d`. Furthermore, notice that the red node in Figure 5.2c may have a red parent, this violation is only temporary, since the tree is rebalanced in reverse order of the search path. The insertion algorithm can elegantly be encoded into the programming language, given support for nested pattern matching (subsection 2.4.2). An implementation (give that the language supports nested pattern matching) has been presented in Listing 7.6.

The number of nodes to visit when searching for a position to place the new node is at most $O(\log_2 n)$ time. The time for one `balance` is at most 1 invocation, the `balance` function is invoked for each travelled node thus it is invoked $O(\log_2 n)$ times, which implies that the combined insertion time is $O(\log_2 n)$.

Deletion

Deletion in red-black trees is a more complicated than insertion. Deletion of a black node can imply a much larger range of rotations than insert otherwise would. In [GM14], an algorithm for deletion is presented. The algorithm, inspired by the Okasaki red-black tree, solves deletion in a similar method as insertion. When a deletion deletes a black node, then the tree will no longer satisfy the black node invariant. The black node invariant is then restored by introducing a double black node. After this double black node has been introduced, a recursive algorithm, similar to the `balance` algorithm, removes all double black nodes by performing rotations. The rotations implied by deletion can be further investigated in [GM14].

Instead of discussing the various rotations, we will move on to a another data structure that will supersede the red-black tree.

5.2.2 Hash array mapped trie

The red-black tree is a staple data structure in the competent programmers tool belt, because of their $O(\log_2 n)$ bound and simple implementation details. In a persistent setting, Okasaki showed that red-black tree's can be implemented elegantly. In a ephemeral setting, lookup oriented data structures can have search bounds of average case constant time (hash tables) [Cor+09]. A suitable candidate which lifts the ephemeral hash table to a persistent setting is Hash Array Mapped Trie (HAMT) [Bag01].

The HAMT is a tree, where all nodes have a constant amount of children, in the practical case, 32 is the number of children.

Remark 5.2.1. 32 seems like an odd number too pick, but has it's practicalities. Bitsets will shortly be introduced to the data structure. A bitset of

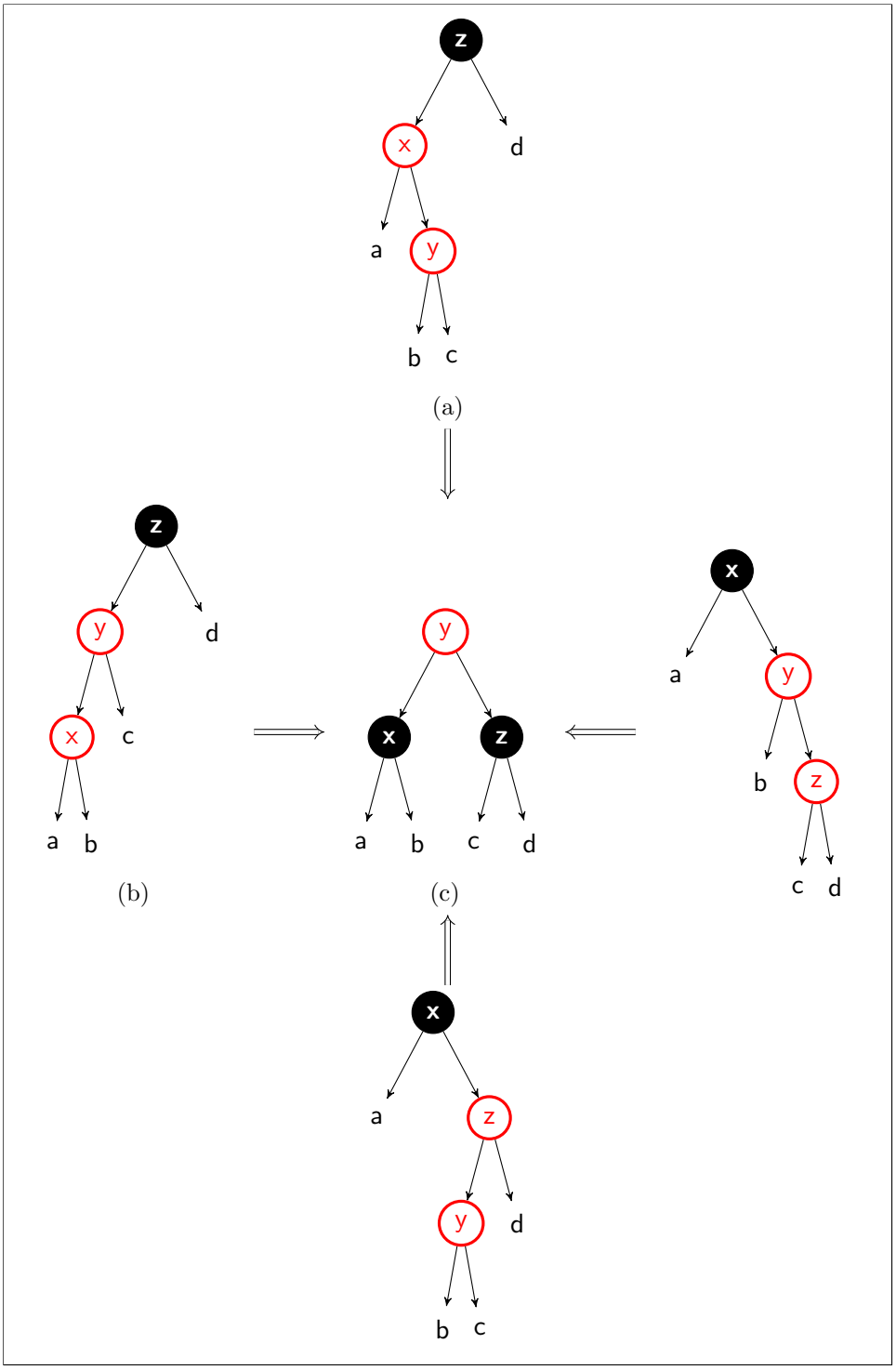


Figure 5.2: Red-black tree rotations

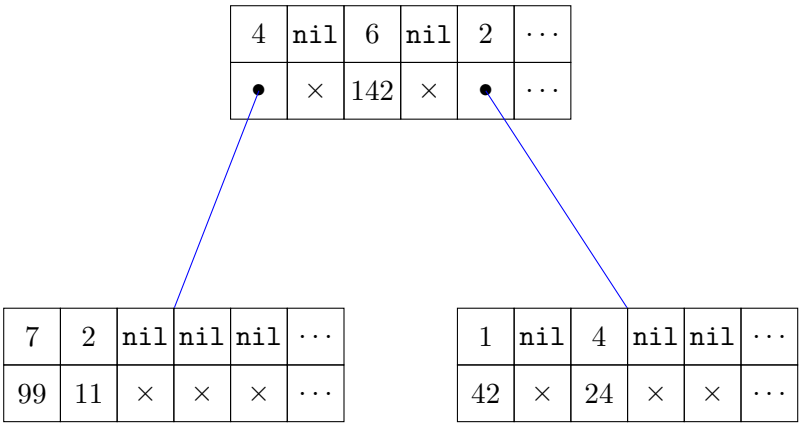


Figure 5.3: Prefix tree

size 32 is exactly a 32 bit word.

A child can either be a terminal value (a key and associated value), or another node.

An enhanced prefix tree

In an introductory data structure to the HAMT, one can let all nodes have a *child array* of size 32, where all entries are `nil` initially. Figure 5.3 is a *prefix tree* that is a prerequisite for the HAMT. A prefix of a number is the first digit. Searching for a number such as 42 will search the first node for the prefix of 42, 4, such that the left child is chosen and the new search number is 2. In the left child the prefix of the remaining number, 2, is chosen such that the cell that contains 11 is found.

Now we argument the prefix tree with hashes instead of digits, this change will become useful in the next iteration of the data structure. A hash function is randomly chosen for the data structure $h \in \mathcal{H}$ that maps the universe of keys into a 32 bit word $\{0, 1 \ldots, 2^{32}\}$, such that the probability of collision is $\frac{1}{2^{32}}$ [Cor+09]. When a key occurs in the 32 sized array, the five most significant bits of the hash are converted to an integer and are to index directly into the array, since $2^5 = 32$.

Example 5.2.1. The five most significant bits of Equation 5.1 are `10010`, which is the index 18.

```
11 11110 01101 11100 10000 01101 10010          (5.1)
```

Now we can efficiently index into the array.

Notice that the prefix tree contains many *sparse arrays*, such that space may potentially be many fold of the actual size of the data. To solve this, we augment the prefix tree with a bitset, such that it becomes a Hash Array

Mapped Trie. Each node is augmented with a bitset of size 32, such that each bit indicates whether a value is present or not. Furthermore, a processor instruction exists (`CTPOP`) that counts the number of set (1's) bits in a bitset. To enhance `CTPOP`, we count the number of set bits until an offset (this can be achieved by a very simple mask and OR operation). When we have the ability to query for how many set bits come before an index in the bitset, we can represent the sparse array compactly. The array will now only contain the set values and the number of set bits before an index in the bitset will be the offset in the array.

Searching and inserting

Now the structure of the HAMT has been completed. Search is performed by combining the aforementioned enhancements:

1. Compute the hash $h(k)$ of key k.
2. If position $h(k)$ of the bitset is empty, return `nil`.
3. Index into the array at the position given by the number of set bits before $h(k)$.
4. If the element found is a value then compare k and the stored key, if they match then return the value, if not the return `nil`.
5. If the element found is a node then remove five bits from k and goto item 2.

Worst case search is clearly bounded by the depth, with a branching factor of 32 the search time becomes $\log_{32} n$, or $O(\log n)$.

Insertion is performed by a simple set of steps:

1. Search until either an unused slot reveals itself or a terminal value is found.
2. If the slot is empty, insert the value and return.
3. If the slot contains a value, create a new HAMT node and insert the old value into the node, and the recurse into the node.
4. If the hash has been exhausted while solving collision, rehash the key k by using a new hash function.

Remark 5.2.2. Collisions can occur, but they can be done away with traditionally. For instance, when collisions occur at the leaf level ($\lfloor \frac{32}{5} \rfloor$) then one can pick a particular hash function for that leaf which local uniqueness, and then add a collision node.

Lazy root resizing

Inevitably, when the tree grows, the probability of a hash not colliding decreases, which prompts for a resizing algorithm. So far the root has been introduced as if it has the same properties as nodes. The root will no longer have a bitset and initially be an array of `nil`. Since the root is only the top node, then the extra space used by the array is but a constant factor of the total size. When the HAMT reaches a point where it's size exceeds some threshold it must begin `lazy resizing`. Lazy resizing is the act of increasing the size of the root and incrementally moving entries from the old root over. Let 2^t be the number of bit's that the root can consume, where $t = 5$ initially, if we increase this number by five, such that $2^(t + 5)$, then the new root can contain all entries of all of the root's children since $(2^5)^2 = 2^{10}$. Effectively, the children of the root are moved into the root once the HAMT's size becomes too large, such that the height is reduced by one. The operation of moving the children into the new root can be amortized over searches and inserts.

Chapter 6

Conclusion

In chapter 2 we introduced the source language and presented strategies for conversion from high-level languages to the untyped lambda calculus. We then introduced types, polymorphism, how to implement a decidable type system and the apparent cost of type systems in section 3.1, after which we then gave a brief introduction of the map of type systems. When a program has been translated and proven, we then sought to determine what the program would evaluate to, by considering various evaluation strategies and machines chapter 4. Finally, we gave a brief overview in chapter 5 of the data structures used to implement the algorithms presented in this work.

6.1 Functional programming languages

Functional programming has throughout time been present in academic circles more often than the industry, whilst imperative has had a big presence in the industry. As of the time of this thesis, functional programming has been on the rise in mainstream languages. Various imperative programming languages have introduced concepts, that had otherwise only been found in functional programming languages. As time progresses, the trend points towards a unification of what previously was considered orthogonal styles, functional and imperative.

Bibliography

[Bag01] Phil Bagwell. *Ideal hash trees*. Tech. rep. 2001.

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

[Bay72] Rudolf Bayer. "Symmetric binary B-trees: Data structure and maintenance algorithms". In: *Acta informatica* 1.4 (1972), pp. 290–306.

[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).

[Cor+09] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[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.

[GM14] Kimball Germane and Matthew Might. "Deletion: The curse of the red-black tree". In: *Journal of Functional Programming* 24.4 (2014), pp. 423–433.

[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.

[Lam78] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

[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.

[Oka99] Chris Okasaki. “Red-black trees in a functional setting”. In: *Journal of functional programming* 9.4 (1999), pp. 471–477.

[Pro+12] Aleksandar Prokopec et al. “Concurrent tries with efficient non-blocking snapshots”. In: *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 2012, pp. 151–160.

[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.

[Wad15] Philip Wadler. “Propositions as types”. In: *Communications of the ACM* 58.12 (2015), pp. 75–84.

[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.

Chapter 7

Appendix

Listing 7.1: The output of an exponential type

```
1  ########## tuple ##########
2  substitution set Map(c0 -> (a0 -> (b0 -> d0)))
3  type (a0 -> (b0 -> ((a0 -> (b0 -> d0)) -> d0)))
4  Type vars in sub = 4
5  ########## tuple ##########
6  ########## one ##########
7  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)) ->
8  g0))
9  type (((h0 -> (i0 -> ((h0 -> (i0 -> j0)) -> j0))) ->
       ((k0 -> (l0 -> ((k0 -> (l0 -> m0)) -> m0))) ->
     g0)) -> g0)
10 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)))
11 Type vars = 14
12 Type vars in sub = 33
13 ########## one ##########
14 ########## two ##########
15 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 ->
16  ((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 ->
17  ((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))
18 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)
19 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 -
20 > 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 ->
21  (c1 -> d1)) -> d1))) -> z0)) -> z0) -> q0)) -> q0))
    )
22 Type vars = 32
23 Type vars in sub = 94
24 ########## two ##########
25 ########## three ##########
26 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 -> (
```

```
53 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 -> (
54 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 ->
55 (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))
56 -> 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))
57 -> 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))
58  -> s0) -> ((((a1 -> (b1 -> ((a1 -> (b1 -> e1)) ->
       e1))) -> ((y0 -> (c1 -> ((y0 -> (c1 -> d1)) ->
       d1))) -> z0)) -> z0) -> q0)) -> q0))
59 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
60 -> (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)
96 -> 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)
97 -> ((((((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) ->
98  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))) -
99 > 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) ->
100 ((((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))
101 -> 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
102 ),(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 && (7.1)\\
=&\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}
$$

(7.2)

Listing 7.2: A program that sums a list of 100000 increasing integers

```
1  type List a =
2    | Nil
3    | Cons a (List a)
4  ;
5
6  fun sum l =
7    match l
8      | Cons x xs ->
9        x + (sum xs);
10     | Nil -> 0;
11   ;
12
13 fun range n s =
14   if (n == s)
15     Nil;
16   else
17     (Cons s (range n (s + 1)));
18   ;
19
20 fun inf from =
21   Cons from (inf (from + 1));
22
23 fun take l n =
24   if (n == 0)
25     Nil;
26   else
27     match l
28       | Nil -> Nil;
29       | Cons x xs -> Cons x (take xs (n - 1));
30     ;
31   ;
32
33 fun main =
34   let garbage = range 100 0;
35   let longlist = inf 1;
36   sum (take longlist 100000);
```

Listing 7.3: A program which calculates the Fibonacci numbers recursively

```
1  fun fib n =
2    if (n == 0)
3      0;
4    else
5      if (n == 1)
6        1;
7      else
8        (fib (n - 1)) + (fib (n - 2));
9      ;
10   ;
11
12 fun main = (fib 30);
```

Listing 7.4: foldl

```
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 range n s =
  if (n == s)
    Nil;
  else
    (Cons s (range n (s + 1)));
  ;

fun inf from =
  Cons from (inf (from + 1));

fun take l n =
  if (n == 0)
    Nil;
  else
    match l
      | Nil -> Nil;
      | Cons x xs -> Cons x (take xs (n - 1));
    ;
  ;

fun main =
  let longlist = inf 1;
  foldl add 0 (take longlist 1000);
```

Listing 7.5: foldr

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

fun add a b = a + b;

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

fun range n s =
  if (n == s)
    Nil;
  else
    (Cons s (range n (s + 1)));
  ;

fun inf from =
  Cons from (inf (from + 1));

fun take l n =
  if (n == 0)
    Nil;
  else
    match l
      | Nil -> Nil;
      | Cons x xs -> Cons x (take xs (n - 1));
    ;
  ;

fun main =
  let longlist = inf 1;
  foldr add 0 (take longlist 1000);
```

Listing 7.6: Red-black tree implementation

```
type Color =
  | R
  | B
;
type Tree =
  | E
  | T (Color) (Tree (Int)) (Int) (Tree (Int))
;
type Boolean =
  | False
  | True
;
fun member x t =
  match t
    | E -> False;
    | T c a y b ->
      if (x < y)
        member x a;
      else
        if (x == y)
          True;
        else
          member x b;
        ;
      ;
  ;
fun makeBlack t =
  match t
    | T c a y b -> T B a y b;
  ;
fun ins x s =
  match s
    | E -> T R E x E;
    | T color a y b ->
      if (x < y)
        balance c (ins a) y b;
      else
        if (x == y)
          T color a y b;
        else
          balance color a y (ins b);
        ;
```

```
43        ;
44    ;
45
46  fun insert x s = makeBlack (ins x s);
47
48  fun balance c a x b =
49    match (T c a x b)
50      | T B (T R (T R a x b) y c) z d ->
51        T R (T B a x b) y (T B c z d);
52      | T B (T R a x (T R b y c)) z d ->
53        T R (T B a x b) y (T B c z d);
54      | T B a x (T R (T R b y c) z d) ->
55        T R (T B a x b) y (T B c z d);
56      | T B a x (T R b y (T R c z d)) ->
57        T R (T B a x b) y (T B c z d);
58      | T color a x b -> T color a x b;
59    ;
```