Type Systems for Programming Languages

Course Notes

Steffen van Bakel
Department of Computing
Imperial College London

Spring 2001
(revised Autumn 2011)

Preface

These notes belong to the course Type Systems for Programming Languages, given to third and fourth year students in Computing and Joint Mathematics and Computing with some experience in reasoning and logic, and students in the Advanced Masters programme at the Department of Computing, Imperial College London.

The course is intended for students interested in theoretical computer science, who possess some knowledge of logic. No prior knowledge on type systems or proof techniques is assumed, other than being familiar with the principle of structural induction.

Aims

- To lay out in detail the design of type assignment systems for programming languages.
- To focus on the importance of a sound theoretical framework, in order to be able to reason about properties of a typed program.
- To understand the concepts of: type checking, type reconstruction, polymorphism, type derivation, typeability, typing of recursive functions, termination in the context of typeability, and undecidable systems.
- To study various systems and various languages, and to compare those and to select.
- To formulate the criteria that a desired systems should satisfy.

Objectives

- The student will understand the difference between *fully typed* languages and *untyped* languages that come with *type assignment*.
- The student will be able to *derive* a type for a given program, and *check* if a given type is correct for a given program.
- The student will know what the main properties of type assignment systems are, and what are the techniques used in proving these properties; the student will be able to reproduce such a proof.
- The student will be able to apply the properties in specific cases to characterise reduction behaviour of a given program.

Contents

Introduction

Adding type information to a program is important for several reasons. Using type assignment it is possible to build an *abstract interpretation* of programs by viewing terms as objects with input and output, and to abstract from the actual values those can have by looking only to what kind (type) they belong.

Type information makes a program more *readable* because it gives a human being additional, abstracted so less detailed information about the structure of a program. Furthermore, type information plays an essential role in the implementation, during code generation: the information is needed to obtain an *efficient* implementation and is also makes *separate compilation* of program modules possible. However, one of the most important aspects of type systems is that they warn the programmer in an early stage (at compile time) if a program contains *severe errors*. If a program is error-free, it is safe to run: *"Typed programs cannot go wrong"* (Milner [40]). The meaning of this well-known quotation is the following: *a compile-time analysis of a program filters out errors in programs which might occur at run-time*, like applying a function defined on integers to a character. However, not all errors can be detected at compile-time: some errors will be caused by variables reaching certain values computed at run-time (e.g. divide by zero and stack overflow) and cannot be detected at compile time. So, only some *special classes of run-time errors* can be found at compile time.

Typing deals with the analysis of the domain and range on which procedures (functions) are defined and a check that these functions are indeed applied consistently; this is achieved through a compile-time approximation of its run-time behaviour, by looking at the syntactic structure of the program only. This course deals with such type assignment systems for functional languages. Because of the strong relation between the Lambda Calculus and functional programming, type assignment systems are normally studied in the context of Lambda Calculus. We will start these notes following this approach, but then focus on how to manipulate these systems in order to be able to deal with *polymorphism*, *recursion*, and see how this comes together in Milner's ML.

We will then investigate the difficulties of dealing with *patterns*, and move our attention to Term Rewriting Systems. After that, we show how to extend the system with algebraic data types and recursive types, and conclude by have a quick look at intersection types. As an example will then see how all this comes together in Haskell.

1 Lambda Calculus

The Lambda Calculus [14, 10] is a formalism, developed in the 1930's, that is equivalent to Turing machines and is an excellent platform to study computing in a formal way because of its elegance and shortness of definition. It is the calculus that lies at the basis of functional languages like Miranda [51], Haskell [32], and CaML [27]. It gets its name from the Greek character λ (lambda).

1.1 λ-terms

The set of *λ-terms*, ranged over by M, is constructed from a set of term-variables $\mathcal{V} = \{x,y,z, x_1,x_2,x_3,\ldots\}$ and two operations, *application* and *abstraction*. It is formally defined by:

Definition 1.1 (λ-TERMS) λ-terms are defined by the following grammar:

$$M ::= \underset{\textit{variable}}{x} \;\mid\; \underset{\textit{abstraction}}{(\lambda x.M)} \;\mid\; \underset{\textit{application}}{(M_1 \cdot M_2)}$$

The operation of application takes two terms M_1 and M_2, and produces a new term, the application of M_1 to M_2. You can see the term M_1 as a function, and M_2 as its operand.

The operation of abstraction takes a term-variable x and a term M and produces an abstraction term $(\lambda x.M)$. In a sense, abstraction builds a function; you can read $\lambda x.M$ as 'given the operand x, this function produces M'. This is best explained by an example: take the function f defined by $f\,x = x^2$, then we can say that $f = \lambda x.x^2$. We will write $(\lambda x.x^2)\,3$ rather than $f3$; this means functions (i.e. abstractions) in LC are *anonymous*, i.e. do not have a name like f. Since we can apply terms to functions, functions are treated like any term, and are therefore *first class citizens*, as they are in functional languages based on LC; this is not true for procedures in imperative languages.

Since '$\cdot$' is the only operation between terms, it is normally omitted, so we write (M_1M_2) rather than $(M_1 \cdot M_2)$. Also, left-most, outer-most brackets are omitted, so $M_1M_2\,(M_3M_4)$ stands for $((M_1 \cdot M_2) \cdot (M_3 \cdot M_4))$. The omitted brackets are sometimes re-introduced to avoid confusion. Also, to avoid writing many abstractions, repeated abstractions are abbreviated, so $\lambda x_1x_2x_3.M$ stands for $(\lambda x_1.(\lambda x_2.(\lambda x_3.M)))$.

Exercise 1.2 Write the following terms in the original, full notation:

i) $\lambda xyz.xzyz$.

ii) $\lambda xyz.xz\,(yz)$.

iii) $(\lambda xy.x)\,(\lambda z.yz)$.

Remove the omissible brackets in the following terms:

iv) $(\lambda x_1.(((\lambda x_2.(x_1x_2))\,x_1)\,x_3))$.

v) $((\lambda x_1.(\lambda x_2.((x_1x_2)\,x_1)))x_3)$.

vi) $((\lambda x.(\lambda y.x))(\lambda z.(za)))$.

The notion of *free* and *bound* term-variables of λ-terms is defined as follows.

Definition 1.3 (FREE AND BOUND VARIABLES) The set of *free* variables of a term M ($fv(M)$) and its *bound* variables ($bv(M)$) are defined by:

$$\begin{array}{lcl@{\qquad}lcl} fv(x) & = & \{x\} & bv(x) & = & \emptyset \\ fv(M_1M_2) & = & fv(M_1) \bigcup fv(M_2) & bv(M_1M_2) & = & bv(M_1) \bigcup bv(M_2) \\ fv(\lambda y.M) & = & fv(M) \backslash \{y\} & bv(\lambda y.M) & = & bv(M) \bigcup \{y\} \end{array}$$

Exercise 1.4 Show that, for all M, $fv(M) \cap bv(M) = \emptyset$.

On Λ the replacement of a variable x by a term N, denoted by $[N/x]$, could be defined by:

$$\begin{array}{rcl@{\qquad}l} x\,[N/x] & = & N & \\ y\,[N/x] & = & y, & \text{if } y \neq x \\ (M_1M_2)\,[N/x] & = & M_1\,[N/x]M_2\,[N/x] & \\ (\lambda y.M)\,[N/x] & = & \lambda y.M, & \text{if } y = x \\ (\lambda y.M)\,[N/x] & = & \lambda y.(M\,[N/x]), & \text{if } y \neq x \end{array}$$

Notice that, in the last alternative, problems can arise, since N can obtain free occurrences of y that would become bound during substitution; this is called a *variable capture*. To avoid these clashes, the notion of substitution needs to be defined like:

$$\begin{array}{rcll} x\,[N/x] &=& N & \\ y\,[N/x] &=& y, & \text{if } y \neq x \\ (M_1 M_2)\,[N/x] &=& M_1\,[N/x] M_2\,[N/x] & \\ (\lambda y.M)\,[N/x] &=& \lambda y.(M[N/x]), & \text{if } y \notin \mathit{fv}(N) \ \&\ y \neq x \\ (\lambda y.M)\,[N/x] &=& \lambda z.(M\,[z/y])\,[N/x], & \text{if } y \in \mathit{fv}(N) \ \&\ y \neq x, z \text{ new} \\ (\lambda y.M)\,[N/x] &=& \lambda y.M, & y = x \end{array}$$

To not have to worry about variable capture, there is a notion of equivalence (or rather, *convergence*) introduced on terms; it considers terms equivalent that can be obtained from each other by renaming bound variables as in the fifth alternative above. Essentially, this relation, called '*α-conversion*' is defined from

$$\lambda y.M \ =_\alpha \ \lambda z.(M\,[z/y])$$

extending it to all terms much in the spirit of β-conversion that we will see below. For us, it suffices to know that we can always, whenever convenient, rename the bound variables of a term. This is such a fundamental feature that normally α-conversion plays no active role; terms are considered modulo α-conversion. In fact, we will assume that bound and free variables are always different (this is called *Barendregt's convention*), and that α conversion will take place (silently) whenever necessary. Then the definition of term substitution becomes:

Definition 1.5 (Term substitution) The substitution of the term variable x by the term N is defined inductively over the structure of terms by:

$$\begin{array}{rcll} x\,[N/x] &=& N & \\ y\,[N/x] &=& y, & \text{if } y \neq x \\ (M_1 M_2)\,[N/x] &=& M_1\,[N/x] M_2\,[N/x] & \\ (\lambda y.M)\,[N/x] &=& \lambda y.(M\,[N/x]) & \end{array}$$

So, by our assumption, since y is bound in $\lambda y.M$, it is not free in N, so variable capture is impossible.

Exercise 1.6 Show that $M\,[N/x]\,[P/y] = M\,[P/y]\,[N\,[P/y]/x]$.

1.2 β-conversion

On Λ, the notion of computation is defined as a relation '$\rightarrow_\beta$' between terms that specifies that, if $M \rightarrow_\beta N$, then M executes 'in one step' to N. The idea of 'running' a term M is then to execute all steps in '$\rightarrow_\beta$' until no more steps are possible.

Definition 1.7 (β-conversion) The binary *one-step* reduction relation '$\rightarrow_\beta$' on λ-terms is defined by (we will write $M \rightarrow_\beta N$ rather than $\langle M,N \rangle \in \rightarrow_\beta$):

$$\begin{array}{l} (\lambda x.M)\,N \ \rightarrow_\beta \ M\,[N/x] \\ M \rightarrow_\beta N \ \Rightarrow \left\{ \begin{array}{rcl} PM &\rightarrow_\beta& PN \\ MP &\rightarrow_\beta& NP \\ \lambda x.M &\rightarrow_\beta& \lambda x.N \end{array} \right. \end{array}$$

The relation '$\twoheadrightarrow_\beta$' is defined as the reflexive, transitive closure of '$\rightarrow_\beta$':

$$\begin{array}{rcl} M \rightarrow_\beta N & \Rightarrow & M \twoheadrightarrow_\beta N \\ M & \twoheadrightarrow_\beta & M \\ M \twoheadrightarrow_\beta N \;\&\; N \twoheadrightarrow_\beta P & \Rightarrow & M \twoheadrightarrow_\beta P \end{array}$$

and '$=_\beta$' is the equivalence relation generated by '$\twoheadrightarrow_\beta$':

$$\begin{array}{rcl} M \twoheadrightarrow_\beta N & \Rightarrow & M =_\beta N \\ M =_\beta N & \Rightarrow & N =_\beta M \\ M =_\beta N \;\&\; N =_\beta P & \Rightarrow & M =_\beta P \end{array}$$

To understand this definition, take again function f defined by $fx = x^2$, so such that $f = (\lambda x.x^2)$. Then $f3$ is the same as $(\lambda x.x^2)\,3$, which reduces by the rule above to $3^2 = 9$.

Using Barendregt's convention, it might seem that α-conversion is no longer needed, but this is not the case, since reduction will break the convention. Take for example the term $(\lambda xy.xy)\,(\lambda xy.xy)$, which adheres to Barendregt's convention. Reducing this term without α-conversion would give:

$$\begin{array}{rcl} (\lambda xy.xy)\,(\lambda xy.xy) & \rightarrow & (\lambda y.xy)\,[(\lambda xy.xy)/x] \\ & = & \lambda y.(\lambda xy.xy)y \end{array}$$

Notice that this term no longer adheres to Barendregt's convention, since y is bound and free in (the sub-term) $(\lambda xy.xy)y$. The problem here is that we need to be able to tell which occurrence of y is bound by which binder. At this stage we could still argue that we can distinguish the two ys by saying that the 'innermost' binding is strongest, and that only a free variable can be bound, and that therefore only *one* y (the right-most) is bound by the outermost binding. If, however, we continue with the reduction, we get

$$\begin{array}{rcl} (\lambda xy.xy)\,(\lambda xy.xy) & \rightarrow & (\lambda y.xy)\,[(\lambda xy.xy)/x] \\ & = & \lambda y.(\lambda xy.xy)y \\ & \rightarrow & \lambda y.(\lambda y.xy)\,[y/x] \\ & = & \lambda y.(\lambda y.yy) \end{array}$$

We would now be forced to accept that *both* ys are bound by the innermost λy. To avoid this, we need to α-convert the term $\lambda y.(\lambda xy.xy)y$ to $\lambda y.(\lambda xz.xz)y$ before performing the reduction. As mentioned above, α-conversion is a silent operation, so we get:

$$(\lambda xy.xy)\,(\lambda xy.xy) \;\rightarrow\; \lambda y.(\lambda xz.xz)y \;\rightarrow\; \lambda y.(\lambda z.yz) \;=\; \lambda yz.yz$$

a reduction that preserves the convention.

The following terms will reappear frequently in this course:

$$\begin{array}{rcl} \mathbf{I} & = & \lambda x.x \\ \mathbf{K} & = & \lambda xy.x \\ \mathbf{S} & = & \lambda xyz.xz(yz) \end{array}$$

Normally also the following reduction rule is considered, which expresses extensionality:

Definition 1.8 (η-REDUCTION) Let $x \notin fv(M)$, then $\lambda x.Mx \rightarrow_\eta M$.

As mentioned above, we can see '$\rightarrow_\beta$' as the 'one step' execution, and '$\twoheadrightarrow_\beta$' as a 'many step' execution. We can then view the relation '$=_\beta$' as 'executing the same function'.

Exercise 1.9 Show that, when $M =_\beta N$, then there are terms $M_1, M_2, \ldots, M_n, M_{n+1}$ such that $M \equiv M_1$, $N \equiv M_{n+1}$, and, for all $1 \leq i \leq n$, either $M_i \twoheadrightarrow_\beta M_{i+1}$, or $M_{i+1} \twoheadrightarrow_\beta M_i$.

This notion of reduction satisfies the 'Church-Rosser Property': if $M \twoheadrightarrow_\beta N$ and $M \twoheadrightarrow_\beta P$, then there exists a term Q such that $N \twoheadrightarrow_\beta Q$ and $P \twoheadrightarrow_\beta Q$.

$$\begin{tikzcd}
& M \arrow[dl, two heads, "\beta"'] \arrow[dr, two heads, "\beta"] & \\
N \arrow[dr, two heads, "\beta"'] & & P \arrow[dl, two heads, "\beta"] \\
& Q &
\end{tikzcd}$$

We will not show the proof of this property.

A popular variant of β-reduction that stands, for example, at the basis of reduction in Haskell [32], is that of *lazy* reduction (often called *call-by-need*), where a computation is delayed until the result is required. "The basic idea behind call-by-need is to start the evaluation of a procedure's body s soon as the procedure shows up in function position, to defer the evaluation of the argument until the evaluation of the procedure's body depends on the value of the argument, and to re-use the value of the argument for all other references to the parameter during the rest of the procedure's body evaluation. The implementation of this parameter-passing technique (...) is typically based on environments" [2]. It is this latter point which brings it out of scope of this course to discuss lazy evaluation strategies in detail: also, the choice of reduction strategy bears no relevance on the type discipline, but only on the efficiency of compilation and running code.

Although the Lambda Calculus itself has a very compact syntax, and its notion of reduction is easily defined, it is, in fact, a very powerful calculus: it is possible to encode all Turing machines (executable programs) into this calculus.

In particular, it is possible to have non-terminating terms.

Example 1.10 i) Take $(\lambda x.xx)\,(\lambda x.xx)$. This term reduces as follows:

$$\begin{array}{rcl}
(\lambda x.xx)\,(\lambda x.xx) & \rightarrow_\beta & (xx)\,[(\lambda x.xx)/x] \\
& = & (\lambda x.xx)\,(\lambda x.xx)
\end{array}$$

so this term reduces only to itself.

ii) Take $\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx))$. This term reduces as follows:

$$\begin{array}{rcl}
\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx)) & \rightarrow_\beta & \lambda f.(f(xx))\,[(\lambda x.f(xx))/x] \\
& = & \lambda f.f((\lambda x.f(xx))(\lambda x.f(xx))) \\
& \rightarrow_\beta & \lambda f.f(f((\lambda x.f(xx))(\lambda x.f(xx)))) \\
& \rightarrow_\beta & \lambda f.f(f(f((\lambda x.f(xx))(\lambda x.f(xx))))) \\
& \vdots & \\
& \rightarrow_\beta & \lambda f.f(f(f(f(f(f(f(\ldots)))))))
\end{array}$$

Actually, the second term also acts as a *fixed point constructor*, i.e. a term that maps any given term M to a term N that M maps unto itself, i.e. such that $MN =_\beta N$:

Exercise 1.11 Show that, for all terms M,

$$M\,((\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx)))\,M) \;=_\beta\; (\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx)))\,M$$

Of course, it is also possible to give λ-terms for which β-reduction is terminating.

Definition 1.12 *i*) A term of the shape $(\lambda x.M)\,N$ is called a *reducible expression* (*redex*); the term $M[N/x]$ that is obtained by reducing this term is called a *contractum* (from 'contracting the redex') or *reduct*.

ii) A term is *in normal form* if it does not contain a redex. Terms in normal form are defined by:

$$N ::= x \mid \lambda x.N \mid xN_1 \cdots N_n \ (n \geq 0)$$

iii) A term M is *in head-normal form* if it is of the shape $\lambda x_1 \cdots x_n.yM_1 \ldots M_m$, with $n \geq 0$ and $m \geq 0$; then y is called the *head-variable*. Terms in head-normal form can be defined by:

$$H ::= x \mid \lambda x.H \mid xM_1 \cdots M_n \ (n \geq 0, M_i \in \Lambda)$$

iv) A term M is *(head-)normalizable* is it has a (head-)normal form, i.e. if there exists a term N in (head-)normal form such that $M \twoheadrightarrow_\beta N$.

v) A term M is *strongly normalizable* if all reduction sequences starting from M are finite.

Exercise 1.13 Show that all terms in normal form are in head-normal form.

Example 1.14 i) The term $\lambda f.(\lambda x.f(xx))\ (\lambda x.f(xx))$ contains an occurence of a redex (being $(\lambda x.f(xx))\ (\lambda x.f(xx))$), so is not in normal form. It is also not in head-normal form, since it does not have a head-variable. However, its reduct $\lambda f.f((\lambda x.f(xx))\ (\lambda x.f(xx)))$ is in head-normal form. Notice that it is not in normal form, since still the same redex occurs.

ii) The term $(\lambda x.xx)\ (\lambda x.xx)$ is a redex, so not in normal form. It does not have a normal form, since it only reduces to itself, so all its reducts will contain a redex. Similarly, it does not have a head-normal form.

iii) The term $(\lambda xyz.xz(yz))\ (\lambda ab.a)$ is a redex, so not in normal form. It has only one reduct, $(\lambda yz.(\lambda ab.a)z(yz))$, which has only one redex $(\lambda ab.a)z$. Contracting this redex gives $\lambda yz.(\lambda b.z)\ (yz)$, which again has only one redex, which reduces to $\lambda yz.z$. We have obtained a normal form, so the original term is normalizable. Also, we have contracted all possible redexes, so the original term is strongly normalizable.

iv) The term $(\lambda ab.b)\ ((\lambda x.xx)\ (\lambda x.xx))$ has two redexes. Contracting the first (outermost) will create the term $\lambda b.b$. This term is in normal form, so the original term has a normal form. Contracting the second redex (innermost) will create the same term, so repeatedly contracting this redex will give an infinite reduction path. In particular, the term is not strongly normalizable.

Exercise 1.15 Show that, if M has a normal form, it is unique (hint: use the Church-Rosser property).

We have already mentioned that it is possible to encode all Turing machines into LC. A consequence of this result is that we have a 'halting problem' also in LC: it is impossible to decide if a given term is going to terminate. It is also impossible to decide if two terms are the same according to $=_\beta$.

1.3 Example: a numeral system

We will now show the expressivity of LC by encoding numbers and some basic operations on them as λ-terms; because of Church's Thesis, all computable functions are, in fact, encodable, but we will not go there.

Definition 1.16 We define the *booleans True* and *False* as:

$$\begin{array}{lcl} \mathit{True} &=& \lambda xy.x \\ \mathit{False} &=& \lambda xy.y \end{array}$$

Then the *conditional* is defined as:

$$\mathit{Cond} = \lambda btf.btf$$

Exercise 1.17 Verify that Cond True M N → M and Cond False M N → N.

In fact, we can see *Cond* as syntactic sugar, since also $\mathit{True}\ M\ N \to M$ and $\mathit{False}\ M\ N \to N$.

Definition 1.18 The operation of *pairing* is defined via

$$[M,N] = \lambda z.zMN$$

and the first and second projection functions are defined by

$$\begin{array}{lcl} \mathit{First} &=& \lambda p.p\mathit{True} \\ \mathit{Second} &=& \lambda p.p\mathit{False} \end{array}$$

Exercise 1.19 Verify that $\mathit{First}\,[M,N] \to_\beta M$ *and* $\mathit{Second}\,[M,N] \to_\beta N$.

Numbers can now be encoded quite easily by specifying how to encode zero and the successor function (remember that $\mathbb{N}$, the set of natural numbers, is defined as the smallest set that contains 0 and is closed for the (injective) successor function):

Definition 1.20 The (Scott) *Numerals* are defined by:

$$\begin{array}{lcl} \llbracket 0 \rrbracket &=& \mathbf{K} \\ \mathit{Succ} &=& \lambda nxy.yn \\ \mathit{Pred} &=& \lambda p.p\mathbf{KI} \end{array}$$

so, for example,

$$\begin{array}{rcl} \llbracket 3 \rrbracket &=& \llbracket S(S(S(0))) \rrbracket \\ &=& \mathit{Succ}\,(\mathit{Succ}\,(\mathit{Succ}\,\llbracket 0 \rrbracket)) \\ &=& (\lambda nxy.yn)((\lambda nxy.yn)((\lambda nxy.yn)(\mathbf{K}))) \\ &\to_\beta& \lambda xy.y(\lambda xy.y(\lambda xy.y\mathbf{K})). \end{array}$$

Notice that $\llbracket 0 \rrbracket = \mathit{True}$. It is now easy to check that

$$\mathit{Cond}\ \llbracket n \rrbracket\ f\ g \twoheadrightarrow_\beta \begin{cases} f & \text{if } \llbracket n \rrbracket = 0 \\ g\ \llbracket n-1 \rrbracket & \text{otherwise} \end{cases}$$

which implies that, in this system, we can define the test *IsZero* as identity, $\lambda x.x$.

Of course, this definition only makes sense if we can actually express, for example, addition and multiplication.

Definition 1.21 Addition and multiplication are now defined by:

$$\begin{array}{lcl} \mathit{Add} &=& \lambda xy.\mathit{Cond}\ (\mathit{IsZero}\ x)\ y\ (\mathit{Succ}\ (\mathit{Add}\ (\mathit{Pred}\ x)\ y)) \\ \mathit{Mult} &=& \lambda xy.\mathit{Cond}\ (\mathit{IsZero}\ x)\ \lambda x.x\ (\mathit{Add}\ (\mathit{Mult}\ (\mathit{Pred}\ x)\ y)\ y) \end{array}$$

Notice that these definitions are recursive; we will later see that we can express recursion in LC.

There are alternative ways of encoding numbers in LC, like the *Church Numerals* $\underline{n} = \lambda fn.f^n x$ which would give an elegant encoding of addition and multiplication, but has a very complicated definition of predecessor.

Exercise 1.22 Verify that these definitions are correct, i.e. show:

$$\begin{array}{lll} \mathit{Pred}\ (\mathit{Succ}\ \llbracket n \rrbracket) & \twoheadrightarrow_\beta & \llbracket n \rrbracket \\ \mathit{Succ}\ (\mathit{Pred}\ \llbracket n \rrbracket) & \twoheadrightarrow_\beta & \llbracket n \rrbracket \\ \mathit{Add}\ \llbracket n \rrbracket\ \llbracket m \rrbracket & \twoheadrightarrow_\beta & \llbracket n+m \rrbracket \\ \mathit{Mult}\ \llbracket n \rrbracket\ \llbracket m \rrbracket & \twoheadrightarrow_\beta & \llbracket n \times m \rrbracket \end{array}$$

For the third result, it suffices to show that both

$$\begin{array}{lll} \mathit{Add}\ \llbracket n \rrbracket\ \llbracket 0 \rrbracket & \twoheadrightarrow_\beta & \llbracket n \rrbracket \text{ and} \\ \mathit{Add}\ \llbracket n \rrbracket\ \llbracket m+1 \rrbracket & \twoheadrightarrow_\beta & \mathit{Succ}(\mathit{Add} \llbracket n \rrbracket\ \llbracket m \rrbracket); \end{array}$$

The fourth result is similar.

2 The Curry type assignment system

In this section, we will present the basic notion of type assignment of LC, as first studied by H.B. Curry in [21]. (See also [22].) Curry's system – the first and most primitive one – expresses abstraction and application and has as its major advantage that the problem of type assignment (given a term M, are there Γ, A such that $\Gamma \vdash_{\mathrm{C}} M : A$) is decidable.

2.1 Curry type assignment

Type assignment follows the *syntactic structure* of terms, building the type of more complex objects out of the type(s) derived for its immediate syntactic component(s). The main feature of the system is that terms of the shape '$\lambda x.M$' will get a type of the shape '$A \rightarrow B$', which accurately expresses the fact that we see the term as a function, 'waiting' for an input. The type for $\lambda x.M$ is built out of the search for the type for M itself: if M has type B, and in this analysis we have used no other type than A for the occurrences x in M, we say that $A \rightarrow B$ is a type for the abstraction. Likewise, if a term M has been given the type $A \rightarrow B$, and a term N the type A, than apparently the second term N is of the right kind to be an input for M, so we can safely build the application MN and say that is has type B.

It is formulated by a system of *derivation rules* that act as description of building stones that are used to build derivations, and the two obvious observations above are reflected by the two derivation rules $(\rightarrow I)$ and $(\rightarrow E)$ as below in Definition 2.2. Such a derivation has a conclusion (the expression of the shape $\Gamma \vdash_{\mathrm{C}} M : A$ that appears in the bottom line), which states the type for the term we are interested in.

Type assignment assigns types to λ-terms, where types are defined as follows:

Definition 2.1 *i*) $\mathcal{T}_{\mathrm{C}}$, the set of *types*, ranged over by $A, B, \ldots$, is defined over a set of *type variables* Φ, ranged over by φ, by:

$$A ::= \varphi \mid (A \rightarrow B)$$

ii) A *statement* is an expression of the form $M{:}A$, where $M \in \Lambda$ and $A \in \mathcal{T}_{\mathrm{C}}$. M is called the *subject* and A the *predicate* of $M{:}A$.

iii) A *context* Γ is a set of statements with only distinct variables as subjects; we use $\Gamma, x{:}A$

for the context defined as $\Gamma \cup x{:}A$ where either $x{:}A \in \Gamma$ or x does not occur in Γ.

The notion of context will be used to collect all statements used for the free variables of a term when typing that term. In the notation of types, right most and outer-most parentheses are normally omitted, so $(\psi_1 \to \psi_2) \to \psi_3 \to \psi_4$ stands for $((\psi_1 \to \psi_2) \to (\psi_3 \to \psi_4))$.

We will now give the definition of Curry type assignment.

Definition 2.2 (cf. [21, 22]) *i*) *Curry type assignment* and *derivations* are defined by the following derivation rules that define a natural deduction system.

$$(Ax): \frac{}{\Gamma, x{:}A \vdash_{\mathrm{c}} x : A}$$

$$(\to I): \frac{\Gamma, x{:}A \vdash_{\mathrm{c}} M : B}{\Gamma \vdash_{\mathrm{c}} \lambda x . M : A \to B}$$

$$(\to E): \frac{\Gamma \vdash_{\mathrm{c}} M_1 : A \to B \quad \Gamma \vdash_{\mathrm{c}} M_2 : A}{\Gamma \vdash_{\mathrm{c}} M_1 M_2 : B}$$

ii) We will write $\Gamma \vdash_{\mathrm{c}} M : A$ if this statement is derivable, i.e. if there exists a derivation, built using these three rules, that has this statement in the bottom line.

Exercise 2.3 Verify the following results:

i) $\emptyset \vdash_{\mathrm{c}} \lambda x . x : A \to A$.

ii) $\emptyset \vdash_{\mathrm{c}} \lambda xy . x : A \to B \to A$.

iii) $\emptyset \vdash_{\mathrm{c}} \lambda xyz . xz(yz) : (A \to B \to C) \to (A \to B) \to A \to C$.

iv) $\emptyset \vdash_{\mathrm{c}} \lambda bc . c : A \to B \to B$.

v) $\emptyset \vdash_{\mathrm{c}} \lambda bc . (\lambda y . c)\ (bc) : (B \to A) \to B \to B$.

vi) $\emptyset \vdash_{\mathrm{c}} \lambda bc . (\lambda xy . x)\, c(bc) : (B \to A) \to B \to B$.

vii) $\emptyset \vdash_{\mathrm{c}} (\lambda abc . ac(bc))\ (\lambda xy . x) : (B \to A) \to B \to B$.

Example 2.4 i) We cannot type '*self-application*', xx. Since a context should contain statements with distinct variables as subjects, there can only be *one* type for x in any context. In order to type xx, the derivation should have the structure

$$\frac{\dfrac{}{\Gamma \vdash_{\mathrm{c}} x : A \to B}\ (x{:}A \to B \in \Gamma) \qquad \dfrac{}{\Gamma \vdash_{\mathrm{c}} x : A}\ (x{:}A \in \Gamma)}{\Gamma \vdash_{\mathrm{c}} xx : B}$$

for certain A and B (we will normally omit the side-condition when constructing derivations). So we need to find a solution for $A \to B = A$, and this is impossible. In fact, the procedure pp_{C} that tries to construct a type for terms as defined below will fail on xx. For this reason, the term $\lambda x . xx$ is not typeable, and neither is $\lambda f . (\lambda x . f(xx))(\lambda x . f(xx))$.

ii) We cannot derive $\emptyset \vdash_{\mathrm{c}} \lambda bc . (\lambda y . c)\ (bc) : A \to B \to B$, so we lose the converse of the Subject Reduction property (see Theorem 2.6), the *Subject Expansion*: If $\Gamma \vdash_{\mathrm{c}} M : A$ and $N \twoheadrightarrow_{\beta} M$, then $\Gamma \vdash_{\mathrm{c}} N : A$. The counter example is in the previous exercise: take $M = \lambda bc . c$, and $N = \lambda bc . (\lambda y . c)\ (bc)$, then it is easy to check that $N \twoheadrightarrow_{\beta} M$, $\emptyset \vdash_{\mathrm{c}} \lambda bc . c : A \to B \to B$, but not $\emptyset \vdash_{\mathrm{c}} \lambda bc . (\lambda y . c)\ (bc) : A \to B \to B$.

Exercise 2.5 Verify the following results:

i) If $\Gamma \vdash_{\mathrm{c}} M : A$, *and* $\Gamma' \supseteq \Gamma$, *then* $\Gamma' \vdash_{\mathrm{c}} M : A$.

ii) If $\Gamma \vdash_{\mathrm{c}} M : A$, *then* $\{ x{:}B \mid x{:}B \in \Gamma \ \&\ x \in fv(M) \} \vdash_{\mathrm{c}} M : A$.

iii) If $\Gamma \vdash_{\mathrm{c}} M : A$ *and* $x \in fv(M)$, *then there exists* B *such that* $x{:}B \in \Gamma$.

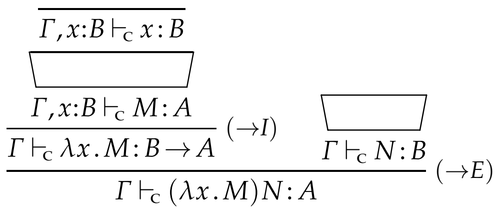
2.2 Subject Reduction

The following property states that types are preserved under reduction; this is an important property within the context of programming, because it states that the type we can assign to a program can also be assigned to the result of running the program; so it the type of a program M is `Integer`, then we can safely put it in a context that demands an `Integer`, as in $1 + M$, because running M will return an `Integer` - we do not know which, of course, until we actually run M, so our type analysis acts like an abstract interpretation of M.

Theorem 2.6 *If $\Gamma \vdash_{\mathrm{c}} M : A$ and $M \twoheadrightarrow_\beta N$, then $\Gamma \vdash_{\mathrm{c}} N : A$.*

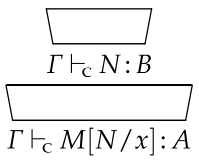
We can illustrate this result by the following:

Example 2.1 Suppose first that $\Gamma \vdash_{\mathrm{c}} (\lambda x.M)N : A$ is derived by $(\rightarrow E)$, then there exists B such that $\Gamma \vdash_{\mathrm{c}} \lambda x.M : B \rightarrow A$ and $\Gamma \vdash_{\mathrm{c}} N : B$. Then $(\rightarrow I)$has to be the last step performed for the first result, and there are sub-derivations for $\Gamma, x{:}B \vdash_{\mathrm{c}} M : A$ and $\Gamma \vdash_{\mathrm{c}} N : B$.



$\Gamma, x{:}B \vdash_{\mathrm{c}} x : B$
$\Gamma, x{:}B \vdash_{\mathrm{c}} M : A$
$(\rightarrow I)$
$\Gamma \vdash_{\mathrm{c}} \lambda x.M : B \rightarrow A$
$\Gamma \vdash_{\mathrm{c}} N : B$
$(\rightarrow E)$
$\Gamma \vdash_{\mathrm{c}} (\lambda x.M)N : A$

Then a derivation for $\Gamma \vdash_{\mathrm{c}} M[N/x] : A$ can be obtained by replacing in the derivation for $\Gamma, x{:}B \vdash_{\mathrm{c}} M : A$, the sub-derivation $\Gamma, x{:}B \vdash_{\mathrm{c}} x : B$ by the derivation for $\Gamma \vdash_{\mathrm{c}} N : B$.



$\Gamma \vdash_{\mathrm{c}} N : B$
$\Gamma \vdash_{\mathrm{c}} M[N/x] : A$

In order to formally prove this, we first prove a term substitution lemma.

Lemma 2.7 $\exists C\ [\Gamma, x{:}C \vdash_{\mathrm{c}} M : A \ \&\ \Gamma \vdash_{\mathrm{c}} N : C] \Rightarrow \Gamma \vdash_{\mathrm{c}} M[N/x] : A$.

Proof: By induction on the structure of terms.

$$\begin{array}{lll}
(M \equiv x): & \exists C\ [\Gamma, x{:}C \vdash_{\mathrm{c}} x : A \ \&\ \Gamma \vdash_{\mathrm{c}} N : C] & \Rightarrow\ (Ax) \\
 & \Gamma, x{:}A \vdash_{\mathrm{c}} x : A \ \&\ \Gamma \vdash_{\mathrm{c}} N : A & \Rightarrow \\
 & \Gamma \vdash_{\mathrm{c}} x[N/x] : A. &
\end{array}$$

$(M \equiv y \neq x):\ \exists C\ [\Gamma, x{:}C \vdash_{\mathrm{c}} y : A \ \&\ \Gamma \vdash_{\mathrm{c}} N : C] \Rightarrow (2.5ii))\ \Gamma \vdash_{\mathrm{c}} y : A$.

$$\begin{array}{lll}
(M \equiv \lambda y.M'): & \exists C\ [\Gamma, x{:}C \vdash_{\mathrm{c}} \lambda y.M' : A \ \&\ \Gamma \vdash_{\mathrm{c}} N : C] & \Rightarrow\ (\rightarrow I) \\
 & \exists C, A', B'\ [\Gamma, x{:}C, y{:}A' \vdash_{\mathrm{c}} M' : B' \ \&\ A = A' \rightarrow B' \ \&\ \Gamma \vdash_{\mathrm{c}} N : C] & \Rightarrow\ (IH) \\
 & \exists A', B'\ [\Gamma, y{:}A' \vdash_{\mathrm{c}} M'[N/x] : B' \ \&\ A = A' \rightarrow B'] & \Rightarrow\ (\rightarrow I) \\
 & \Gamma \vdash_{\mathrm{c}} \lambda y.M'[N/x] : A. & = \\
 & \Gamma \vdash_{\mathrm{c}} (\lambda y.M')[N/x] : A. &
\end{array}$$

$$\begin{array}{lll}
(M \equiv M_1 M_2): & \exists C\ [\Gamma, x{:}C \vdash_{\mathrm{c}} M_1 M_2 : A \ \&\ \Gamma \vdash_{\mathrm{c}} N : C] & \Rightarrow\ (\rightarrow E) \\
 & \exists B\ [\Gamma, x{:}C \vdash_{\mathrm{c}} M_1 : B \rightarrow A \ \&\ \Gamma, x{:}C \vdash_{\mathrm{c}} M_2 : B \ \&\ \Gamma \vdash_{\mathrm{c}} N : C] & \Rightarrow\ (IH) \\
 & \exists B\ [\Gamma \vdash_{\mathrm{c}} M_1[N/x] : B \rightarrow A \ \&\ \Gamma \vdash_{\mathrm{c}} M_2[N/x] : B] & \Rightarrow\ (\rightarrow E) \\
 & \Gamma \vdash_{\mathrm{c}} M_1[N/x]\ M_2[N/x] : A & = \\
 & \Gamma \vdash_{\mathrm{c}} (M_1 M_2)[N/x] : A.\ \blacksquare &
\end{array}$$

The proof for Theorem 2.6 then becomes:

Proof: By induction on the definition of $\twoheadrightarrow_\beta$. The only part shown here is that of a redex, $\Gamma \vdash_c (\lambda x.M)N : A \Rightarrow \Gamma \vdash_c M[N/x] : A$. Notice that, if $\Gamma \vdash_c (\lambda x.M)N : A$, then, by $(\to E)$ and $(\to I)$, there exists C such that $\Gamma, x{:}C \vdash_c M : A$ and $\Gamma \vdash_c N : C$. The result then follows from Lemma 2.7. ■

Exercise 2.8 Finish this proof; all added cases should follow by straightforward induction.

We can also show that type assignment is closed for η-reduction:

Theorem 2.9 *If $\Gamma \vdash_c M : A$ and $M \to_\eta N$, then $\Gamma \vdash_c N : A$.*

Proof: By induction on the definition of $\to_\eta$, of which only the part $\lambda x.Mx \to_\eta M$ is shown, where x does not occur free in M. The other parts are dealt with by straightforward induction. Assume $x \notin fv(M)$, then

$$\begin{array}{lll}
\Gamma \vdash_c \lambda x.Mx : A & \Rightarrow & (\to I) \\
\exists B,C\, [A = B\to C \;\&\; \Gamma, x{:}B \vdash_c Mx : C] & \Rightarrow & (\to E) \\
\exists B,C,D\, [A = B\to C \;\&\; \Gamma, x{:}B \vdash_c M : D\to C \;\&\; \Gamma, x{:}B \vdash_c x : D] & \Rightarrow & (2.3) \\
\exists B,C,D\, [A = B\to C \;\&\; \Gamma \vdash_c M : D\to C \;\&\; B = D] & \Rightarrow & \\
\exists B,C\, [A = B\to C \;\&\; \Gamma \vdash_c M : B\to C] & \Rightarrow & \Gamma \vdash_c M : A. \;\blacksquare
\end{array}$$

2.3 The principal type property

Principal type schemes for Curry's system were first defined in [30]. In this paper Hindley actually proved the existence of principal types for an object in Combinatory Logic, but the same construction can be used for a proof of the principal type property for terms in the Lambda Calculus.

The principal type property expresses that, amongst the whole family of types you can assign to a term, there is one that can be called 'principal' in the sense that all other types can be created from it. For the system as defined in this section, the 'creation' of types is done by (type-)substitution, defined as the operation on types that replaces type variables by types in a consistent way.

Definition 2.10 (Type substitution) *i*) *a*) The *substitution* $(\varphi \mapsto C) : \mathcal{T}_c \to \mathcal{T}_c$, where φ is a type variable and $C \in \mathcal{T}_c$, is inductively defined by:

$$\begin{array}{lcl}
(\varphi \mapsto C)\; \varphi & = & C \\
(\varphi \mapsto C)\; \varphi' & = & \varphi', \text{ if } \varphi' \neq \varphi \\
(\varphi \mapsto C)\; A\to B & = & ((\varphi \mapsto C)\; A) \to ((\varphi \mapsto C)\; B)
\end{array}$$

b) If S_1, S_2 are substitutions, then so is $S_1 \circ S_2$, where $S_1 \circ S_2\, A = S_1\,(S_2\, A)$.

c) $S\,\Gamma = \{ x{:}S\,B \mid x{:}B \in \Gamma \}$.

d) $S\,\langle \Gamma, A \rangle = \langle S\,\Gamma, S\,A \rangle$.

ii) If there is a substitution S such that $S\,A = B$, then B is a *(substitution) instance* of A.

Exercise 2.11 Show that, for all substitutions S and types A and B, $S(A \to B) = SA \to SB$.

So, for the Curry type assignment system, the principal type property is expressed by: for each typeable term M, there exist a *principal pair* of context Π and type P such that $\Pi \vdash_c M : P$, and for all context Γ, and types A, if $\Gamma \vdash_c M : A$, then there exists a substitution S such that

$S\,\Pi = \Gamma$ and $S\,P = A$.

The principal type property for type assignment systems plays an important role in programming languages that model *polymorphic* functions, where a function is called polymorphic if it can be correctly applied to objects of various types. In fact, the principal type there functions as a general *type-scheme* that models all possible types for the procedure involved.

Before we come to the actual proof that the Curry type assignment system has the principal type property, we need to show that substitution is a sound operation:

Lemma 2.12 (Soundness of substitution) *For every substitution S: if (we have a derivation for) $\Gamma \vdash_{\rm c} M : A$, then $S\,\Gamma \vdash_{\rm c} M : S\,A$.*

Proof: By induction on the structure of derivations.

- (Ax) : Then $M \equiv x$, and $x{:}A \in \Gamma$. Notice that then $x{:}S\,A \in S\,\Gamma$, so, by rule (Ax), $S\,\Gamma \vdash_{\rm c} x : S\,A$.
- $(\rightarrow I)$: Then there are M', A, C such that $M \equiv \lambda x . M'$, $A = C \rightarrow D$, and $\Gamma, x{:}C \vdash_{\rm c} M' : D$. Since this statement is derived in a sub-derivation, we know that $S\,(\Gamma, x{:}C) \vdash_{\rm c} M' : S\,D$ follows by induction. Since $S\,(\Gamma, x{:}C) = S\,\Gamma, x{:}S\,C$, we also have $S\,\Gamma, x{:}S\,C \vdash_{\rm c} M' : S\,D$. To this result we can apply rule $(\rightarrow I)$, to obtain $S\,\Gamma \vdash_{\rm c} \lambda x . M' : S\,C \rightarrow S\,D$. Since $S\,C \rightarrow S\,D = S\,(C \rightarrow D) = S\,A$ by definition of substitutions, we get $S\,\Gamma \vdash_{\rm c} \lambda x . M' : S\,A$.
- $(\rightarrow E)$: Then M_1, M_2, B such that $M \equiv M_1 M_2$, $\Gamma \vdash_{\rm c} M_1 : B \rightarrow A$, and $\Gamma \vdash_{\rm c} M_2 : B$. Since these two statements are derived in a sub-derivation, we know that $S\,\Gamma \vdash_{\rm c} M_1 : S\,(B \rightarrow A)$ and $S\,\Gamma \vdash_{\rm c} M_2 : S\,B$ follow by induction. Since $S\,(B \rightarrow A) = S\,B \rightarrow S\,A$ by definition of substitution, so we also have $S\,\Gamma \vdash_{\rm c} M_1 : S\,B \rightarrow S\,A$, and we can apply rule $(\rightarrow E)$ to obtain $S\,\Gamma \vdash_{\rm c} M_1 M_2 : S\,A$. ■

Principal types for λ-terms are defined using the notion of unification of types that was defined by Robinson in [48]. Robinson's unification is a procedure on types which, given two arguments, returns a substitution that maps the arguments to a smallest common instance with respect to substitution. It can be defined as follows:

Definition 2.13 Let Id_S be the substitution that replaces all type variables by themselves.

i) Robinson's unification algorithm. Unification of Curry types is defined by:

$$\begin{array}{llll}
\textit{unify}\ \varphi & \varphi & = & (\varphi \mapsto \varphi), \\
\textit{unify}\ \varphi & B & = & (\varphi \mapsto B), \text{ if } \varphi \text{ does not occur in } B \\
\textit{unify}\ A & \varphi & = & \textit{unify}\ \varphi\ A \\
\textit{unify}\ (A \rightarrow B) & (C \rightarrow D) & = & S_2 \circ S_1, \\
 & & & \text{where } S_1 = \textit{unify}\ A\ C \\
 & & & \phantom{\text{where }} S_2 = \textit{unify}\ (S_1\,B)\ (S_1\,D)
\end{array}$$

ii) By defining the operation *UnifyContexts*, the operation *unify* can be generalised to contexts:

$$\begin{array}{llll}
\textit{UnifyContexts}\ (\Gamma_0, x{:}A) & (\Gamma_1, x{:}B) & = & S_2 \circ S_1, \\
 & & & \text{where } S_1 = \textit{unify}\ A\ B \\
 & & & \phantom{\text{where }} S_2 = \textit{UnifyContexts}\ (S_1\,\Gamma_0)\ (S_1\,\Gamma_1) \\
\textit{UnifyContexts}\ (\Gamma_0, x{:}A) & \Gamma_1 & = & \textit{UnifyContexts}\ \Gamma_0\ \Gamma_1, \text{ if } x \text{ does not occur in } \Gamma_1. \\
\textit{UnifyContexts}\ \emptyset & \Gamma_1 & = & Id_S.
\end{array}$$

The following property of Robinson's unification is very important for all systems that depend on it, and formulates that *unify* returns the *most general unifier* of two types. This means that if two types A and B have a *common substitution instance*, then they have a *least* *common instance* γ which can be created through applying the unifier of A and B to A (or to B), and all their common instances can be obtained from γ by substitution.

Proposition 2.14 ([48]) *For all A, B: if S_1 is a substitution such that $S_1 A = S_1 B$, then there are substitutions S_2 and S_3 such that*

$$\begin{array}{rcll} S_2 & = & \mathit{unify}\ A\ B & \text{and} \\ S_1 A = S_3{\circ}S_2 A & = & S_3{\circ}S_2 B = S_1 B. & \end{array}$$

Exercise 2.15 Show that unify $(\mathit{unify}\ A\ B\ A)\ C = \mathit{unify}\ A\ (\mathit{unify}\ B\ C\ C)$. (Notice that unify A B is a substitution that gets applied to A before the second unification on the left takes place (with C); moreover, we could have applied it to B rather than A with no resulting difference.)

The definition of principal pairs for λ-terms in Curry's system then looks like:

Definition 2.16 We define for every term M the *(Curry) principal pair* by defining the notion $pp_C\, M = \langle \Pi, P\rangle$ inductively by:

i) For all x, φ: $pp_C\, x = \langle x{:}\varphi, \varphi\rangle$.

ii) If $pp_C\, M = \langle \Pi, P\rangle$, then:

a) If $x \in fv(M)$, then, by Exercise 2.5 *iii*), there is a A such that $x{:}A \in \Pi$, and $pp_C\, \lambda x.M = \langle \Pi\backslash x, A{\to}P\rangle$.

b) otherwise $pp_C\, (\lambda x.M) = \langle \Pi, \varphi{\to}P\rangle$, where φ does not occur in $\langle \Pi, P\rangle$.

iii) If $pp_C\, M_1 = \langle \Pi_1, P_1\rangle$ and $pp_C\, M_2 = \langle \Pi_2, P_2\rangle$ (we choose, if necessary, trivial variants by renaming type variables such that the $\langle \Pi_i, P_i\rangle$ have no type variables in common), φ is a type variable that does not occur in either of the pairs $\langle \Pi_i, P_i\rangle$, and

$$\begin{array}{rcl} S_1 & = & \mathit{unify}\ P_1\ (P_2{\to}\varphi) \\ S_2 & = & \mathit{UnifyContexts}\ (S_1\, \Pi_1)\ (S_1\, \Pi_2), \end{array}$$

then $pp_C\, (M_1 M_2) = S_2{\circ}S_1\, \langle \Pi_1 \cup \Pi_2, \varphi\rangle$.

A principal pair for M is often called a *principal typing*.

Notice that, rather than using *UnifyContexts*, we could have closed the terms by building abstractions towards the variables occurring in the union contexts, using fresh variables whenever a term variable occurs in one context only, and unify the resulting arrow types.

This definition in fact gives an algorithm that finds the principal pair for λ-terms. Below, where we specify that algorithm more like a program, we do not deal explicitly with the error cases. This is mainly for readability: including an error case somewhere (which would originate from *unify*) would mean that we would have to 'catch' those in every function call to filter out the fact that the procedure can return an error; we will see this later also when we discuss Haskell's monads.

The algorithm as presented here is not purely functional. The 0-ary function *fresh* is supposed to return a new, unused type variable. It is obvious that such a function is not referential transparent, but for the sake of readability, we prefer not to be explicit on the handling of type variables.

Definition 2.17 (pp_C, PRINCIPAL PAIR ALGORITHM FOR Λ)

$$\begin{array}{rcl} pp_C\ x & = & \langle x{:}\varphi, \varphi\rangle \\ & & \text{where } \varphi = \mathit{fresh} \end{array}$$

$$\begin{array}{lcll}
pp_C\ (\lambda x.M) & = & \langle \Pi, A{\to}P\rangle, \text{ if } pp_C\, M = \langle \Pi \cup \{x{:}A\}, P\rangle & \\
 & & \langle \Pi, \varphi{\to}P\rangle, \text{ if } pp_C\, M = \langle \Pi, P\rangle & x \notin \Pi \\
 & & \qquad\text{where } \varphi = \mathit{fresh} & \\
pp_C\ MN & = & S_2{\circ}S_1\,\langle \Pi_1 \cup \Pi_2, \varphi\rangle & \\
 & & \text{where } \varphi \qquad\;\; = \mathit{fresh} & \\
 & & \qquad \langle \Pi_1, P_1\rangle = pp_C\, M & \\
 & & \qquad \langle \Pi_2, P_2\rangle = pp_C\, N & \\
 & & \qquad S_1 \qquad\;\; = \mathit{unify}\ P_1\ (P_2{\to}\varphi) & \\
 & & \qquad S_2 \qquad\;\; = \mathit{UnifyContexts}\ (S_1\,\Pi_1)\ (S_1\,\Pi_2) &
\end{array}$$

Extending this into a runnable program is a matter of patient specification: for example, above we ignore how contexts are represented, as well as the fact that in implementation, substitutions are not that easily extended from types to contexts in a program.

Exercise 2.18 Show that, if $pp_C\, M = \langle \Pi, P\rangle$, then $\Pi \vdash_c M{:}\,P$. You'll need Lemma 2.12 here.

The proof that the procedure 'pp_C' indeed returns the principal pairs is given by showing that all possible pairs for a typeable term M can be obtained from the principal one by applying substitutions. In this proof, Property 2.14 is needed.

Theorem 2.19 (COMPLETENESS OF SUBSTITUTION.) *If $\Gamma \vdash_c M{:}\,A$, then there are Π, P and a substitution S such that: $pp_C\, M = \langle \Pi, P\rangle$, and $S\,\Pi \subseteq \Gamma$, $S\,P = A$.*

Proof: By induction on the structure of terms in Λ.

$(M \equiv x)$: Then, by rule (Ax), $x{:}A \in \Gamma$, and $pp_C\, x = \langle \{x{:}\varphi\}, \varphi\rangle$ by definition. Take $S = (\varphi \mapsto A)$.

$(M \equiv \lambda x.M')$: Then, by rule $(\to I)$, there are C, D such that $A = C{\to}D$, and $\Gamma, x{:}C \vdash_c M'{:}\,D$. Then, by induction, there are Π', P' and S' such that $pp_C\, M' = \langle \Pi', P'\rangle$, and $S'\Pi' \subseteq \Gamma, x{:}C$, $S\,P' = D$. Then either:

a) x occurs free in M', and there exists an A' such that $x{:}A' \in \Pi'$, and $pp_C\,(\lambda x.M') = \langle \Pi' \backslash x, A'{\to}P\rangle$. Since $S'\,\Pi' \subseteq \Gamma, x{:}C$, in particular $S'\,A' = C$. Also $S'\,(\Pi' \backslash x) \subseteq \Gamma$. Notice that now $S'\,(A'{\to}P') = C{\to}D$. Take $\Pi = \Pi' \backslash x$, $P = A'{\to}P'$, and $S = S'$.

b) otherwise $pp_C\,(\lambda x.M) = \langle \Pi', \varphi{\to}P'\rangle$, x does not occur in Π', and φ does not occur in $\langle \Pi', P'\rangle$. Since $S'\,\Pi' \subseteq \Gamma, x{:}C$, in particular $S'\,\Pi' \subseteq \Gamma$. Take $S = S' \circ (\varphi \mapsto C)$, then, since φ does not occur in Π', also $S\,\Pi' \subseteq \Gamma$. Notice that $S\,(\varphi{\to}P') = C{\to}D$; take $\Pi = \Pi'$, $P = \varphi{\to}P'$.

$(M = M_1 M_2)$: Then, by rule $(\to E)$, there exists a B such that $\Gamma \vdash_c M_1{:}\,B{\to}A$ and $\Gamma \vdash_c M_2{:}\,B$. By induction, there are S_1, S_2, $\langle \Pi_1, P_1\rangle = pp_C\, M_1$ and $\langle \Pi_2, P_2\rangle = pp_C\, M_2$ (no type variables shared) such that $S_1\,\Pi_1 \subseteq \Gamma$, $S_2\,\Pi_2 \subseteq \Gamma$, $S_1\,P_1 = B{\to}A$ and $S_2\,P_2 = B$.

Let φ be a type variable that does not occur in any of the pairs $\langle \Pi_i, P_i\rangle$, and

$$\begin{array}{lcl}
S_3 & = & \mathit{unify}\ P_1\ (P_2{\to}\varphi) \\
S_4 & = & \mathit{UnifyContexts}\ (S_3\,\Pi_1)\ (S_3\,\Pi_2)
\end{array}$$

then, by the definition above, $pp_C\,(M_1 M_2) = S_4{\circ}S_3\,\langle (\Pi_1 \cup \Pi_2),\ \varphi\rangle$. Since we know that

$$\begin{array}{lcl}
S_1{\circ}S_2\,P_1 & = & B{\to}A, \\
S_1{\circ}S_2\,P_2 & = & B, \text{and} \\
S_1{\circ}S_2\,\varphi & = & A
\end{array}$$

so $S_1 \circ S_2 \circ (\varphi \mapsto A)$ unifies P_1 and $P_2 \rightarrow \phi$, as well as

$$\begin{array}{lcl} S_1 \circ S_2 \Pi_1 & \subseteq & \Gamma, \text{ and} \\ S_1 \circ S_2 \Pi_1 & \subseteq & \Gamma \end{array}$$

by Proposition 2.14 there exists a substitution S' such that $S_1 \circ S_2 \circ (\varphi \mapsto A) = S' \circ S_3 \circ S_4$. ■

Below, we will present an alternative definition to the above algorithm, which makes the procedure *UnifyContexts* obsolete. This is explained by the fact that the algorithm starts with a context that assigns a different type variable for every term variable, executes any change as a result of unification to the types in contexts, and passes (modified) contexts from one call to the other. Since there is only *one* context in the whole program, all term variables have *only one* type, which gets updated while the program progresses.

A difference between the above program and the one following below is that above a context is a set of statements with term variables as subjects, but below this will be represented by a function from term variables to types. We will freely use the notation we introduced above, so use $\emptyset$ for the empty context, which acts as a function that returns a fresh type variable for every term-variable, and the context $\Gamma, x{:}A$ should be defined like:

$$\begin{array}{lcl} \Gamma, x{:}A \;\; x & = & A \\ \Gamma, x{:}A \;\; y & = & \Gamma\, y, \text{ if } x \neq y \end{array}$$

Also, we should define

$$S \;\; \Gamma \;\; x \; = \; S A, \text{ if } \Gamma\, x = A$$

so we can use $S\Gamma$ as well.

Notice that this gives a type-conflict. We already have that substitutions are of type $\mathcal{T}_C \rightarrow \mathcal{T}_C$; now they are also of type $(\mathcal{V} \rightarrow \mathcal{T}_C) \rightarrow \mathcal{V} \rightarrow \mathcal{T}_C$. These types are not unifiable; in practice, in systems that do not allow for overloaded definitions, this is solved by introducing a separate substitution for contexts.

Definition 2.20 (*pp*$_C$; Alternative definition)

$$\begin{array}{lcl} pp_C \;\; M & = & \langle S\emptyset, A \rangle \\ & & \text{where } \langle S, A \rangle = pt_C \; \emptyset \; M \end{array}$$

$$\begin{array}{lcl} pt_C \;\; \Gamma \;\; x & = & \langle Id_S, \Gamma\, x \rangle \\ pt_C \;\; \Gamma \;\; (\lambda x.M) & = & \langle S, \; S\,(\varphi \rightarrow A) \rangle \\ & & \quad \text{where } \varphi \quad\;\; = \textit{fresh} \\ & & \quad\qquad\;\; \langle S, A \rangle \; = \; pt_C \; (\Gamma, x{:}\varphi) \; M \\ pt_C \;\; \Gamma \;\; MN & = & \langle S_1 \circ S_2 \circ S_3, S_1\, \varphi \rangle \\ & & \quad \text{where } \varphi \quad\;\;\; = \textit{fresh} \\ & & \quad\qquad\;\; \langle S_3, B \rangle \; = \; pt_C \; \Gamma \; M \\ & & \quad\qquad\;\; \langle S_2, A \rangle \; = \; pt_C \; (S_3 \Gamma) \; N \\ & & \quad\qquad\;\; S_1 \qquad = \; \textit{unify} \; (S_2\, B) \; (A \rightarrow \varphi) \end{array}$$

We will see this approach later in Section 5 when we discuss Milner's algorithm $\mathcal{W}$.

We have seen that there exists an algorithm '*pp*$_C$' that, given a term M, produces its principal pair $\langle \Pi, P \rangle$ if M is typeable, and returns *'error'* if it is not[1]. This property expresses that type assignment can be effectively implemented. This program can be used to build a new program that produces the output *'Yes'* if a term is typeable, and *'No'* if it is not; you can *decide* the question with a program, and, therefore, the problem is called decidable.

Because of the decidability of type assignment in this system, it is feasible to use type assignment at compile time (you can wait for an answer, since it exists), and many of the now existing type assignment systems for functional programming languages are therefore based on Curry's system. In fact, both Hindley's initial result on principal types for Combinatory Logic [30], and Milner's seminal paper [40] both are on variants Curry's system. These two papers are considered to form the foundation of types for programming languages, and such systems often are referred to as Hindley-Milner systems.

Every term that is typeable in Curry's system is strongly normalizable. This implies that, although the Lambda Calculus itself is expressive enough to allow all possible programs, when allowing only those terms that are typeable using Curry's system, it is not possible to type non-terminating programs. This is quite a strong restriction that would make it unusable within programming languages, but which is overcome in other systems, that we will discuss later in Section 5.

3 Dealing with polymorphism

In this section, we will show how to extend the Lambda Calculus in such a way that we can express that functions can be applied to inputs of different types.

To illustrate the need for polymorphic procedures, consider the following example. Suppose we have a programming language in which we can write the following program:

$$\begin{array}{l} Ix = x \\ II \end{array}$$

The definition of I is of course a definition for the identity function. In order to find a type for this program, we can translate it directly to the λ-term $(\lambda x . x)\ (\lambda x . x)$ and type that. But then we would be typing the term $\lambda x . x$ *twice*, which seems a waste of effort. We could translate the program to the term $(\lambda a . aa)\ (\lambda x . x)$, but we know we cannot type it. By the principal type property of the previous section, we know that the types we will derive for *both* the occurrences of $\lambda x . x$ will be instances of its *principal type*. So, why not calculate that (done once), take a fresh instance of this type for each occurrence of the term - to guarantee optimal results - and work with those? This is the principle of polymorphism.

The first to introduce this concept in a formal computing setting was R. Milner [40]. Milner's Type Assignment System makes it possible to express that various occurrences of I can have different types, as long as these types are related (by Curry-substitution) to the type derived for the definition of I.

This corresponds to the implementation of type assignment for Λ^{N}, a λ-calculus with *definitions* like $\mathsf{n} = M$, which we will present below. Since the intention of the definitions is to specify each *name* n only once, and to see the occurrences of n in the final term as calls to the body of n, when type checking we would like to be able to associate each call to n to its definition, but without checking the type for n for each call over and over again.

This is used to find types for terms that contain names: each occurrence of a name n in the program can be regarded as an abbreviation of the right-hand side in its definition, and

[1]Notice that, if M is typeable, by the principal pair property, its principal pair exists, and is produced by the algorithm; if M is not typeable, there should be no principal pair, since also that is a valid pair.

therefore the type associated to the occurrence should be an instance of the principal type of the right-hand side term; we associate n to that term, and store the principal type of the right-hand side with n in an *environment*: the type of n is called *generic*, and the function defined by n is called a *polymorphic function*. In fact, we would like to model that each call to n can have a *different* type; we then call such a *name polymorphic*.

3.1 The language Λ^{N}

First, we will look at an extension of the Lambda Calculus, Λ^{N} (*Lambda Calculus with Names*), that enables us to focus on polymorphism by introducing names for λ-terms, and allowing names to be treated as term variables during term construction. The idea is to define a program as a list of *definitions*, followed by a *term* which may contain *calls*.

Definition 3.1 *i*) The syntax for programs in Λ^{N} is defined by:

$$\begin{array}{lcl} M & ::= & x \mid (M_1 \cdot M_2) \mid (\lambda x.M) \\ \mathit{name} & ::= & \text{'}\mathit{string\ of\ characters}\text{'} \\ \mathit{Defs} & ::= & (\mathit{name} = M)\,;\mathit{Defs} \mid \varepsilon \\ \mathit{Term} & ::= & x \mid \mathit{name} \mid (\mathit{Term}_1 \cdot \mathit{Term}_2) \mid (\lambda x.\mathit{Term}) \\ \mathit{Program} & ::= & \mathit{Defs} : \mathit{Term} \end{array}$$

Like before, redundant brackets will be omitted.

ii) Reduction on *terms* in Λ^{N} is defined as normal β-reduction for the Lambda Calculus, extended by $\mathsf{n} \to M$ if $(\mathsf{n} = M)$ appears in the list of definitions.

We put the restriction that in $(\mathsf{n} = M)$, M is a closed λ-term; we could deal with open terms as well, but this would complicate the definitions and results below.

Notice that, in the body of definitions, calls to other definitions are not allowed; we could even restrict bodies of definitions such that no redexes occur there, but that is not necessary for the present purpose.

A particular example of a program in Λ^{N} is:

Example 3.2 (Combinatory Logic (cl) in Λ^{N})

$$\begin{array}{lcll} \mathsf{S} & = & \lambda xyz.xz(yz) & ; \\ \mathsf{K} & = & \lambda xy.x & ; \\ \mathsf{I} & = & \lambda x.x & : \\ T \end{array}$$

where T is a term composed out of S, K, and I.

We define $\mathcal{T}_{\mathrm{CL}}$ as the terms generated by the grammar

$$t ::= x \mid \mathsf{I} \mid \mathsf{K} \mid \mathsf{S} \mid t_1 t_2$$

(this is not the original definition of Combinatory Logic; see Definition 7.3). Observe that abstraction is not used in cl.

Programs written in Λ^{N} can easily be translated to λ-terms; the translation consists of replacing, starting with the final term, all names by their bodies.

Definition 3.3 It is possible to give a natural interpretation of Λ^{N} in Λ.

$$\begin{array}{lll}
\langle x\rangle_\lambda & = x, & \text{for all } x \in \mathcal{V}, \\
\langle \mathsf{n}\rangle_\lambda & = \langle M\rangle_\lambda & \text{if } (\mathsf{n} = M) \text{ appears in the list of definitions.} \\
\langle t_1 t_2\rangle_\lambda & = \langle t_1\rangle_\lambda \langle t_2\rangle_\lambda, & \\
\langle \lambda x.t\rangle_\lambda & = (\lambda x.\langle t\rangle_\lambda), &
\end{array}$$

It is now straightforward to check that reduction is preserved by this interpretation:

Exercise 3.4 Prove that, if $t \to t'$ in Λ^{N}, then $\langle t\rangle_\lambda \twoheadrightarrow_\beta \langle t'\rangle_\lambda$.

3.2 Type assignment for Λ^{N}

In this section we will develop a notion of type assignment on programs in Λ^{N}; in order give the intuition, we first give the definition of type assignment on combinatory terms.

Definition 3.5 (Type Assignment for cl) *Type assignment* on terms in cl is defined by the following natural deduction system.

$$\begin{array}{rl}
(Ax): & \dfrac{}{\Gamma, x{:}A \vdash x : A} \\[2ex]
(\mathsf{S}): & \dfrac{}{\Gamma \vdash \mathsf{S} : (A{\to}B{\to}C){\to}(A{\to}B){\to}A{\to}C} \\[2ex]
(\mathsf{K}): & \dfrac{}{\Gamma \vdash \mathsf{K} : A{\to}B{\to}A} \\[2ex]
(\mathsf{I}): & \dfrac{}{\Gamma \vdash \mathsf{I} : A{\to}A} \\[2ex]
({\to}E): & \dfrac{\Gamma \vdash t_1 : A{\to}B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B}
\end{array}$$

Example 3.6 It is easy to check that the term $\mathsf{S\,K\,I\,I}$ can be assigned the type $\varphi{\to}\varphi$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{S} : B{\to}C{\to}D{\to}\varphi{\to}\varphi} \quad \dfrac{}{\vdash \mathsf{K} : B}}{\vdash \mathsf{S\,K} : C{\to}D{\to}\varphi{\to}\varphi} \quad \dfrac{}{\vdash \mathsf{I} : C}}{\vdash \mathsf{S\,K\,I} : D{\to}\varphi{\to}\varphi} \quad \dfrac{}{\vdash \mathsf{I} : D}}{\vdash \mathsf{S\,K\,I\,I} : \varphi{\to}\varphi}$$

where
$$\begin{array}{rcl}
B & = & (\varphi{\to}\varphi){\to}(\varphi{\to}\varphi){\to}\varphi{\to}\varphi \\
C & = & (\varphi{\to}\varphi){\to}\varphi{\to}\varphi \\
D & = & \varphi{\to}\varphi
\end{array}$$

Notice that the system of Definition 3.5 *only* specifies how to type *terms*; it does not specify how to type a *program* - we need to deal with definitions as well; we will do so below. Notice that each type that can be assigned to each $\mathsf{n} \in \{\mathsf{S},\mathsf{K},\mathsf{I}\}$ using the system above is a *substitution instance* of the *principal type* of the body:

$$\begin{array}{lcl}
pp_{\mathrm{C}}(\lambda xyz.xy(yz)) & = & (\varphi_1{\to}\varphi_2{\to}\varphi_3){\to}(\varphi_1{\to}\varphi_2){\to}\varphi_1{\to}\varphi_3, \\
pp_{\mathrm{C}}(\lambda xy.x) & = & \varphi_4{\to}\varphi_5{\to}\varphi_4, \\
pp_{\mathrm{C}}(\lambda x.x) & = & \varphi_6{\to}\varphi_6.
\end{array}$$

This feature is used when defining how to type a program. We need to type check each definition to make sure that the body of each definition is a typeable term, and store the type found (i.e. the principal type for the body of the definition) in an *environment*. When encountering a call to n, we will force the type for n here to be an instance of the principal type by copying the type in the environment, and allowing unification to change the copy, thereby instantiating it.

The notion of type assignment we will study below for this extended calculus is an extension of Curry's system for LC, and has been studied extensively in the past, e.g. in [30, 26, 8]. Basically, Curry principal types are assigned to named λ-terms. When trying to find a type for the final term in a program, each definition is typed separately. Its right-hand side is treated as a λ-term, and the principal type for the term is derived as discussed above.

That is what the system below specifies.

Of course, when typing a term, we just use the *environment*; in the definition how that *environment* is created is not an issue, we only need to check that the *environment* is sound.

The notion of type assignment defined below uses the notation $\Gamma;\mathcal{E} \vdash M : A$; here Γ is a context, $\mathcal{E}$ an environment, M either a λ-term or a term in *Term*. We also use the notion $\mathcal{E} \vdash Defs : \Diamond$; here $\Diamond$ is not really a type, but just a notation for 'OK'; also, since definitions involve only closed terms, we need not consider contexts there.

Definition 3.7 (Type Assignment) *i*) An *environment* $\mathcal{E}$ is a mapping from *names* to types, similar to contexts; $\mathcal{E},\mathsf{n}{:}A$ is the environment defined as $\mathcal{E} \cup \mathsf{n}{:}A$ where either $\mathsf{n}{:}A \in \mathcal{E}$ or n does not occur in $\mathcal{E}$.

ii) *Type assignment* (with respect to $\mathcal{E}$) is defined by the following natural deduction system. Notice the use of a substitution in rule (*name*).

$$(Ax) : \frac{}{\Gamma,x{:}A;\mathcal{E} \vdash x : A}$$

$$(name) : \frac{}{\Gamma;\mathcal{E},name{:}A \vdash name : S\,A}$$

$$(\to I) : \frac{\Gamma,x{:}A;\mathcal{E} \vdash M : B}{\Gamma;\mathcal{E} \vdash \lambda x.M : A\to B}$$

$$(\to E) : \frac{\Gamma;\mathcal{E} \vdash t_1 : A\to B \quad \Gamma;\mathcal{E} \vdash t_2 : A}{\Gamma;\mathcal{E} \vdash t_1 t_2 : B}$$

$$(\varepsilon) : \frac{}{\mathcal{E} \vdash \varepsilon : \Diamond}$$

$$(Defs) : \frac{\emptyset \vdash_{\mathrm{c}} M : A \qquad \mathcal{E},name{:}A \vdash Defs : \Diamond}{\mathcal{E},name{:}A \vdash (name = M)\,; Defs : \Diamond}$$

$$(Program) : \frac{\mathcal{E} \vdash Defs : \Diamond \qquad \Gamma;\mathcal{E} \vdash M : A}{\Gamma;\mathcal{E} \vdash Defs : Term : A}$$

Notice that, in rule (*Defs*), we use $\vdash_{\mathrm{c}}$ to type the term M, and insist that it is closed by demanding it be typed using an empty context.

For this notion of type assignment, we can easily show the subject reduction result:

Exercise 3.8 If $\Gamma;\mathcal{E} \vdash t : A$, and $t \to t'$, then $\Gamma;\mathcal{E} \vdash t' : A$.

In general, it is assumed that names are defined before they are used; this of course creates problems for recursive definitions, which we will deal with in the next section.

Remark that for CL the *environment* could specify

$$\begin{array}{lcl}
\mathcal{E}\,(\mathsf{S}) & = & (\varphi_1\to\varphi_2\to\varphi_3)\to(\varphi_1\to\varphi_2)\to\varphi_1\to\varphi_3 \\
\mathcal{E}\,(\mathsf{K}) & = & \varphi_4\to\varphi_5\to\varphi_4 \\
\mathcal{E}\,(\mathsf{I}) & = & \varphi_6\to\varphi_6
\end{array}$$

but that the system allows for any valid type for the λ-terms to be added to the environment. Notice that the substition in rule (*name*) is essential: without it, we would not be able to type the term $\mathsf{I}\,\mathsf{I}$.

We will now give an algorithm that, using an environment, checks if the term in a program in Λ^{N} can be typed. Notice that we cannot 'generate' the environment as we do contexts: when dealing with an occurrence of a name, we cannot just assume its type to be a type variable, and hope that unification will construct the needed type. In other words, we can treat term variables as 'local', but have to treat names as 'global'; we therefore have to assume an environment exists when we deal with a name and have to pass it through as a parameter.

As mentioned above, for every definition a pair – consisting of the principal type (found for the right-hand side) and the name of the defined function – is put in the *environment*. Every time n is encountered in a term, the algorithm looks in the environment to see what its (principal) type is. It takes a fresh instance of this type, and uses this instance to find a type for the term; notice that this is only safe (sound) if we demand that the body of the definition is a closed term.

Since the algorithm calls unification which gets applied to types, this could in principle change the type in the environment. By creating a fresh instance we avoid this; we will at most substitute the freshly created type variables, and never those already in the environment. This way we are sure that unifications that take place to calculate the type for one occurrence of n does not affect the type already found for another. Moreover, this way the types actually used for n will always be instances of the principal type that is associated to n in the environment.

The algorithm that calculates the types for a program in Λ^{N} is defined by:

Definition 3.9 (Principal environments and types for Λ^{N})

$$\begin{array}{lcl}
pp_{\Lambda^N}\ x\ \mathcal{E} & = & \langle \{x{:}\varphi\}, \varphi \rangle \\
 & & \text{where } \varphi = \mathit{fresh} \\
pp_{\Lambda^N}\ \mathsf{n}\ \mathcal{E} & = & \langle \emptyset, \mathit{FreshInstance}\,(\mathcal{E}\ \mathsf{n}) \rangle \\
pp_{\Lambda^N}\ (\lambda x.M)\ \mathcal{E} & = & \langle \Gamma, A{\to}B \rangle,\ \text{if } pp_{\Lambda^N}\ M\ \mathcal{E} = \langle \Gamma \cup \{x{:}A\}, B \rangle \\
 & & \langle \Gamma, \varphi{\to}B \rangle,\ \text{if } pp_{\Lambda^N}\ M\ \mathcal{E} = \langle \Gamma, B \rangle \\
 & & \qquad\qquad\quad \text{where } \varphi = \mathit{fresh} \\
pp_{\Lambda^N}\ MN\ \mathcal{E} & = & S_2{\circ}S_1\,\langle \Gamma_1 \cup \Gamma_2, \varphi \rangle \\
 & & \text{where } \begin{array}[t]{lcl} \varphi & = & \mathit{fresh} \\ \langle \Gamma_1, B \rangle & = & pp_{\Lambda^N}\ M\ \mathcal{E} \\ \langle \Gamma_2, A \rangle & = & pp_{\Lambda^N}\ N\ \mathcal{E} \\ S_1 & = & \mathit{unify}\ B\ (A{\to}\varphi) \\ S_2 & = & \mathit{UnifyContexts}\ (S_1\,\Gamma_1)\ (S_1\,\Gamma_2) \end{array} \\
pp_{\Lambda^N}\ (\varepsilon : N)\ \mathcal{E} & = & pp_{\Lambda^N}\ N\ \mathcal{E} \\
pp_{\Lambda^N}\ ((\mathsf{n} = M)\ ;\mathit{Defs} : N)\ \mathcal{E} & = & pp_{\Lambda^N}\ (\mathit{Defs} : N)\ \mathcal{E}, \mathsf{n}{:}A \\
 & & \text{where } \langle \emptyset, A \rangle = pp_{\Lambda^N}\ M\ \mathcal{E}
\end{array}$$

Notice that the environment gets extended by the call to pp_{Λ^N} that types a program by traversing the list of definitions; it adds the principal type of the body of a definition to the name of that definition.

This algorithm is more in common with the practice of programming languages: it type-checks the definitions, builds the *environment*, and calculates the principal type for the final term.

Notice that pp_{Λ^N}, restricted to λ-terms, is exactly pp_{C}, and that then the second argument - the *environment* - becomes obsolete. In fact, since definitions do not contain calls, we could have specified

$$\begin{array}{rcl} pp_{\Lambda^{N}}\ ((\mathsf{n} = M)\ ; \mathit{Defs} : N)\,\mathcal{E} & = & pp_{\Lambda^{N}}\ (\mathit{Defs} : N)\ \ \mathcal{E}, \mathsf{n}{:}A \\ & & \text{where } \langle \emptyset, A \rangle = pp_{\mathrm{C}}\, M \end{array}$$

We leave the soundness of this algorithm as an exercise.

Exercise 3.10 If $pp_{\Lambda^{N}}\ (\mathit{Defs} : N)\ \ \mathcal{E} = \langle \Gamma, A \rangle$, *then* $\Gamma ; \mathcal{E} \vdash N : A$.

4 Dealing with recursion

In this section, we will focus on some aspects of two type assignment systems that were defined for a simple applicative language called $\mathcal{L}_{\mathrm{ML}}$ that is in fact an extended λ-calculus. As above, we will do this be defining a extension of the Lambda Calculus.

4.1 The language Λ^{NR}

First, we will look at an extension of the Lambda Calculus, Λ^{NR}, that enables us to focus on polymorphism and *recursion*, by introducing names for λ-terms and allowing names to occur in all terms, so also in definitions.

Definition 4.1 The syntax for terms in Λ^{NR} is defined by:

$$\begin{array}{lcl} \mathit{name} & ::= & \text{'}\mathit{string\ of\ characters}\text{'} \\ M & ::= & x \mid \mathit{name} \mid (M_1 \cdot M_2) \mid (\lambda x . M) \\ \mathit{Defs} & ::= & (\mathit{name} = M) ; \mathit{Defs} \mid \varepsilon \\ \mathit{Program} & ::= & \mathit{Defs} : M \end{array}$$

Reduction on Λ^{NR}-terms is defined as before for Λ^{N}.

Again, we will assume that M in $(\mathit{name} = M)$ is a closed term. Notice that, with respect to Λ^{N}, by allowing names to appear within the body of definitions, we not only create a dependency between definitions, but also the possibility of *recursive* definitions:

Exercise 4.2 Take the Λ^{NR} *program*

$$\begin{array}{lcll} \mathsf{S} & = & \lambda xyz.xz(yz) & ; \\ \mathsf{K} & = & \lambda xy.x & ; \\ \mathsf{I} & = & \mathsf{SKK} & ; \\ \mathbf{Y} & = & \lambda m.m(\mathbf{Y}m) & : \\ \mathbf{Y}\,\mathsf{I} & & & \end{array}$$

Show that the final term reduces to itself.

4.2 Expressing recursion in LC

Programs written in Λ^{NR} can easily be translated to λ-terms; for non-recursive programs the translation consists of replacing, starting with the final term, all names by their bodies. In case of a recursive definition, we will have to use a fixed-point constructor.

Example 4.3 As an illustration, take the well-known factorial function

$$\begin{aligned} \mathit{Fac}\ n &= 1, & n = 0 \\ \mathit{Fac}\ n &= n \times (\mathit{Fac}\ n{-}1), & \text{otherwise} \end{aligned}$$

In order to create the Λ term that represents this function, we first observe that we could write it in Λ^{NR} as:

$$\mathit{Fac} = \lambda n.(\mathit{Cond}\ (n=0)\ 1\ (n \times (\mathit{Fac}\ n{-}1))),$$

Now, since this last definition for *Fac* is recursive, we need the fixed-point operator to define it in Λ. Remember that we know that (where $\mathbf{Y} = \lambda f.(\lambda x.f(xx))\ (\lambda x.f(xx))$)

$$\mathbf{Y}M =_\beta M(\mathbf{Y}M)$$

Now, if $M = \lambda x.N$, this becomes

$$\mathbf{Y}(\lambda x.N) =_\beta (\lambda x.N)\ (\mathbf{Y}(\lambda x.N)) \to_\beta N[\mathbf{Y}(\lambda x.N)/x]$$

This gives us an equation like:

$$F = N[F/x]$$

with solution $F = \mathbf{Y}(\lambda x.N)$. So, in general, when we have an equation like $F = C[F]$, where $C[\ ]$ is a term with a hole (so $C[F]$ is a notation for a term in which F occurs) then we can write

$$F = C[F] = (\lambda f.C[f])F = \mathbf{Y}(\lambda f.C[f])$$

In case of our factorial function, the term we look for then becomes:

$$\mathit{Fac} = \mathbf{Y}(\lambda fn.\mathit{Cond}\ (n=0)\ 1\ (n \times (f\ (n-1))))$$

Notice that this approach is going to cause a problem when we want to type terms in Λ^{NR}: we cannot just translate the term, type it in $\vdash_{\mathrm{c}}$ and give that type to the original Λ^{NR} term, since, for recursive terms, this would involve typeing $\mathbf{Y}$. This is impossible.

Instead, a more ad-hoc approach is used: rather than trying to type a term containing self-application, an explicit construction for recursion is added. Take a look at the example above. We know that *num→num* should be the type for *Fac*, and that

$$\mathit{Fac} = \lambda n.\mathit{Cond}\ (n=0)\ 1\ (n \times (\mathit{Fac}\ n{-}1))$$

so *num→num* should also be the type for

$$\lambda n.\mathit{Cond}\ (n=0)\ 1\ (n \times (\mathit{Fac}\ n{-}1))$$

Notice that the occurrence of *Fac* in this term has type *num→num* as well. Therefore,

$$\lambda gn.\mathit{Cond}\ (n=0)\ 1\ (n \times (g\ (n-1)))$$

should have type $(\mathit{num}\to\mathit{num})\to\mathit{num}\to\mathit{num}$. These observations together imply that a desired type for $\lambda f.(\lambda x.f(xx))\ (\lambda x.f(xx))$ to serve in this case would be

$$((\mathit{num}\to\mathit{num})\to\mathit{num}\to\mathit{num})\to\mathit{num}\to\mathit{num}.$$

Example 4.4 We can generalise the observation of the previous example, and deduce that, in order to type both $\mathbf{Y}M$ and $M(\mathbf{Y}M)$ with the same type, we need to assume that $\mathbf{Y}$ has type $(A \to A) \to A$, for all A; so when enforcing typeable recursion for the λ-calculus, we would need to extend the language, the reduction rules, and the type assigment as follows:

$$M,N ::= \cdots \mid \mathbf{Y}$$

$$\mathbf{Y}M \rightarrow M(\mathbf{Y}M)$$

$$\frac{}{\Gamma \vdash_{c} \mathbf{Y} : (A \rightarrow A) \rightarrow A}$$

This extension is enough to encode all Λ^{NR} programs.

So it suffices to add a *recursive definition* Y to the language, defined by $Y ::= \lambda m.m(Ym)$, that gets the type $(\varphi{\rightarrow}\varphi){\rightarrow}\varphi$ in the environment. When we will discuss ML in the next section, we add a new *language construct*, i.e. a second kind of abstraction, $\mathsf{Fix}\ g.E$, but the way to type this is essentially the same.

4.3 Type assignment and algorithms

Naturally, the notion of type assignment for Λ^{NR} is an extension of that of Λ^{N}.

Definition 4.5 (Type assignment for Λ^{NR})

$$(Ax) : \frac{}{\Gamma, x{:}A; \mathcal{E} \vdash x : A}$$

$$(name) : \frac{}{\Gamma; \mathcal{E}, name{:}A \vdash name : S\,A}\ (\textit{not a recursive call})$$

$$(rec\ name) : \frac{}{\Gamma; \mathcal{E}, name{:}A \vdash name : A}\ (\textit{recursive call})$$

$$({\rightarrow}I) : \frac{\Gamma, x{:}A; \mathcal{E} \vdash M : B}{\Gamma; \mathcal{E} \vdash \lambda x.M : A{\rightarrow}B}$$

$$({\rightarrow}E) : \frac{\Gamma; \mathcal{E} \vdash t_1 : A{\rightarrow}B \quad \Gamma; \mathcal{E} \vdash t_2 : A}{\Gamma; \mathcal{E} \vdash t_1 t_2 : B}$$

$$(\varepsilon) : \frac{}{\mathcal{E} \vdash \varepsilon : \Diamond}$$

$$(Defs) : \frac{\emptyset; \mathcal{E}, name{:}A \vdash M : A \qquad \mathcal{E}, name{:}A \vdash Defs : \Diamond}{\mathcal{E}, name{:}A \vdash (name = M)\,; Defs : \Diamond}$$

$$(Program) : \frac{\mathcal{E} \vdash Defs : \Diamond \qquad \Gamma; \mathcal{E} \vdash M : A}{\Gamma; \mathcal{E} \vdash Defs : Term : A}$$

Exercise 4.6 Take P to be the program in Exercise 4.2. Find an enviroment $\mathcal{E}$ such that $\emptyset;\mathcal{E} \vdash P : \phi$, and build the derivation that justifies this judgement.

Notice that the main difference between this and the notion we defined for Λ^{N} lies in rule (*Defs*); since we now allow names to appear inside the body of definitions, we can no longer type the body as a pure λ-term. To make sure that the type derived for a recursive function is the same as for its recursive calls inside the bodies, we insist on $\emptyset;\mathcal{E},name{:}A \vdash M : A$ as a premise, not just $\emptyset;\mathcal{E} \vdash M : A$; this is paired with the presence of rule (*rec name*), which enforces that all recursive calls are typed with exactly the environment type.

When looking to type a program, we need a type for every name we encounter in the environment, so assume that definitions appear before their use; with recursive definitions, this creates an obvious problem. We solve this as follows. Assume $\mathsf{n} = M$ is a recursive definition. When constructing the type of M, we assume n has a type variable as type; this can be changed by unification into a more complex type depending on the context of the (recursive) call. At the end of the analysis of the body, we will have constructed a certain type A for n, and need to check that the type produced for the body is unifiable with A; if all steps are successful, this will produce the correct type, both for M as for n.

Note that only for recursion we need to modify a type for a name in the environment; all other types created for names a simply stored in the environment.

We now give the principal type algorithm for Λ^{NR}; notice that, since the environment gets changed for recursive definitions, it is also returned as result.

Definition 4.7 (Principal environments and types for Λ^{NR})

$$\begin{array}{lll}
pp_{\Lambda^{\mathsf{NR}}}\ x\ \mathcal{E} & = & \langle \{x{:}\varphi\},\varphi,\mathcal{E}\rangle \\
 & & \text{where } \varphi = \mathit{fresh} \\
pp_{\Lambda^{\mathsf{NR}}}\ \mathsf{n}\ \mathcal{E} & = & \langle \emptyset, A, \mathcal{E}\rangle,\ \mathsf{n} \text{ is recursive and depends on this occurrence} \\
 & & \langle \emptyset, \mathit{FreshInstance}\, A, \mathcal{E}\rangle,\ \text{otherwise} \\
 & & \qquad\qquad \text{where } \mathsf{n}{:}A \in \mathcal{E} \\
pp_{\Lambda^{\mathsf{NR}}}\ (\lambda x.M)\ \mathcal{E} & = & \langle \Gamma, A{\to}B, \mathcal{E}'\rangle,\ \text{if } pp_{\Lambda^{\mathsf{NR}}}\ M\ \mathcal{E} = \langle \Gamma \cup \{x{:}A\}, B, \mathcal{E}'\rangle \\
 & & \langle \Gamma, \varphi{\to}B, \mathcal{E}'\rangle,\ \text{if } pp_{\Lambda^{\mathsf{NR}}}\ M\ \mathcal{E} = \langle \Gamma, B, \mathcal{E}'\rangle \\
 & & \qquad\qquad \text{where } \varphi = \mathit{fresh} \\
pp_{\Lambda^{\mathsf{NR}}}\ (MN)\ \mathcal{E} & = & S_2{\circ}S_1\,(\langle (\Gamma_1 \cup \Gamma_2), \varphi, \mathcal{E}''\rangle) \\
 & & \text{where } \begin{array}[t]{lcl} \varphi & = & \mathit{fresh} \\ \langle \Gamma_1, B, \mathcal{E}'\rangle & = & pp_{\Lambda^{\mathsf{NR}}}\ M\ \mathcal{E} \\ \langle \Gamma_2, A, \mathcal{E}''\rangle & = & pp_{\Lambda^{\mathsf{NR}}}\ N\ \mathcal{E}' \\ S_1 & = & \mathit{unify}\ B\ (A{\to}\varphi) \\ S_2 & = & \mathit{UnifyContexts}\ (S_1\,\Gamma_0)\ (S_1\,\Gamma_1) \end{array} \\
pp_{\Lambda^{\mathsf{NR}}}\ (\mathsf{n} = M; \mathit{Defs} : N)\ \mathcal{E} & = & pp_{\Lambda^{\mathsf{NR}}}\ (\mathit{Defs} : N)\ \mathcal{E}, \mathsf{n}{:}S\,(A),\ \text{otherwise} \\
 & & \text{where } \begin{array}[t]{lcl} \varphi & = & \mathit{fresh} \\ \langle \Gamma, A, \mathcal{E}'\rangle & = & pp_{\Lambda^{\mathsf{NR}}}\ M\ \mathcal{E}, \mathsf{n}{:}\varphi \\ S & = & \mathit{unify}\ A\ \mathcal{E}'(\mathsf{n}) \end{array} \\
pp_{\Lambda^{\mathsf{NR}}}\ (\varepsilon : N)\ \mathcal{E} & = & pp_{\Lambda^{\mathsf{NR}}}\ N\ \mathcal{E}
\end{array}$$

Notice that, in the case for application, $S_2{\circ}S_1$ gets applied to $\mathcal{E}$, producing a changed environment; this is only relevant when typing a recursive definition. Again, we assume that the body of a definition is a closed term.

As before in Definition 2.20, we can give an alternative algorithm that does not need to unify contexts; since it changes also the environment, that is not returned as result.

Definition 4.8 (*Milner*)

$$\begin{array}{lll}
\mathit{Milner}\ \Gamma\ x\ \mathcal{E} & = & \langle \mathit{Id}_S, \Gamma\, x\rangle \\
\mathit{Milner}\ \Gamma\ \mathsf{n}\ \mathcal{E} & = & \langle \mathit{Id}_S, \mathcal{E}\,\mathsf{n}\rangle,\ \mathsf{n} \text{ is recursive and depends on this occurrence} \\
 & & \langle \mathit{Id}_S, \mathit{FreshInstance}\ (\mathcal{E}\,\mathsf{n})\rangle,\ \text{otherwise} \\
\mathit{Milner}\ \Gamma\ (\lambda x.M)\ \mathcal{E} & = & \langle S,\ S\,(\varphi{\to}C)\rangle \\
 & & \text{where } \begin{array}[t]{lcl} \varphi & = & \mathit{fresh} \\ \langle S, C\rangle & = & \mathit{Milner}\ (\Gamma, x{:}\varphi)\ M\ \mathcal{E} \end{array} \\
\mathit{Milner}\ \Gamma\ (MN)\ \mathcal{E} & = & \langle S_3{\circ}S_2{\circ}S_1,\ S_3\,\varphi\rangle \\
 & & \text{where } \begin{array}[t]{lcl} \varphi & = & \mathit{fresh} \\ \langle S_1, C\rangle & = & \mathit{Milner}\ \Gamma\ M\ \mathcal{E} \\ \langle S_2, A\rangle & = & \mathit{Milner}\ (S_1\,\Gamma)\ (S_1\,\mathcal{E})\ N \\ S_3 & = & \mathit{unify}\ (S_2\,C)\ (A{\to}\varphi) \end{array}
\end{array}$$

$$\begin{array}{lcl}
\mathit{FindType}\ \langle \varepsilon : N\rangle\ \mathcal{E} & = & \langle S\emptyset, A\rangle, \\
& & \text{where } \langle S,A\rangle = \mathit{Milner}\ \emptyset\ N\ \mathcal{E} \\
\mathit{FindType}\ \langle \mathsf{n} = M; \mathit{Defs} : N\rangle\ \mathcal{E} & = & \mathit{FindType}\ \langle \mathit{Defs}, N\rangle\ \mathcal{E}, \mathsf{n}{:}A, \quad \mathsf{n}\text{ not recursive} \\
& & \text{where } \langle S,A\rangle = \mathit{Milner}\ \emptyset\ M\ \mathcal{E} \\
& = & \mathit{FindType}\ \langle \mathit{Defs}, N\rangle\ \mathcal{E}, \mathsf{n}{:}S_1\,A],\ \text{otherwise} \\
& & \text{where } \begin{array}[t]{lcl} \varphi & = & \mathit{fresh} \\ \langle S_2, A\rangle & = & \mathit{Milner}\ \emptyset\ M\ \mathcal{E}, \mathsf{n}{:}\varphi \\ S_1 & = & \mathit{unify}\ (S_2\,\varphi)\ A \end{array}
\end{array}$$

5 Milner's ML

In [40], a formal type discipline was presented for polymorphic procedures in a simple programming language called $\mathcal{L}_{\text{ML}}$, designed to express that certain procedures work well on objects of a wide variety. This kind of procedures is called *(shallow) polymorphic*, and they are essential to obtain enough flexibility in programming.

$\mathcal{L}_{\text{ML}}$ is based on LC, but adds two syntactical constructs: one that expresses that a sub-term can be used in different ways, and one that expresses recursion; this is paired with a type assignment system that accurately deals with these new constructs. In [23] an algorithm $\mathcal{W}$ was defined that has become very famous and is implemented in a type checker that is embedded in the functional programming language ML. $\mathcal{W}$ is shown to be semantically sound (based on a formal semantics for the language [40] – so typed programs cannot go *wrong*), and syntactically sound, so if $\mathcal{W}$ accepts a program, then it is well typed.

We will also present a variant as defined by A. Mycroft [43], which is a generalisation of Milner's system, by allowing a more permissive rule for recursion. Both systems are present in the implementation of the functional programming languages Miranda [51] and Haskell [32]. Milner's system is used when the type assignment algorithm infers a type for an object; Mycroft's system is used when the type assignment algorithm does type checking, i.e. when the programmer has specified a type for an object.

5.1 The ML Type Assignment System

In this subsection, we present Milner's Type Assignment System as was done in [24], and not as in [23, 40], because the former presentation is more detailed and clearer.

Definition 5.1 (ML EXPRESSIONS) *i*) ML *expressions* are defined by the grammar:

$$E ::= x \mid (\lambda x.E) \mid (E_1 \cdot E_2) \mid (\mathsf{let}\ x = E_1\ \mathtt{in}\ E_2) \mid (\mathsf{Fix}\ g.E)$$

As before, we will economise on brackets.

ii) The notion of reduction on $\mathcal{L}_{\text{ML}}$, $\rightarrow_{\text{ML}}$, is defined as $\rightarrow_\beta$, extended by:

$$\begin{array}{rcl}
(\mathsf{let}\ x = E_1\ \mathtt{in}\ E_2) & \rightarrow_{\text{ML}} & E_2[E_1/x] \\
\mathsf{Fix}\ g.E & \rightarrow_{\text{ML}} & E\,[(\mathsf{Fix}\ g.E)/g]
\end{array}$$

With this extended notion of reduction, the terms $(\mathsf{let}\ x = E_2\ \mathtt{in}\ E_1)$ and $(\lambda x.E_1)E_2$ are denotations for reducible expressions (redexes) that both reduce to the term $E_1[E_2/x]$. However, the semantic interpretation of these terms is different. The term $(\lambda x.E_1)E_2$ is interpreted as a function with an operand, whereas the term $(\mathsf{let}\ x = E_2\ \mathtt{in}\ E_1)$ is interpreted as the term $E_1[E_2/x]$ would be interpreted. This difference is reflected in the way the type assignment system treats these terms.

In fact, the let-construct is added to ML to cover precisely those cases in which the term $(\lambda x.E_1)E_2$ is not typeable, but the contraction $E_1[E_2/x]$ is, while it is desirable for the term $(\lambda x.E_1)E_2$ to be typeable. The problem to overcome is that, in assigning a type to $(\lambda x.E_1)E_2$, the term-variable x can only be typed with *one* Curry-type; this is not required for x in $(\mathsf{let}\ x = E_2\ \mathtt{in}\ E_1)$. As argued in [23], it is of course possible to set up type assignment in such a way that $E_1[E_2/x]$ is typed every time the let-construct is encountered, but that would force us to type E_2 perhaps many times; even though in normal implementations E_2 is shared, the various references to it could require it to have different types. The elegance of the let-construct is that E_2 is typed only *once*, and that its (generalised) principal type is used when typing $E_1[E_2/x]$.

The language defined in [40] also contains a conditional-structure (if-then-else). It is not present in the definition of $\mathcal{L}_{\text{ML}}$ in [23], so we have omitted it here. The construct Fix is introduced to model recursion; it is present in the definition of $\mathcal{L}_{\text{ML}}$ in [40], but not in [23]. Since it plays a part in the extension defined by Mycroft of this system, we have inserted it here. Notice that Fix is not a combinator, but an other abstraction mechanism, like $(\lambda\cdot.\cdot)$.

The set of ML *types* is defined much in the spirit of Curry types, ranged over by A,B; these types can be quantified, creating *generic* types, ranged over by ϕ,ψ. An ML-*substitution* on types is defined like a Curry-substitution as the replacement of type variables by types, as before. ML-substitution on a type-scheme ψ is defined as the replacement of *free* type variables by renaming the generic type variables of A if necessary.

Definition 5.2 ([40]) *i*) The set of ML *types* is defined in two layers.

$$\begin{array}{rcl} A,B & ::= & \varphi \mid c \mid (A \to B) \\ \phi,\psi & ::= & A \mid (\forall\varphi.\psi) \end{array}$$

We call types of the shape $\forall\varphi.\psi$ *quantified* or *polymorphic* types. We will omit brackets as before, and abbreviate $(\forall\varphi_1.(\forall\varphi_2.\cdots(\forall\varphi_n.A)\cdots))$ by $\forall\overrightarrow{\varphi}.A$.

ii) An ML-*substitution* on types is defined by:

$$\begin{array}{lcll} (\varphi\mapsto C)\ \varphi & = & C & \\ (\varphi\mapsto C)\ c & = & c & \\ (\varphi\mapsto C)\ \varphi' & = & \varphi', & \text{if } \varphi' \neq \varphi \\ (\varphi\mapsto C)\ A\to B & = & ((\varphi\mapsto C)\ A) \to ((\varphi\mapsto C)\ B) & \\ (\varphi\mapsto C)\ \forall\varphi'.\psi & = & \forall\varphi'.((\varphi\mapsto C)\ \psi), & \text{if } \varphi \neq \varphi'\ \&\ \varphi' \notin fv(C). \end{array}$$

As is the case for λ-terms, we keep names of *bound* and *free* type variables separate, where we see φ as bound in $\forall\varphi.\psi$, and a type variable is free in ψ if it occurs in that type, and is not bound. We also see φ as bound by $(\varphi\mapsto C)$, so can safely assume that, in $(\varphi\mapsto C)\ \forall\varphi'.\psi$, $\varphi \neq \varphi'$.

iii) A type obtained from another by mere substitution is called an *instance*.

iv) We define the relation '$\succeq$' as the least reflexive and transitive relation such that:

$$\forall\overrightarrow{\varphi}.\psi \succeq \forall\overrightarrow{\varphi'}.\psi[\overrightarrow{B/\varphi}].$$

provided no φ' is free in ψ. If $\phi \succeq \psi$, we call ψ a *generic instance* of ϕ.

v) We define the relation '$\leq$' as the least pre-order such that:

$$\begin{array}{rcl} \psi & \leq & \forall\varphi.\psi, \\ \forall\varphi.\psi & \leq & \psi[B/\varphi]. \end{array}$$

Notice that we need to consider types also modulo some kind of α-conversion, in order to avoid binding of free type variables while substituting; from now on, we will do that.

Notice that the set of ML-types is the set of Curry-types, extended with type constants. Notice also that if $B = \forall\varphi_1 \cdots \forall\varphi_n.A$, then the set of type variables occurring in A is not necessarily equal to $\{\varphi_1, \ldots, \varphi_n\}$.

Exercise 5.3 If $A \leq B$, then there exists a substitution S such that $S\,A = B$.

We now define the closure of a type with respect to a context:

Definition 5.4 $\overline{\Gamma}\ A = \forall\overrightarrow{\varphi}.A$ where $\overrightarrow{\varphi}$ are the type variables that occur free in A but not in Γ.

In the following definition, we show derivation rules for Milner's system as presented in [24]. (In [40] there are no derivation rules; instead, a rather complicated definition of 'well typed prefixed expressions' is given.)

Definition 5.5 ([24]) ML-*type assignment* and ML-*derivations* are defined by the following deduction system.

$$(Ax):\ \dfrac{}{\Gamma \vdash_{\mathrm{ML}} x : \psi}\ (x{:}\psi \in \Gamma)$$

$$(\to I):\ \dfrac{\Gamma, x{:}A \vdash_{\mathrm{ML}} E : B}{\Gamma \vdash_{\mathrm{ML}} \lambda x.E : A \to B}$$

$$(\to E):\ \dfrac{\Gamma \vdash_{\mathrm{ML}} E_1 : A \to B \quad \Gamma \vdash_{\mathrm{ML}} E_2 : A}{\Gamma \vdash_{\mathrm{ML}} E_1 E_2 : B}$$

$$(\mathsf{let}):\ \dfrac{\Gamma \vdash_{\mathrm{ML}} E_1 : \psi \quad \Gamma, x{:}\psi \vdash_{\mathrm{ML}} E_2 : B}{\Gamma \vdash_{\mathrm{ML}} \mathsf{let}\ x = E_1\ \mathtt{in}\ E_2 : B}$$

$$(\mathsf{Fix}):\ \dfrac{\Gamma, g{:}A \vdash_{\mathrm{ML}} E : A}{\Gamma \vdash_{\mathrm{ML}} \mathsf{Fix}\ g.E : A}$$

$$(\forall I):\ \dfrac{\Gamma \vdash_{\mathrm{ML}} E : \psi}{\Gamma \vdash_{\mathrm{ML}} E : \forall\varphi.\psi}\ (\varphi \text{ not free in } \Gamma)$$

$$(\forall E):\ \dfrac{\Gamma \vdash_{\mathrm{ML}} E : \forall\varphi.\psi}{\Gamma \vdash_{\mathrm{ML}} E : \psi[A/\varphi]}$$

The let-construct corresponds in a way to the use of *definitions* in Λ^{NR}; notice that we could represent $\mathsf{n} = N : M$ by $\mathsf{let}\ \mathsf{n} = N\ \mathtt{in}\ M$.

Example 5.6 The program '$I = \lambda x.x : II$' translates as '$\mathsf{let}\ i = \lambda x.x\ \mathtt{in}\ ii$' which we can type by:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}\varphi \vdash_{\mathrm{ML}} x : \varphi}\,(Ax)}{\emptyset \vdash_{\mathrm{ML}} \lambda x.x : \varphi\to\varphi}\,(\to I)}{\emptyset \vdash_{\mathrm{ML}} \lambda x.x : \forall\varphi.\varphi\to\varphi}\,(\forall I) \qquad \dfrac{\dfrac{\dfrac{}{i{:}\forall\varphi.\varphi\to\varphi \vdash_{\mathrm{ML}} i : \forall\varphi.\varphi\to\varphi}}{i{:}\forall\varphi.\varphi\to\varphi \vdash_{\mathrm{ML}} i : (A\to A)\to A\to A} \quad \dfrac{\dfrac{}{i{:}\forall\varphi.\varphi\to\varphi \vdash_{\mathrm{ML}} i : \forall\varphi.\varphi\to\varphi}\,(Ax)}{i{:}\forall\varphi.\varphi\to\varphi \vdash_{\mathrm{ML}} i : A\to A}\,(\forall E)}{x{:}\forall\varphi.\varphi\to\varphi \vdash_{\mathrm{ML}} ii : A\to A}\,(\to E)}{\emptyset \vdash_{\mathrm{ML}} \mathsf{let}\ i = \lambda x.x\ \mathtt{in}\ ii : A\to A}\,(\mathsf{let})$$

But let is more general than that. First of all, a let-expression can occur at any point in the ML-term, not just on the outside, and, more significantly, N need not be a closed term. In ML it is possible to define a term that is *partially polymorphic*, i.e. has a type like $\forall\overrightarrow{\varphi}.A$, where A has also *free* type variables. Notice that, when applying rule $(\forall I)$, we only need to check if the type variable we are trying to bind does not occur in the context; this can generate a derivation for $\Gamma \vdash_{\text{ML}} M : \forall\overrightarrow{\varphi}.A$, where the free type variables in A are those occurring in Γ.

The ML notion of type assignment, when restricted to the pure Lambda Calculus, is also a restriction of the Polymorphic Type Discipline, or System F, as presented in [29]. This system is obtained from Curry's system by adding the type constructor '$\forall$': if φ is a type variable and A is a type, then $\forall\varphi.A$ is a type. A difference between the types created in this way and the types (or type-schemes) of Milner's system is that in Milner's type-schemes the $\forall$-symbol can occur only at the outside of a type (so polymorphism is *shallow*); in System F, $\forall$ is a general type constructor, so $A{\rightarrow}\forall\varphi.B$ is a type in that system.

In understanding the (let)-rule, notice that the generic type ψ is used. Assume that $\psi = \forall\overrightarrow{\varphi}.A$, and that in building the derivation for the statement $E_2{:}B$, ψ is instantiated (otherwise the rules $({\rightarrow}I)$ and $({\rightarrow}E)$ cannot be used) into the types $A_1,\ldots,A_n$, so, for every A_i there exists are $\overrightarrow{B}$ such that $A_i = A[\overrightarrow{B/\varphi}]$. So, for every A_i there is a substitution S_i such that $S_i\,A = A_i$. Assume, without loss of generality, that $E_1{:}\psi$ is obtained from $E_1{:}A$ by applying the rule $(\forall I)$. Notice that the types actually used for x in the derivation for $E_2{:}B$ are, therefore, substitution instances of the type derived for E_1.

In fact, this is the only true use of quantification of types in ML: although the rules allow for a lot more, essentially quantification serves to enable polymorphism:

$$\dfrac{\dfrac{\dfrac{\bigtriangledown}{\Gamma \vdash_{\text{ML}} E_1 : A}}{\emptyset \vdash_{\text{ML}} E_1 : \forall\varphi.A}\,(\forall I) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x{:}\forall\varphi.A \vdash_{\text{ML}} x : \forall\varphi.A}\,(Ax)}{x{:}\forall\varphi.A \vdash_{\text{ML}} x : A[B/\varphi]}\,(\forall E) \quad \dfrac{\dfrac{}{x{:}\forall\varphi.A \vdash_{\text{ML}} x : \forall\varphi.A}\,(Ax)}{x{:}\forall\varphi.A \vdash_{\text{ML}} x : A[C/\varphi]}\,(\forall E)}{\bigtriangledown}}{x{:}\forall\varphi.\varphi\rightarrow\varphi \vdash_{\text{ML}} E_2 : D}}{\emptyset \vdash_{\text{ML}} \textsf{let}\ x = E_1\ \texttt{in}\ E_2 : D}\,(\textsf{let})$$

Since we can show the Substitution Lemma also for ML, we can type the contraction of the redex as follows (notice that quantification is no longer used):

$$\dfrac{\dfrac{\dfrac{\bigtriangledown}{\Gamma \vdash_{\text{ML}} E_1 : A[B/\varphi]} \qquad \dfrac{\bigtriangledown}{\Gamma \vdash_{\text{ML}} E_1 : A[C/\varphi]}}{\bigtriangledown}}{x{:}\forall\varphi.\varphi\rightarrow\varphi \vdash_{\text{ML}} E_2[E_1/x] : D}$$

As for rule $(\textsf{Fix})$, remember that, to express recursion, we look for a solution to an equation like: $F = N[F/x]$ which has as solution $F = \mathbf{Y}(\lambda x.N)$. One way of dealing with this, and the approach of [23], is to add the constant $\mathbf{Y}$ to the calculus as discussed in Example 4.4.

Instead, here we follow the approach of [24] and add recursion via additional syntax. Since, by the reasoning above, we normally are only interested in fixed-points of *abstractions*, in some papers this has led the definition of $\textsf{Fix}\ g\,x.E$ as general fixed-point constructor, which would correspond to our $\textsf{Fix}\ g.\lambda x.E$; the rule then is formulated as follows:

$$(\textsf{Fix}) : \dfrac{\Gamma, g{:}A{\rightarrow}B, x{:}A \vdash_{\text{ML}} E : B}{\Gamma \vdash_{\text{ML}} \textsf{Fix}\ g\,x.E : A{\rightarrow}B}$$

This is, for example, the approach of [44].

Another approach is the use of `letrec`, a combination of let and Fix, of the form

$\texttt{letrec}\ g = \lambda x.E_1\ \texttt{in}\ E_2$

with derivation rule

$$(\texttt{letrec}) : \dfrac{\Gamma, g{:}B{\rightarrow}C, x{:}B \vdash_{\text{ML}} E_1 : C \qquad \Gamma, g{:}\psi \vdash_{\text{ML}} E_2 : A}{\Gamma \vdash_{\text{ML}} \texttt{letrec}\ g = \lambda x.E_1\ \texttt{in}\ E_2 : A}\ (\psi = \overline{\Gamma}(B{\rightarrow}C))$$

This construct `letrec` can be viewed as syntactic sugar for

$$\mathsf{let}\ h = (\mathsf{Fix}\ g.\lambda x.E_1)\ \mathtt{in}\ E_2[h/g]$$

For the above defined notion of type assignment, we have the following result:

Exercise 5.7 (Generation Lemma) *i) If $\Gamma \vdash_{\mathrm{ML}} x : \chi$, then there exists $x{:}\psi \in \Gamma$ such that $\psi \leq \chi$.*

ii) If $\Gamma \vdash_{\mathrm{ML}} E_1E_2 : \chi$, then there exist A,B such that $\Gamma \vdash_{\mathrm{ML}} E_1 : A{\rightarrow}B$, $\Gamma \vdash_{\mathrm{ML}} E_2 : A$, and $\chi = \forall\overrightarrow{\varphi_i}.B$, with each φ_i not in Γ.

iii) If $\Gamma \vdash_{\mathrm{ML}} \lambda x.E : \chi$, then there exist A,B such that $\Gamma,x{:}A \vdash_{\mathrm{ML}} E : B$, and $\chi = \forall\overrightarrow{\varphi_i}.A{\rightarrow}B$, with each φ_i not in Γ.

iv) If $\Gamma \vdash_{\mathrm{ML}} \mathsf{Fix}\ g.E : \chi$, then there exists A such that $\Gamma,g{:}A \vdash_{\mathrm{ML}} E : A$, and $\chi = \forall\overrightarrow{\varphi_i}.A$, with each φ_i not in Γ.

v) If $\Gamma \vdash_{\mathrm{ML}} \mathsf{let}\ x = E_1\ \mathtt{in}\ E_2 : \chi$, then there exists A,ψ such that $\Gamma,x{:}\psi \vdash_{\mathrm{ML}} E_1 : A$, and $\Gamma \vdash_{\mathrm{ML}} E_2 : \psi$, and $\chi = \forall\overrightarrow{\varphi_i}.A$, with each φ_i not in Γ.

Notice that we do not have that $\Gamma \vdash_{\mathrm{ML}} E : \chi$ and $\chi \leq \psi$ imply $\Gamma \vdash_{\mathrm{ML}} E : \psi$.

The subject reduction property holds also for ML:

Exercise 5.8 If $\Gamma \vdash_{\mathrm{ML}} M : A$, and $M \rightarrow N$, then $\Gamma \vdash_{\mathrm{ML}} N : A$

We will now define Milners' algorithm $\mathcal{W}$. Notice that, different from the algorithms we considered above, $\mathcal{W}$ does not distinguish names and variables, so deals only with a context. Above we needed to pass the environment as a parameter; this was mainly because we could not safely assume a type for names or depend on unification to construct the correct type. A similar thing is true for let-bound variables: these might need to have a quantified type, which does not get constructed by $\mathcal{W}$; so, for the same reason, $\mathcal{W}$ passes the context as a parameter, which should have the correct type for variables to make the term typeable.

Definition 5.9 (Milner's Algorithm $\mathcal{W}$)

$$\begin{array}{lll}
\mathcal{W}\ \Gamma\ x & = & \langle id, B\rangle \\
 & & \text{where } \begin{array}[t]{lcl} x{:}\forall\vec{\varphi}.A & \in & \Gamma \\ B & = & A[\overrightarrow{\varphi'/\varphi}] \\ \text{all } \varphi' & = & \mathit{fresh} \end{array} \\
\mathcal{W}\ \Gamma\ (\lambda x.E) & = & \langle S_1,\ S_1(\varphi{\rightarrow}A)\rangle \\
 & & \text{where } \begin{array}[t]{lcl} \langle S_1,A\rangle & = & \mathcal{W}\ (\Gamma,x{:}\varphi)\ E \\ \varphi & = & \mathit{fresh} \end{array} \\
\mathcal{W}\ \Gamma\ (E_1E_2) & = & \langle S_3{\circ}S_2{\circ}S_1, S_3\,\varphi\rangle \\
 & & \text{where } \begin{array}[t]{lcl} \varphi & = & \mathit{fresh} \\ \langle S_1,A\rangle & = & \mathcal{W}\ \Gamma\ E_1 \\ \langle S_2,B\rangle & = & \mathcal{W}\ (S_1\Gamma)\ E_2 \\ S_3 & = & \mathit{unify}\ (S_2\,A)\ (B{\rightarrow}\varphi) \end{array} \\
\mathcal{W}\ \Gamma\ (\mathsf{let}\ x = E_1\ \mathtt{in}\ E_2) & = & \langle S_2{\circ}S_1, B\rangle \\
 & & \text{where } \begin{array}[t]{lcl} \langle S_1,A\rangle & = & \mathcal{W}\ \Gamma\ E_1 \\ \langle S_2,B\rangle & = & \mathcal{W}\ (S_1\Gamma, x{:}\overline{S_1\Gamma\,A})\ E_2 \end{array} \\
\mathcal{W}\ \Gamma\ (\mathsf{Fix}\ g.E) & = & \langle S_2{\circ}S_1, S_2\,A\rangle \\
 & & \text{where } \begin{array}[t]{lcl} \langle S_1,A\rangle & = & \mathcal{W}\ (\Gamma,g{:}\varphi)\ E \\ S_2 & = & \mathit{unify}\ (S_1\varphi)\ A \\ \varphi & = & \mathit{fresh} \end{array}
\end{array}$$

Notice the use of $\overline{S_1\,\Gamma}\,A$ in the case for let, where we add a quantified type for x to the context.

This system has several important properties:

- The system has a principal type property, in that, given Γ and E, there exists a principal type, calculated by $\mathcal{W}$. It does not enjoy the principal *pair* property, as argued in [53]. This is essentially due to the fact that, when a derivation for $\Gamma, x{:}\psi \vdash_{\textsc{ml}} E : A$ might exists, the abstraction $\lambda x.E$ need not be typeable.
- Type assignment is decidable. In fact, $\mathcal{W}$ satisfies:
  - *Completeness of* $\mathcal{W}$. If for a term E there are contexts Γ and Γ' and type A, such that Γ' is an instance of Γ and $\Gamma' \vdash_{\textsc{ml}} E : A$, then $\mathcal{W}\,\Gamma\,E = \langle S, B\rangle$, and there is a substitution S' such that $\Gamma' = S'\,(S\,\Gamma)$ and $S'\,(S\,B) \succeq A$.
  - *Soundness of* $\mathcal{W}$. For every term E: if $\mathcal{W}\,\Gamma\,E = \langle S, A\rangle$, then $S\,\Gamma \vdash_{\textsc{ml}} E : A$.

Example 5.10 To express *addition* in ML, we can proceed as follows. We have seen in Definition 1.21 that we can define addition by:

$$Add \;=\; \lambda xy.Cond\ (IsZero\ x)\ y\ (Succ\ (Add\ (Pred\ x)\ y))$$

We have seen in the first section that we can express *Succ*, *Pred*, and *IsZero* in the λ-calculus, and now know that we can express recursive definitions in ML: so we can write

$$Add \;=\; \mathsf{Fix}\ a.\lambda xy.Cond\ (IsZero\ x)\ y\ (Succ\ (a\ (Pred\ x)\ y))$$

Assuming we can type the added constructs as follows:

$$\begin{array}{ll}
Succ & : Num{\to}Num \\
Pred & : Num{\to}Num \\
IsZero & : Num{\to}Bool \\
Cond & : \forall\varphi.Bool{\to}\varphi{\to}\varphi{\to}\varphi
\end{array}$$

we can type the definition of addition as follows (where we write N for *Num*, B for *Bool*, and Γ for $a{:}N{\to}N{\to}N, x{:}N, y{:}N$):

Let

$$\mathcal{D}_1 = \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash Cond : \forall\varphi.B{\to}\varphi{\to}\varphi{\to}\varphi}\,(Ass)}{\Gamma \vdash Cond : B{\to}N{\to}N{\to}N}\,(\forall E) \quad \dfrac{\dfrac{}{\Gamma \vdash IsZero : N{\to}B}\,(Ass) \quad \dfrac{}{\Gamma \vdash x : N}\,(Ax)}{\Gamma \vdash IsZero\ x : B}\,({\to}E)}{\Gamma \vdash Cond\ (IsZero\ x) : N{\to}N{\to}N}\,({\to}E) \quad \dfrac{}{\Gamma \vdash y : N}\,(Ax)}{\Gamma \vdash Cond\ (IsZero\ x)\ y : N{\to}N}\,({\to}E)$$

$$\mathcal{D}_2 = \dfrac{\dfrac{}{\Gamma \vdash_{\textsc{ml}} Succ : N{\to}N}\,(Ass) \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\textsc{ml}} a : N{\to}N{\to}N}\,(Ax) \quad \dfrac{\dfrac{}{\Gamma \vdash_{\textsc{ml}} Pred : N{\to}N}\,(Ass) \quad \dfrac{}{\Gamma \vdash_{\textsc{ml}} x : N}\,(Ax)}{\Gamma \vdash_{\textsc{ml}} Pred\ x : N}\,({\to}E)}{\Gamma \vdash_{\textsc{ml}} a\ (Pred\ x) : N{\to}N}\,({\to}E) \quad \dfrac{}{\Gamma \vdash_{\textsc{ml}} y : N}\,(Ax)}{\Gamma \vdash_{\textsc{ml}} a\ (Pred\ x)\ y : N}\,({\to}E)}{\Gamma \vdash_{\textsc{ml}} Succ\ (a\ (Pred\ x)\ y) : N}\,({\to}E)$$

then we can construct:

$$
\cfrac{\cfrac{\cfrac{\cfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma \vdash Cond\ (IsZero\ x)\ y : N{\to}N\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ \Gamma \vdash Succ\ (a\ (Pred\ x)\ y) : N\end{array}}{\Gamma \vdash Cond\ (IsZero\ x)\ y\ (Succ\ (a\ (Pred\ x)\ y)) : N}\ (\to E)}{a{:}N{\to}N{\to}N, x{:}N \vdash \lambda y.Cond\ (IsZero\ x)\ y\ (Succ\ (a\ (Pred\ x)\ y)) : N{\to}N}\ (\to I)}{a{:}N{\to}N{\to}N \vdash \lambda xy.Cond\ (IsZero\ x)\ y\ (Succ\ (a\ (Pred\ x)\ y)) : N{\to}N{\to}N}\ (\to I)}{\vdash \mathsf{Fix}\ a.\lambda xy.Cond\ (IsZero\ x)\ y\ (Succ\ (a\ (Pred\ x)\ y)) : N{\to}N{\to}N}\ (\mathsf{Fix})
$$

Exercise 5.11 Find an ML*-term for multiplication, and type it; you can abbreviate the derivation above.*

5.2 Polymorphic recursion

In [43] (and, independently, in [34]) A. Mycroft defined a generalisation of Milner's Type Assignment System. This generalisation is made to obtain more permissive types for recursively defined objects.

The example that Mycroft gives to justify his generalisation is the following (using the notation of Λ^{NR}):

$$
\begin{array}{lll}
\mathsf{map} & = \lambda ml.Cond\ (\mathsf{null}\ l)\ \mathsf{nil}\ (\mathsf{cons}\ (m\ (\mathsf{hd}\ l))(\mathsf{map}\ m\ (\mathsf{tl}\ l))) & ; \\
\mathsf{squarelist} & = \lambda l.\mathsf{map}\ (\lambda x.x \times x)\ l & : \\
\multicolumn{3}{l}{\mathsf{squarelist\ (cons\ 2\ nil)}}
\end{array}
$$

where hd, tl, null, nil, and cons are assumed to be familiar list constructors and functions. In the implementation of ML, there is no check if functions are independent or are mutually recursive, so all definitions are dealt with in one step. For this purpose, the language $\mathcal{L}_{\mathrm{ML}}$ is formally extended with a pairing function '$\langle,\rangle$', and the translation of the above expression into $\mathcal{L}_{\mathrm{ML}}$ will be:

$$
\begin{array}{l}
\mathsf{let}\langle \mathsf{map},\mathsf{squarelist}\rangle \ = \ \mathsf{Fix}\,\langle m,s\rangle.\ \langle \lambda gl.Cond\ (\mathsf{null}\ l)\ \mathsf{nil}\ (\mathsf{cons}\ (g\ (\mathsf{hd}\ l))(mg(\mathsf{tl}\ l))), \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \lambda l.(m(\lambda x.x \times x)\ l)\rangle \\
\mathtt{in}\ (\mathsf{squarelist\ (cons\ 2nil\ )})
\end{array}
$$

Within Milner's Type Assignment System these definitions (when defined simultaneously in ML) would get the types:

$$
\begin{array}{ll}
\mathsf{map} & ::\ (num{\to}num) \to [num] \to [num] \\
\mathsf{squarelist} & ::\ [num] \to [num]
\end{array}
$$

while the definition of map alone would yield the type:

$$
\mathsf{map}\ ::\ \forall \varphi_1\varphi_2.(\varphi_1{\to}\varphi_2){\to}[\varphi_1]{\to}[\varphi_2].
$$

Since the definition of map does not depend on the definition of squarelist, one would expect the type inferrer to find the second type for map. That such is not the case is caused by the fact that all occurrences of a recursively defined function on the right hand side within the definition must have the same type as in the left hand side.

There is more than one way to overcome this problem. One is to recognize mutual recursive rules, and treat them as one definition. (Easy to implement, but difficult to formalize, a problem we run into in Section 7). Then, the translation of the above program could be:

$$
\begin{array}{l}
\mathsf{let\ map}\ =\ (\mathsf{Fix}\, mgl.Cond\ (\mathsf{null}\ l)\ \mathsf{nil}\ (\mathsf{cons}\ (g\ (\mathsf{hd}\ l))(mg(\mathsf{tl}\ l)))) \\
\qquad\qquad \mathtt{in}\ (\mathsf{let\ squarelist}\ =\ (\lambda l.(\mathsf{map}\ (\lambda x.x \times x)\ l)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{in}\ (\mathsf{squarelist\ (cons\ 2\ nil)}))
\end{array}
$$

The solution chosen by Mycroft is to allow of a more general rule for recursion than Milner's (Fix)-rule; the set of types used by Mycroft is the same as defined by Milner. In the following definition we show the derivation rules for Mycroft's system.

Definition 5.12 ([43]) *Mycroft type assignment* is defined by replacing rule (Fix) by:

$$(\mathsf{Fix}) : \frac{\Gamma, g{:}\psi \vdash_{\mathrm{MYC}} E : \psi}{\Gamma \vdash_{\mathrm{MYC}} \mathsf{Fix}\; g.E : \psi}$$

We can achieve Mycroft type assignment for Λ^{NR} by changing the rules (*name*) and (*rec name*) into one rule:

$$(\mathit{name}) : \frac{}{\Gamma; \mathcal{E}, \mathit{name}{:}A \vdash \mathit{name} : S\,A}$$

so dropping the distinction between recursive and non-recursive calls.

Thus, the only difference lies in the fact that, in this system, the derivation rule (Fix) allows for type-schemes instead of types, so the various occurrences of x in M can be typed with different Curry-types.

Mycroft's system has the following properties:

- Like in Milner's system, in this system polymorphism can be modelled.
- Type assignment in this system is *undecidable*, as shown by A.J. Kfoury, J. Tiuryn and P. Urzyczyn in [35].

For Λ^{NR}, Mycroft's approach results in the following implementation:

Definition 5.13 (*Mycroft*)

$$\begin{array}{lll}
\mathit{Mycroft}\;\Gamma\;x\;\mathcal{E} & = & \langle \mathit{Id}_S, \Gamma\;x \rangle \\
\mathit{Mycroft}\;\Gamma\;\mathsf{n}\;\mathcal{E} & = & \langle \mathit{Id}_S, \mathit{FreshInstance}\;(\mathcal{E}\;\mathsf{n}) \rangle \\
\mathit{Mycroft}\;\Gamma\;(E_1E_2)\;\mathcal{E} & = & \langle S_1{\circ}S_2{\circ}S_3,\; S_1\,\varphi \rangle \\
 & & \text{where } \begin{array}[t]{lcl} \varphi & = & \mathit{fresh} \\ \langle S_3, C\rangle & = & \mathit{Mycroft}\;\Gamma\;E_1\;\mathcal{E} \\ \langle S_2, A\rangle & = & \mathit{Mycroft}\;(S_3\,\Gamma)\;E_2\;\mathcal{E} \\ S_1 & = & \mathit{unify}\;(S_2\,C)\;(A{\rightarrow}\varphi) \end{array} \\
\mathit{Mycroft}\;\Gamma\;(\lambda x.E)\;\mathcal{E} & = & \langle S_1,\; S_1\,(\varphi{\rightarrow}C) \rangle \\
 & & \text{where } \begin{array}[t]{lcl} \varphi & = & \mathit{fresh} \\ \langle S_1, C\rangle & = & \mathit{Mycroft}\;(\Gamma, x{:}\varphi)\;E\;\mathcal{E} \end{array}
\end{array}$$

$$\begin{array}{llll}
\mathit{CheckType}\;\langle \varepsilon : E\rangle\;\mathcal{E} & = & A, & \\
 & & \text{where } \langle S, A\rangle = \mathit{Mycroft}\;\emptyset\;E\;\mathcal{E} & \\
\mathit{CheckType}\;\langle \mathsf{n} = E_1; \mathit{Defs} : E_2\rangle\;\mathcal{E} & = & \mathit{CheckType}\;\langle \mathit{Defs} : E_2\rangle\;\mathcal{E}, & \text{if } A = B \\
 & & \text{where } \begin{array}[t]{lcl} A & = & \mathcal{E}\;\mathsf{n} \\ \langle S, B\rangle & = & \mathit{Mycroft}\;\emptyset\;E_1\;\mathcal{E} \end{array} &
\end{array}$$

Notice that, in this approach, the environment never gets updated, so has to be provided (by the user) before it starts running.

5.3 The difference between Milner's and Mycroft's system

Since Mycroft's system is a true extension of Milner's, there are terms typeable in Mycroft's system that are not typeable in Milner's. For example,

$$\textsf{Fix}\ g.((\lambda ab.a)\,(g\lambda c.c)\,(g\lambda de.d)) : \forall\varphi_1\varphi_2.(\varphi_1\to\varphi_2).$$

is a derivable statement in Mycroft's system (where $\Gamma = \{g{:}\forall\varphi_1\varphi_2.(\varphi_1\to\varphi_2)\}$:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\dfrac{\dfrac{\dfrac{\Gamma,a{:}\varphi_3\to\varphi_4,b{:}B\vdash a:\varphi_3\to\varphi_4}{\Gamma,b{:}B\vdash\lambda b.a:(\varphi_3\to\varphi_4)\to B\to\varphi_3\to\varphi_4}}{\Gamma\vdash\lambda ab.a:(\varphi_3\to\varphi_4)\to B\to\varphi_3\to\varphi_4}\quad\dfrac{\dfrac{}{\Gamma\vdash g:(C\to C)\to\varphi_3\to\varphi_4}\quad\dfrac{\dfrac{}{\Gamma,c{:}C\vdash c:C}}{\Gamma\vdash\lambda c.c:C\to C}}{\Gamma\vdash g\lambda c.c:\varphi_3\to\varphi_4}}{\Gamma\vdash(\lambda ab.a)\,(g\lambda c.c):B\to\varphi_3\to\varphi_4}\\ \vdots\end{array}\quad\dfrac{\dfrac{}{\Gamma\vdash g:(D\to E\to D)\to B}\quad\dfrac{\dfrac{}{\Gamma,d{:}D,e{:}E\vdash d:D}}{\Gamma\vdash\lambda de.d:D\to E\to D}}{\Gamma\vdash g\lambda de.d:B}}{\Gamma\vdash((\lambda ab.a)\,(g\lambda c.c)\,(g\lambda de.d)):\varphi_3\to\varphi_4}}{\Gamma\vdash((\lambda ab.a)\,(g\lambda c.c)\,(g\lambda de.d)):\forall\varphi_1\varphi_2.(\varphi_1\to\varphi_2)}}{\emptyset\vdash\textsf{Fix}\ g.((\lambda ab.a)\,(g\lambda c.c)\,(g\lambda de.d)):\forall\varphi_1\varphi_2.(\varphi_1\to\varphi_2)}$$

It is easy to see that this term is not typeable using Milner's system, because the types needed for g in the body of the term cannot be unified.

But, the generalisation allows for more than was aimed at by Mycroft: in contrast to what Mycroft suggests, type assignment in this system is undecidable. And not only is the set of terms that can be typed in Mycroft's system larger than in Milner's, it is also possible to assign more general types to terms that are typeable in Milner's system. Take, for example,

$$R = \textsf{Fix}\ r.(\lambda xy.(r\,(ry\,(\lambda ab.a))\,x)),$$

then the statement

$$R : \forall\varphi_1\varphi_2\varphi_3.(\varphi_1\to\varphi_2\to\varphi_3)$$

is derivable in Mycroft's system (where $\Gamma = \{r{:}\forall\varphi_1\varphi_2\varphi_3.(\varphi_1\to\varphi_2\to\varphi_3), x{:}\varphi_4, y{:}\varphi_5\}$).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\vdash r:\varphi_7\to\varphi_4\to\varphi_6}\quad\dfrac{\dfrac{\dfrac{}{\Gamma\vdash r:\varphi_5\to(A\to B\to A)\to\varphi_7}\quad\dfrac{}{\Gamma\vdash y:\varphi_5}}{\Gamma\vdash ry:(A\to B\to A)\to\varphi_7}\quad\dfrac{\dfrac{}{\Gamma,a{:}A,b{:}B\vdash a:A}}{\Gamma\vdash\lambda ab.a:A\to B\to A}}{\Gamma\vdash ry\,(\lambda ab.a):\varphi_7}}{\Gamma\vdash r\,(ry\,(\lambda ab.a)):\varphi_4\to\varphi_6}\quad\dfrac{}{\Gamma\vdash x:\varphi_4}}{\Gamma\vdash r\,(ry\,(\lambda ab.a))\,x:\varphi_6}}{r{:}\forall\varphi_1\varphi_2\varphi_3.(\varphi_1\to\varphi_2\to\varphi_3)\vdash\lambda xy.(r\,(ry\,(\lambda ab.a))\,x):\varphi_4\to\varphi_5\to\varphi_6}}{r{:}\forall\varphi_1\varphi_2\varphi_3.(\varphi_1\to\varphi_2\to\varphi_3)\vdash\lambda xy.(r\,(ry\,(\lambda ab.a))\,x):\forall\varphi_1\varphi_2\varphi_3.(\varphi_1\to\varphi_2\to\varphi_3)}}{\emptyset\vdash\textsf{Fix}\ r.(\lambda xy.(r\,(ry\,(\lambda ab.a))\,x)):\forall\varphi_1\varphi_2\varphi_3.(\varphi_1\to\varphi_2\to\varphi_3)}$$

R is also typeable in Milner's system:

Exercise 5.14 Check the following result:

$$\vdash_{\text{ML}} R : \forall\varphi_4\varphi_5.((\varphi_4\to\varphi_5\to\varphi_4)\to(\varphi_4\to\varphi_5\to\varphi_4)\to\varphi_4\to\varphi_5\to\varphi_4)$$

6 Combinatory Logic

In this section, we will focus on Curry's Combinatory Logic [20], an alternative approach to express computability, developed at about the same time as Church's λ-calculus. It will be defined as a special kind of applicative TRS [37] that we will study in the next section, with the restriction that formal parameters of function symbols are not allowed to have structure, and right-hand sides of term rewriting rules are constructed of term-variables only.

Definition 6.1 (COMBINATORY LOGIC [20]) The original definition of Combinatory Logic defines two rules:

$$\begin{array}{lcl} \mathsf{K}\ x\ y & \to & x \\ \mathsf{S}\ x\ y\ z & \to & xz(yz) \end{array}$$

and defines terms as

$$t ::= \mathsf{K} \mid \mathsf{S} \mid t_1 t_2$$

The first rule expresses *removal* of information, whereas the second expresses *distribution*: notice that its third parameter gets distributed over the first and second.

Notice that we can define I as SKK, since $\mathsf{SKK}x \to \mathsf{K}x(\mathsf{K}x) \to x$. We therefore will consider the following rule to be present as well.

$$\mathsf{I}\ x \quad \to\ x$$

Notice that this defines a *higher order language* with a *first order* reduction system: the combinators K, S, and I offically do not have a *fixed arity* and can be applied to any number of terms, though they need a specific number present for their rules to become applicable.

6.1 The relation between CL and LC

To emphasise the power of a system as simple as CL, we now focus on the relation between CL and in LC, and show that every λ-term can be translated to a CL program. This shows of course that Combinatory Logic is Turing Complete: all computable functions can be expressed in terms of S, K, and I.

Example 6.2 For CL, the interpretation of terms in Λ is given by:

$$\begin{array}{lcll} \langle x\rangle_\lambda & = & x, & \text{for all } x \in \mathcal{V}, \\ \langle \mathsf{S}\rangle_\lambda & = & (\lambda xyz.xz(yz)), & \\ \langle \mathsf{K}\rangle_\lambda & = & (\lambda xy.x), & \\ \langle \mathsf{I}\rangle_\lambda & = & (\lambda x.x), & \\ \langle t_1t_2\rangle_\lambda & = & \langle t_1\rangle_\lambda\ \langle t_2\rangle_\lambda & \end{array}$$

We will now define a mapping from λ-terms to expressions in CL.

Definition 6.3 The mapping $[\![\ ]\!]_{\mathrm{CL}} : \Lambda \to \mathcal{T}_{\mathrm{CL}}$ is defined by:

$$\begin{array}{lcl} [\![x]\!]_{\mathrm{CL}} & = & x, \\ [\![\lambda x.M]\!]_{\mathrm{CL}} & = & \mathit{Fun}\ x\ [\![M]\!]_{\mathrm{CL}}, \\ [\![MN]\!]_{\mathrm{CL}} & = & [\![M]\!]_{\mathrm{CL}}\ [\![N]\!]_{\mathrm{CL}}. \end{array}$$

where *Fun x t*, with $t \in \mathcal{T}_{\mathrm{CL}}$, is defined by induction on the structure of t:

$$\begin{array}{lll} \mathit{Fun}\ x\ x & = \mathsf{I}, & \\ \mathit{Fun}\ x\ t & = \mathsf{K}\,t, & \text{if } x \text{ not in } t, \\ \mathit{Fun}\ x\ (t_1t_2) & = \mathsf{S}\,(\mathit{Fun}\ x\ t_1)\,(\mathit{Fun}\ x\ t_2). & \end{array}$$

Notice that the auxiliary function *Fun*, that takes a variable and a term in $\mathcal{T}_{\rm CL}$ and returns a term in $\mathcal{T}_{\rm CL}$, is only evaluated in the definition of $[\![\cdot]\!]_{\rm CL}$ with a variable or an application as second argument.

As for the accuracy of the above definitions, we show first that *Fun* acts as abstraction:

Lemma 6.4 $(\mathit{Fun}\ x\ t)\ v \to t[v/x]$.

Proof:
$$\begin{array}{llll} (\mathit{Fun}\ x\ x)\ t & = \mathsf{I}\,t & \to\ t & \\ (\mathit{Fun}\ x\ t_1)\ t_2 & = \mathsf{K}\,t_1\,t_2 & \to\ t_1, & \text{if } x \text{ not in } t \\ (\mathit{Fun}\ x\ (t_1t_2))\ t_3 & = \mathsf{S}\,(\mathit{Fun}\ x\ t_1)\,(\mathit{Fun}\ x\ t_2)\ t_3 & \to & \\ & \quad ((\mathit{Fun}\ x\ t_1)\ t_3)\,((\mathit{Fun}\ x\ t_2)\ t_3) & \to\ (\mathit{IH}) & \\ & \quad t_1[t_3/x]\ t_2[t_3/x] & =\ (t_1\,t_2)[t_3/x]. & \blacksquare \end{array}$$

For the interpretations defined above the following property holds:

Lemma 6.5 ([10]) *i)* $\langle \mathit{Fun}\ x\ t\rangle_\lambda \twoheadrightarrow_\beta \lambda x.\langle t\rangle_\lambda$
ii) $\langle [\![M]\!]_{\rm CL}\rangle_\lambda \twoheadrightarrow_\beta M$.
iii) If $t \to u$ *in* CL, *then* $\langle t\rangle_\lambda \twoheadrightarrow_\beta \langle u\rangle_\lambda$.

Proof: i) By induction on the definition of the function *Fun*.

a) $\mathit{Fun}\ x\ x = \mathsf{I}$, and $\langle \mathsf{I}\rangle_\lambda = \lambda x.x$.

b) $\mathit{Fun}\ x\ t = \mathsf{K}\,t$, and $\langle \mathsf{K}\,t\rangle_\lambda = \langle \mathsf{K}\rangle_\lambda\ \langle t\rangle_\lambda = (\lambda ab.a)\ \langle t\rangle_\lambda \to_\beta \lambda b.\langle t\rangle_\lambda$, if x not in t.

c) $\mathit{Fun}\ x\ (t_1t_2) = \mathsf{S}\,(\mathit{Fun}\ x\ t_1)\,(\mathit{Fun}\ x\ t_2)$, and

$$\begin{array}{ll} \langle \mathsf{S}\,(\mathit{Fun}\ x\ t_1)\,(\mathit{Fun}\ x\ t_2)\rangle_\lambda & \triangleq \\ \langle \mathsf{S}\rangle_\lambda\ \langle \mathit{Fun}\ x\ t_1\rangle_\lambda\ \langle \mathit{Fun}\ x\ t_2\rangle_\lambda & \to_\beta\ (\mathit{IH}) \\ \langle \mathsf{S}\rangle_\lambda\ (\lambda x.\langle t_1\rangle_\lambda)\ (\lambda x.\langle t_2\rangle_\lambda) & \triangleq \\ (\lambda abc.ac(bc))\ (\lambda x.\langle t_1\rangle_\lambda)\ (\lambda x.\langle t_2\rangle_\lambda) & \twoheadrightarrow_\beta \\ \lambda c.(\lambda x.\langle t_1\rangle_\lambda)\,c\,((\lambda x.\langle t_2\rangle_\lambda)\,c) & \to_\beta \\ \lambda c.(\langle t_1\rangle_\lambda[c/x])\ (\langle t_2\rangle_\lambda[c/x]) & =_\alpha\ \lambda x.(\langle t_1\rangle_\lambda \langle t_2\rangle_\lambda) \triangleq \lambda x.\langle t_1t_2\rangle_\lambda \end{array}$$

ii) By induction on the structure of (λ-)terms.

a) $M = x$. Since $\langle [\![x]\!]_{\rm CL}\rangle_\lambda = \langle x\rangle_\lambda = x$, this is immediate.

b) $M = \lambda x.N$. Since $\langle [\![\lambda x.N]\!]_{\rm CL}\rangle_\lambda = \langle \mathit{Fun}\ x\ [\![N]\!]_{\rm CL}\rangle_\lambda \twoheadrightarrow_\beta \lambda x.\langle [\![N]\!]_{\rm CL}\rangle_\lambda$ by the previous part, and $\lambda x.\langle [\![N]\!]_{\rm CL}\rangle_\lambda \twoheadrightarrow_\beta \lambda x.N$ by induction.

c) $M = PQ$. Since $\langle [\![PQ]\!]_{\rm CL}\rangle_\lambda = \langle [\![P]\!]_{\rm CL}\rangle_\lambda\ \langle [\![Q]\!]_{\rm CL}\rangle_\lambda \twoheadrightarrow_\beta PQ$ by induction.

iii) We focus on the case that $t = Ct_1 \cdots t_n$ for some name C with arity n. Let $Cx_1 \cdots x_n \to t'$ be the definition for C, then $u = t'[\overrightarrow{t_i/x_i}]$. Notice that $\langle t\rangle_\lambda = (\lambda x_1 \cdots x_n.\langle t'\rangle_\lambda)\langle t_1\rangle_\lambda \cdots \langle t_n\rangle_\lambda \twoheadrightarrow_\beta \langle t'\rangle_\lambda[\overrightarrow{\langle t_i/x_i\rangle_\lambda}] = \langle t'[\overrightarrow{t_i/x_i}]\rangle_\lambda = \langle u\rangle_\lambda$. $\blacksquare$

For example,
$$\begin{array}{ll} [\![\lambda xy.x]\!]_{\rm CL} & = \mathit{Fun}\ x\ [\![\lambda y.x]\!]_{\rm CL} \\ & = \mathit{Fun}\ x\ (\mathit{Fun}\ y\ x) \\ & = \mathit{Fun}\ x\ (\mathsf{K}\,x) \\ & = \mathsf{S}\,(\mathit{Fun}\ x\ \mathsf{K})\ (\mathit{Fun}\ x\ x) \\ & = \mathsf{S}\,(\mathsf{K}\,\mathsf{K})\,\mathsf{I} \end{array}$$

$$\begin{array}{lll} \text{and } \langle [\![\lambda xy.x]\!]_{\text{CL}} \rangle_\lambda & = & \langle \mathsf{S}\,(\mathsf{K}\,\mathsf{K}))\,\mathsf{I} \rangle_\lambda \\ & = & (\lambda xyz.xz(yz))((\lambda xy.x)\lambda xy.x)\lambda x.x \\ & \twoheadrightarrow_\beta & \lambda xy.x. \end{array}$$

There exists no converse of the second property: notice that $[\![\langle \mathsf{K} \rangle_\lambda]\!]_{\text{CL}} = \mathsf{S}\,(\mathsf{K}\,\mathsf{K})\,\mathsf{I}$ which are both in normal form, and not the same; moreover, the mapping $\langle\ \rangle_\lambda$ does not preserve normal forms or reductions:

Example 6.6 ([10]) *i*) $\mathsf{S}\,\mathsf{K}$ is a normal form, but $\langle \mathsf{S}\,\mathsf{K} \rangle_\lambda \twoheadrightarrow_\beta \lambda xy.y$,

ii) $t = \mathsf{S}\,(\mathsf{K}\,(\mathsf{S}\,\mathsf{I}\,\mathsf{I}))\,(\mathsf{K}\,(\mathsf{S}\,\mathsf{I}\,\mathsf{I}))$ is a normal form, but $\langle t \rangle_\lambda \twoheadrightarrow_\beta \lambda c.(\lambda x.xx)(\lambda x.xx)$, which does not have a β-normal form,

iii) $t = \mathsf{S}\,\mathsf{K}\,(\mathsf{S}\,\mathsf{I}\,\mathsf{I}\,(\mathsf{S}\,\mathsf{I}\,\mathsf{I}))$ has no normal form, while $\langle t \rangle_\lambda \twoheadrightarrow_\beta \lambda x.x$.

6.2 Extending CL

Bracket abstraction algorithms like $\langle\ \rangle_\lambda$ are used to translate λ-terms to combinator systems, and form, together with the technique of *lambda lifting* the basis of the Miranda [51] compiler. It is possible to define such a translation also for combinator systems that contain other combinators. With some accompanying optimization rules they provide an interesting example. If in the bracket abstraction we would use the following combinator set:

$$\begin{array}{llll} \mathsf{I} & x & \to & x \\ \mathsf{K} & x\ y & \to & x \\ \mathsf{B} & x\ y\ z & \to & x(yz) \\ \mathsf{C} & x\ y\ z & \to & xzy \\ \mathsf{S} & x\ y\ z & \to & xz(yz) \end{array}$$

then we could, as in [22], extend the notion of reduction by defining the following optimizations:

$$\begin{array}{llllll} \mathsf{S} & (\mathsf{K}\,x) & (\mathsf{K}\,y) & \to & \mathsf{K}\,(x\,y) \\ \mathsf{S} & (\mathsf{K}\,x) & \mathsf{I} & \to & x \\ \mathsf{S} & (\mathsf{K}\,x) & y & \to & \mathsf{B}\,x\,y \\ \mathsf{S} & x & (\mathsf{K}\,y) & \to & \mathsf{C}\,x\,y \end{array}$$

The correctness of these rules is easy to check.

Exercise 6.7 Check that the above rules are admissible, and check if they introduce any conflict with respect to types.

Notice that, by adding this optimisation, we are stepping outside the realm of λ-calculus: we cannot express (arbitrary) pattern matching in the λ-calculus. Also, we now allow reduction of terms starting with S that have *only two* arguments present.

Moreover, the rule $\mathsf{S}\,(\mathsf{K}\,x)\,(\mathsf{K}\,y) \to \mathsf{K}\,(x\,y)$ expresses that this reduction can only take place when the first and second argument of S are of the shape $\mathsf{K}\,t$. So, in particular, these arguments can not be any term as is the case with normal combinator rules, but *must* have a precise structure. In fact, adding these rules introduces *pattern matching* and would give a notion of rewriting that is known as *term rewriting*, which we will look at in Section 7.

7 Pattern matching: term rewriting

The notion of reduction (rewriting) we will study in this section is that of *term rewriting* [37, 38], a notion of computation which main feature is that of *pattern matching*, making it syntactically closer to most functional programming languages that pure LC.

7.1 Term Rewriting Systems

Term rewriting systems are an extension of LC by allowing the formal parameters to have structure. Terms are built out of variables, *function symbols* and application; there is no abstraction, functions are modelled via *rewrite rules* that describe how terms can be modified.

Definition 7.1 (SYNTAX) *i*) An *alphabet* or *signature* Σ consists of a countable, infinite set $\mathcal{X}$ of variables $x_1,x_2,x_3,\ldots$ (or $x,y,z,x',y',\ldots$), a non-empty set $\mathcal{F}$ of *function symbols* $F,G,\ldots$, each with a fixed arity.

ii) The set $T(\mathcal{F},\mathcal{X})$ of *terms*, ranged over by t, is defined by:

$$t ::= x \mid F \mid (t_1 \cdot t_2)$$

As before, we will omit '$\cdot$' and obsolete brackets.

A term substitution, written $\{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\}$ or by a capital character like 'R' when we need not be specific, is an operation on terms where term variables are replaced by terms, and corresponds to the implicit substitution of the λ-calculus. We write t^{R} for the result of applying the term substitution R to t.

Reduction on $T(\mathcal{F},\mathcal{X})$ is defined through rewrite rules. They are intended to show how a term can be modified, by stating how a (sub)term that matches a certain structure will be replaced by another, that might be constructed using parts of the original term.

Definition 7.2 (REDUCTION) *i*) A *rewrite rule* is a pair (l,r) of terms. Often, a rewrite rule will get a name, e.g. $\mathbf{r}$, and we write

$$\mathbf{r} : l \to r$$

Two conditions are imposed:

a) $l = F\,t_1 \cdots t_n$, for some $F \in \mathcal{F}$ and $t_1,\ldots,t_n \in T(\mathcal{F},\mathcal{X})$, and

b) $fv(r) \subseteq fv(l)$.

ii) The *patterns* of this rule are the terms t_i, $1 \leq i \leq n$, such that either t_i is not a variable, or t_i is variable and there is a t_j $(1 \leq i \neq j \leq n)$ such that $t_i \in fv(t_j)$.

iii) A rewrite rule $l \to r$ determines a set of *rewrites* $l^{\mathrm{R}} \to r^{\mathrm{R}}$ for all term substitutions R. The left-hand side l^{R} is called a *redex*, the right-hand side r^{R} its *contractum*.

iv) A redex t may be substituted by its contractum t' inside a context C[ ]; this gives rise to *rewrite steps* $\mathrm{C}[t] \to \mathrm{C}[t']$. Concatenating rewrite steps we have *rewrite sequences* $t_0 \to t_1 \to t_2 \to \cdots$. If $t_0 \to \cdots \to t_n$ $(n \geq 0)$ we also write $t_0 \twoheadrightarrow t_n$.

Notice that, if $l \to r$ is a rule, then l is not a variable, nor an application that 'starts with' a variable. Also, r does not introduce new variables: this is because, during rewriting, the variables in l are not part of the term information, but are there only to be 'filled' with subterms, that are then used when building the replacement term; a new variable in r would have no term to be replaced with. In fact, we could define term rewriting correctly by not allowing any variables at all outside rewrite rules; remember CL, that we can now write as a set of term rewrite rules:

Definition 7.3 (COMBINATORY LOGIC (CL))

$$\begin{array}{lll}\mathsf{S}\,x\,y\,z & \rightarrow & x\,z\,(y\,z),\\ \mathsf{K}\,x\,y & \rightarrow & x,\\ \mathsf{I}\,x & \rightarrow & x.\end{array}$$

This is in fact the standard definition of Combinatory Logic (CL); the last rule was not part of the original definition, but is nowadays normally added. A difference between the two notions of CL is that in the latter the reduction is *weak*: we can reduce a term that contains S using the first rule *only* if three parameters are provided; in the former, a redex is present even if S is given only *one* argument.

In Section 6.1, we have seen that we can express every λ-term in CL, so to model all computable functions, we only need variables in rules.

Definition 7.4 A *Term Rewriting System* (TRS) is defined by a pair $(\Sigma, \mathbf{R})$ of an alphabet Σ and a set $\mathbf{R}$ of rewrite rules.

We take the view that in a rewrite rule a certain symbol is defined.

Definition 7.5 In a rewrite rule $F\,t_1\,\cdots\,t_n \rightarrow r$, F is called *the defined symbol* of $\mathbf{r}$, and $\mathbf{r}$ is said to *define* F. F is *a defined symbol* if there is a rewrite rule that defines F, and $Q \in \mathcal{F}$ is called a *constructor* if Q is not a defined symbol.

Notice that the defined symbol of a rule is allowed to appear more than once in a rule; in particular, it is allowed to appear on the right-hand side, thereby modelling recursion.

Example 7.6 The following is a set of rewrite rules that defines the functions append and map on lists and establishes the associativity of append. The function symbols nil and cons are constructors.

$$\begin{array}{lll}\mathsf{append}\;\mathsf{nil}\;l & \rightarrow & l\\ \mathsf{append}\;(\mathsf{cons}\;x\;l)\;l' & \rightarrow & \mathsf{cons}\;x\;(\mathsf{append}\;l\;l')\\ \mathsf{append}\;(\mathsf{append}\;l\;l')\;l'' & \rightarrow & \mathsf{append}\;l\;(\mathsf{append}\;l'\;l'')\\ \mathsf{map}\;f\;\mathsf{nil} & \rightarrow & \mathsf{nil}\\ \mathsf{map}\;f\;(\mathsf{cons}\;y\;l) & \rightarrow & \mathsf{cons}\;(f\;y)\;(\mathsf{map}\;f\;l)\end{array}$$

With this notion of rewriting, we obtain more than just the normal functional paradigm: there the set $\mathcal{F}$ is divided into *function symbols* and *constructors*, and, in a rewrite rule, function symbols are not allowed to appear in 'constructor position' and vice-versa. For example, in the rule $F\,t_1\,\cdots\,t_n \rightarrow r$, the symbol F appears in function position and is thereby a function symbol (we call those *defined symbols*); the terms t_i can contain symbols from $\mathcal{F}$, as long as those are not function symbols, i.e. are constructors. This division is not used in TRS: the symbol 'append' appears in the third rule in both function and constructor position, so, in TRS, the distinction between the notions of function symbol and constructor is lost.

So, in particular, the following is a correct TRS.

$$\begin{array}{lll}\mathsf{In\text{-}left}\;(\mathsf{Pair}\;x\;y) & \rightarrow & x\\ \mathsf{In\text{-}right}\;(\mathsf{Pair}\;x\;y) & \rightarrow & y\\ \mathsf{Pair}\;(\mathsf{In\text{-}left}\;x)\;(\mathsf{In\text{-}right}\;x) & \rightarrow & x\end{array}$$

A difficulty with this TRS is that it forms Klop's famous 'Surjective Pairing' example [36]; this function cannot be expressed in LC because when added to LC, the Church-Rosser property no longer holds. This implies that, although both LC and TRS are Turing-machine complete, so are expressive enough to encode all computable functions (algorithms), there is no general syntactic solution for patterns in LC, so a full-purpose translation (interpretation) of TRS in LC is not feasible.

7.2 Type assignment for TRS

We will now set up a notion of type assignment for TRS, as defined and studied in [9, 6, 7]. For the same reasons as before we use an *environment* providing a type for each function symbol. From it we can derive many types to be used for different occurrences of the symbol in a term, all of them 'consistent' with the type provided by the environment; an environment functions as in the *Milner* and *Mycroft* algorithms.

Definition 7.7 (ENVIRONMENT) An *environment* is a mapping $\mathcal{E} : \mathcal{F} \rightarrow \mathcal{T}_C$.

We define type assignment much as before, but with a small difference. Since there is no notion of abstraction in TRS, we have no longer the need to require that contexts are *mappings* from variables to types; instead, here we will use the following definition.

Definition 7.8 A TRS-*context* is a set of statements with variables (not necessarily distinct) as subjects.

Notice that this generalisation would allow for xx to be typeable.

Definition 7.9 (TYPE ASSIGNMENT ON TERMS) *Type assignment* (with respect to an environment $\mathcal{E}$) is defined by the following natural deduction system. Note the use of a substitution in rule $(\mathcal{F})$.

$$(Ax) : \frac{}{\Gamma \vdash x : A} \; (x{:}A \in \Gamma)$$

$$(\mathcal{F}) : \frac{}{\Gamma \vdash F : A} \; (\exists S \; [S(\mathcal{E}\,F) = A])$$

$$(\rightarrow E) : \frac{\Gamma \vdash t_1 : A \rightarrow B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B}$$

As before, the use of an environment in rule $(\mathcal{F})$ introduces a notion of polymorphism for our function symbols, which is an extension like the ML-style of polymorphism. The environment returns the 'principal type' for a function symbol; this symbol can be used with types that are 'instances' of its principal type, obtained by applying substitutions.

The main properties of this system are:

- Principal types. We will focus on this in Section 7.3.
- Subject reduction. This will be proven in Section 7.4.
- It is, in general, not possible to prove a strong normalisation result: take t that is typeable, and the rule $t \rightarrow t$, then clearly t is not normalisable. However, it is possible to prove a strong normalisation result for systems where recursive rules are restricted to be of the shape

  $$F\overrightarrow{C[\vec{x}]} \;\rightarrow\; C'(F\overrightarrow{C_1[\vec{x}]}) \;\ldots\; (F\overrightarrow{C_m[\vec{x}]}),$$

  where, for every $j \in \underline{m}$, $\overrightarrow{C_j[\vec{x}]}$ is a subterm of $\overrightarrow{C[\vec{x}]}$, so F is only called recursively on terms that are substructures of its initial arguments (see [7] for details); this scheme generalizes primitive recursion.

Notice that, for example, the rules of a combinator system like CL are not recursive, so this result gives us immediately a strong normalisation result for combinator systems.

The relation between type assignment in LC and that in CS (restricted to CL with the *environment* $\mathcal{E}$ of Example 3.6) is very strong, as the following theorem shows.

Exercise 7.10 Show, by induction on the definition of Fun, $[\![\ ]\!]_{\text{CL}}$ *and* $\langle\ \rangle_\lambda$, *that*

i) $\Gamma, y{:}A \vdash_{\mathcal{E}_{\text{CL}}} t{:}B$ *implies* $\Gamma, y{:}A \vdash_{\mathcal{E}_{\text{CL}}} \mathit{Fun}\ y\ t{:}A{\to}B$.

ii) $\Gamma \vdash_{\text{c}} M{:}A$ *implies* $\Gamma \vdash_{\mathcal{E}_{\text{CL}}} [\![M]\!]_{\text{CL}}{:}A$.

iii) $\Gamma \vdash_{\mathcal{E}_{\text{CL}}} t{:}A$ *implies* $\Gamma \vdash_{\text{c}} \langle t\rangle_\lambda{:}A$.

As a corollary of this exercise, we obtain the decidability of type assignment in our system (or, more precisely, the decidability of type assignment in CL with respect to $\mathcal{E}$). As a matter of fact, decidability of type assignment was first proven by J.R. Hindley for CL [30].

7.3 The principal pair for a term

In this subsection, the principal pair for a term t with respect to the environment $\mathcal{E}$ – $pp_{Env}\,t$ – is defined, consisting of context Π and type P, using Robinsons unification algorithm *unify*. In the following, we will show that for every typeable term, this is a legal pair and is indeed the most general one.

Definition 7.11 We define, for every term t the notion $pp_{Env}\,t = \langle \Pi, P\rangle$ inductively by:

$$\begin{array}{lcl}
pp_{Env}\,x & = & \langle x{:}\varphi, \varphi\rangle \\
 & & \text{where } \varphi = \mathit{fresh} \\
pp_{Env}\,F & = & \langle \emptyset, \mathcal{E}\ F\rangle \\
pp_{Env}\,t_1t_2 & = & S\,\langle \Pi_1 \cup \Pi_2, \varphi\rangle \\
 & & \text{where } \begin{array}{lcl} \varphi & = & \mathit{fresh} \\ \langle \Pi_1, P_1\rangle & = & pp_{Env}\,t_1 \\ \langle \Pi_2, P_2\rangle & = & pp_{Env}\,t_2 \\ S & = & \mathit{unify}\ P_1\ (P_2{\to}\varphi) \end{array}
\end{array}$$

Notice that, since we allow a context to contain more than one statement for each variable, we do not require Π_1 and Π_2 to agree via the unification of contexts.

The following shows that substitution is a sound operation on derivations.

Exercise 7.12 (SOUNDNESS OF SUBSTITUTION) *If* $\Gamma \vdash t{:}A$, *then* $S\,\Gamma \vdash t{:}S\,A$, *for every substitution* S.

Exercise 7.13 (SOUNDNESS OF pp_{Env}) *Verify that* $pp_{Env}\,t = \langle \Pi, P\rangle$ *implies* $\Pi \vdash t{:}P$.

In the following theorem we show that the operation of substitution is complete.

Theorem 7.14 (COMPLETENESS OF SUBSTITUTION) *If* $\Gamma \vdash t{:}A$, *then there are* Π, P, *and a substitution* S *such that:* $pp_{Env}\,t = \langle \Pi, P\rangle$, *and* $S\,\Pi \subseteq \Gamma$, $S\,P = A$.

Proof: By induction on the structure of t.

i) $t \equiv x$. Then $\{x{:}A\} \subseteq \Gamma$. Then there is a φ such that $pp_{Env}\,x = \langle \{x{:}\varphi\}, \varphi\rangle$. Take $S = (\varphi \mapsto A)$.

ii) $t \equiv t_1\,t_2$, and there is a $B \in \mathcal{T}_{\text{c}}$ such that $\Gamma \vdash t_1{:}B{\to}A$, and $\Gamma \vdash t_2{:}B$. By induction, for $i = 1,2$, there are Π_i, P_i, and a substitution S_i such that

$pp_{Env}\,t_i = \langle \Pi_i, P_i \rangle$, $S_1\,\Pi_1 \subseteq \Gamma$, $S_2\,\Pi_2 \subseteq \Gamma$, $S_1\,P_1 = B{\rightarrow}A$, and $S_2\,P_2 = B$,

Let φ be a fresh type-variable, and assume, without loss of generality, that $S_2\,\varphi = A$. Since now $S_1\,P_1 = B{\rightarrow}A = S_2\,(P_2{\rightarrow}\varphi)$, by Property 2.14, there exists substitutions S_u, S' such that $S_u = \mathit{unify}\ P_1\ P_2{\rightarrow}\varphi$, and $S_1{\circ}S_2 = S'{\circ}S_u = S_2{\circ}S_1$. Take $S = S_1{\circ}S_2$, and $C = \varphi$.

7.4 Subject reduction

By Definition 7.2, if a term t is rewritten to the term t' using the rewrite rule $l \rightarrow r$, there is a subterm t_0 of t, and a replacement R, such that $l^{\mathrm{R}} = t_0$, and t' is obtained by replacing t_0 by r^{R}. To guarantee the subject reduction property, we should accept only those rewrite rules $l \rightarrow r$, that satisfy:

For all replacements R, contexts Γ and types A: if $\Gamma \vdash l^{\mathrm{R}}{:}\,A$, then $\Gamma \vdash r^{\mathrm{R}}{:}\,A$.

because then we are sure that all possible rewrites are safe. It might seem straightforward to show this property, and indeed, in many papers that consider a language with pattern matching, the property is just claimed and no proof is given. But, as we will see in this section, it does not automatically hold. However, in the notion of type assignment as presented in this paper, it is easy to formulate a condition that rewrite rules should satisfy in order to be acceptable.

Definition 7.15 *i*) We say that $l \rightarrow r \in \mathbf{R}$ with defined symbol F *is typeable with respect to $\mathcal{E}$*, if there are Π, P such that $\langle \Pi, P \rangle$ is the principal pair for l, $\Pi \vdash r{:}\,P$, and such that the left-most occurrence of F in $\Pi \vdash l{:}\,P$ is typed with $\mathcal{E}(F)$.

ii) We say that a TRS *is typeable with respect to $\mathcal{E}$*, if all $\mathbf{r} \in \mathbf{R}$ are.

Notice that the notion $pp_{Env}\,t$ is defined independently from the definition of typeable rewrite rules. Also, since only '*the left-most occurrence of F in $\Pi \vdash l{:}\,P$ is typed with $\mathcal{E}(F)$*', this notion of type assignment uses Mycroft's solution for recursion; using Milner's, the definition would have read '*all occurrences of F in $\Pi \vdash l{:}\,P$ and $\Pi \vdash r{:}\,P$ are typed with $\mathcal{E}(F)$*'.

In the following lemma we show that if F is the defined symbol of a rewrite rule, then the type $\mathcal{E}\ F$ dictates not only the type for the left and right-hand side of that rule, but also the principal type for the left-hand side.

Lemma 7.16 If F is the defined symbol of the typeable rewrite rule $l \rightarrow r$, then there are contexts Π, Γ, amd types A_i ($i \in \underline{n}$) and A such that

$$\begin{array}{lcl} \mathcal{E}\ F & = & A_1{\rightarrow}\cdots{\rightarrow}A_n{\rightarrow}A, \\ pp_{Env}\,l & = & \langle \Pi, A \rangle, \\ \Gamma \vdash l{:}\,A, \text{ and} && \\ \Gamma \vdash r{:}\,A. && \end{array}$$

Proof: Easy, using 7.14 and the fact that if B is a substitution instance of A, and A a substitution instance of B, then $A = B$. ■

As an example of a rule that is not safe, take the typed rewrite rule in the next example: the types assigned to the nodes containing x and y are not the most general ones needed to find the type for the left-hand side of the rewrite rule.

Example 7.17 As an example of a rewrite rule that does not satisfy the above restriction, so will *not* be considered to be typeable, take

$$
\begin{array}{lcl}
S\,x\,y\,z & \to & x\,z\,(y\,z) \\
K\,x\,y & \to & x \\
I\,x & \to & x \\
M\,(S\,x\,y) & \to & S\,I\,y.
\end{array}
$$

Take the environment

$$
\begin{array}{lcl}
\mathcal{E}\;S & = & (\varphi_1\to\varphi_2\to\varphi_3)\to(\varphi_1\to\varphi_2)\to\varphi_1\to\varphi_3 \\
\mathcal{E}\;K & = & \varphi_4\to\varphi_5\to\varphi_4 \\
\mathcal{E}\;I & = & \varphi_6\to\varphi_6 \\
\mathcal{E}\;M & = & ((\varphi_7\to\varphi_8)\to\varphi_9)\to(\varphi_7\to\varphi_8)\to\varphi_8.
\end{array}
$$

To obtain $pp_{Env}\,(M\,(S\,x\,y))$, we assign types to nodes in the tree in the following way. Let

$$
\begin{array}{lcl}
A & = & ((\varphi_1\to\varphi_2)\to\varphi_4\to\varphi_3)\to((\varphi_1\to\varphi_2)\to\varphi_4)\to(\varphi_1\to\varphi_2)\to\varphi_3, \text{ and} \\
\Gamma & = & \{x{:}(\varphi_1\to\varphi_2)\to\varphi_4\to\varphi_3, y:(\varphi_1\to\varphi_2)\to\varphi_4\}
\end{array}
$$

$$
\dfrac{\dfrac{}{\Gamma\vdash M{:}\,((\varphi_1\to\varphi_2)\to\varphi_3)\to(\varphi_1\to\varphi_2)\to\varphi_2}\quad \dfrac{\dfrac{\dfrac{}{\Gamma\vdash S{:}\,A}\quad\dfrac{}{\Gamma\vdash x{:}\,(\varphi_1\to\varphi_2)\to\varphi_4\to\varphi_3}}{\Gamma\vdash S\,x{:}\,((\varphi_1\to\varphi_2)\to\varphi_4)\to(\varphi_1\to\varphi_2)\to\varphi_3}\quad\dfrac{}{\Gamma\vdash y{:}\,(\varphi_1\to\varphi_2)\to\varphi_4}}{\Gamma\vdash S\,x\,y{:}\,(\varphi_1\to\varphi_2)\to\varphi_3}}{\Gamma\vdash M\,(S\,x\,y){:}\,(\varphi_1\to\varphi_2)\to\varphi_2}
$$

If the right-hand side term of the rewrite rule should be typed with $(\varphi_1\to\varphi_2)\to\varphi_2$, the type needed for y is $(\varphi_1\to\varphi_2)\to\varphi_1$ where

$$
\begin{array}{lcl}
B & = & ((\varphi_1\to\varphi_2)\to\varphi_1\to\varphi_2)\to((\varphi_1\to\varphi_2)\to\varphi_1)\to(\varphi_1\to\varphi_2)\to\varphi_2, \text{ and} \\
\Gamma' & = & \{y:(\varphi_1\to\varphi_2)\to\varphi_1\}
\end{array}
$$

$$
\dfrac{\dfrac{\dfrac{}{\Gamma'\vdash S{:}\,B}\quad\dfrac{}{\Gamma'\vdash I{:}\,(\varphi_1\to\varphi_2)\to\varphi_1\to\varphi_2}}{\Gamma'\vdash S\,I{:}\,((\varphi_1\to\varphi_2)\to\varphi_1)\to(\varphi_1\to\varphi_2)\to\varphi_2}\quad\dfrac{}{\Gamma'\vdash y{:}\,(\varphi_1\to\varphi_2)\to\varphi_1}}{\Gamma'\vdash S\,I\,y{:}\,(\varphi_1\to\varphi_2)\to\varphi_2}
$$

Take the term $M\,(S\,K\,I)$, which rewrites to $S\,I\,I$. Although the first term is typeable, with

$$
\begin{array}{lcl}
C & = & \varphi_4\to\varphi_5 \\
D & = & (C\to C\to C)\to(C\to C)\to C\to C
\end{array}
$$

$$
\dfrac{\dfrac{}{\emptyset\vdash M{:}\,(C\to C)\to C\to\varphi_5}\quad\dfrac{\dfrac{\dfrac{}{\emptyset\vdash S{:}\,D}\quad\dfrac{}{\emptyset\vdash K{:}\,C\to C\to C}}{\emptyset\vdash S\,K{:}\,(C\to C)\to C\to C}\quad\dfrac{}{\emptyset\vdash I{:}\,C\to C}}{\emptyset\vdash S\,K\,I{:}\,C\to C}}{\emptyset\vdash M\,(S\,K\,I){:}\,(\varphi_4\to\varphi_5)\to\varphi_5}
$$

the term $S\,I\,I$ is not typeable with the type $(\varphi_4\to\varphi_5)\to\varphi_5$. In fact, it is not typeable at all: $pp_{Env}\,(S\,I\,I)$ fails on unification.

So this rule is not safe, and should therefore be rejected.

The problem with the above rewrite system is that the principal pair for the left-hand side of the rule that defines M is not a valid pair for the right-hand side; the latter is only typeable if the types in the context are changed further. Now, when typing an *instance* of the left-hand side, we have no knowledge of the rule, and will type this term as it stands; the changes enforced by the right-hand side will not be applied, which, in this case, leads to an untypeable term being produced by reduction.

We solve this problem by rejecting rules that would pose extra restrictions while typing the right-hand side. In the following theorem, we prove that our solution is correct. For this we need the following lemma that formulates the relation between replacements performed on a term and possible type assignments for that term.

Lemma 7.18 (Replacement lemma) *i) If $pp_{Env}\, t = \langle \Pi, P\rangle$, and for the replacement* R *there are Γ, A such that $\Gamma \vdash t^{\mathrm{R}} : A$, then there is a substitution S, such that $S\,P = A$, and, for every statement $x{:}C \in \Pi$: $\Gamma \vdash x^{\mathrm{R}} : S\,C$.*

ii) If $\Gamma \vdash t : A$, and R *is a replacement and Γ' a context such that for every statement $x{:}C \in \Gamma$: $\Gamma' \vdash x^{\mathrm{R}} : C$, then $\Gamma' \vdash t^{\mathrm{R}} : A$.*

Proof: By induction on the structure of t.

i) *a*) $t \equiv x$, then $\Pi = x{:}\varphi$, and $P = \varphi$. Take $S = (\varphi \mapsto A)$. By assumption, $\Gamma \vdash x^{\mathrm{R}} : A$, so for every $x{:}C$ in Γ we have $\Gamma \vdash x^{\mathrm{R}} : S\,C$.

b) $t \equiv F$, then $\Pi = \emptyset$, and $P = \mathcal{E}F$. By $(\mathcal{F})$ there exists a substitution S_0 such that $A = S_0\,(\mathcal{E}\,F)$, and we have $\Gamma \vdash F^{\mathrm{R}} : A$ for every context Γ.

c) $t \equiv t_1\, t_2$. Let φ be a type-variable not occurring in any other type. If $pp_{Env}\,(t_1\, t_2) = \langle \Pi, P\rangle$, then for $i = 1,2$, there are $\langle \Pi_i, P_i\rangle$ (disjoint), such that $pp_{Env}\,(t_i) = \langle \Pi_i, P_i\rangle$. By induction, for $i = 1,2$, there is a Curry-substitution S_i such that $S_i\,(P_i) = A_i$, and, for every $x{:}A' \in \Pi_i$, $\Gamma \vdash x^{\mathrm{R}} : S_i\,(A')$. Notice that S_1 and S_2 do not interfere in that they are defined on separate sets of type variables. Take $S' = S_2 \circ S_1 \circ (\varphi \mapsto A)$, then, for every $x{:}A' \in \Pi_1 \cup \Pi_2$, $\Gamma \vdash x^{\mathrm{R}} : S'\,(A')$, and $S'\,(\varphi) = A$. By property 2.14 there are Curry-substitutions S and S_g such that

$$S_g = unify\,(\mathcal{E}\,F, P_1 \rightarrow \varphi), \text{ and } S' = S \circ S_g \text{ and } \langle \Pi, P\rangle = S_g\,(\langle \Pi_1 \cup \Pi_2, \varphi\rangle).$$

Then, for every $x{:}B' \in S_g(\Pi_1 \cup \Pi_2)$, $\Gamma \vdash x^{\mathrm{R}} : S\,(B')$, and $S\,(S_g\,(\varphi)) = A$.

ii) *a*) $t \equiv x, F$. Trivial.

b) $t \equiv t_1\, t_2$. Then there exists B, such that $\Gamma \vdash t_1 : B \rightarrow A$ and $\Gamma \vdash t_2 : B$. By induction, $\Gamma' \vdash {t_1}^{\mathrm{R}} : B \rightarrow A$ and $\Gamma' \vdash {t_2}^{\mathrm{R}} : B$. So, by $(\rightarrow E)$, we obtain $\Gamma' \vdash t_1\, {t_2}^{\mathrm{R}} : A$. ∎

Theorem 7.19 (Subject Reduction Theorem) *If $\Gamma \vdash t : A$ and $t \rightarrow t'$, then $\Gamma \vdash t' : A$.*

Proof: Let $l \rightarrow r$ be the typeable rewrite rule applied in the rewrite step $t \rightarrow t'$. We will prove that for every term substitution R and type D, if $\Gamma \vdash l^{\mathrm{R}} : D$, then $\Gamma \vdash r^{\mathrm{R}} : D$, which proves the theorem. Since **r** is typeable, there are Π, P such that $\langle \Pi, P\rangle$ is a principal pair for l with respect to $\mathcal{E}$, and $\Pi \vdash r : P$. Suppose R is a term substitution such that $\Gamma \vdash l^{\mathrm{R}} : D$. By Lemma 7.18*i*) there is a Γ' such that for every $x{:}C \in \Gamma'$, $\Gamma \vdash x^{\mathrm{R}} : C$, and $\Gamma' \vdash l : D$. Since $\langle \Pi, P\rangle$ is a principal pair for l with respect to $\mathcal{E}$, by Definition 7.11 there is a substitution S such that $S\,\langle \Pi, P\rangle = \langle \Gamma', D\rangle$. Since $\Pi \vdash r : P$, by Theorem 7.12 also $\Gamma' \vdash r : D$. Then by Lemma 7.18*ii*) $\Gamma \vdash r^{\mathrm{R}} : D$. ∎

7.5 An example

This example is taken from Dershowitz and Jouannaud [25]. It deals with the very well known definition of stacks of natural numbers and contains next to the operations *Top* and *Pop*, the operation *Alternate* that combines two stacks. Stacks of natural number are presented as terms of the form

$Push(s_1, (Push(s_2, \ldots, Push(s_n, \Lambda) \ldots)),$

where Λ is the empty stack, and the s_i denote representations of natural numbers, 0, $Succ(0)$, $Succ(Succ(0))$, etc.

The precise syntax of these representations is given in the following inductive way:

$$\begin{array}{lcl} Zero &=& 0 \\ Nat &=& Zero \cup Succ(Nat) \\ Empty &=& \Lambda \\ Stack &=& Empty \cup Push(Nat, Stack) \end{array}$$

Push and Λ can be seen as stack constructors, and *Zero* and *Succ* can be seen as number constructors. Operational semantics of the functions is given by the following rules:

$$\begin{array}{lcl} Top(Push(x,y)) &\rightarrow& x \\ Pop(Push(x,y)) &\rightarrow& y \\ Alternate(\Lambda,z) &\rightarrow& z \\ Alternate(Push(x,y),z) &\rightarrow& Push(x, Alternate(z,y)) \end{array}$$

With these rules, it can be shown, for example, that

$$Alternate(Push(Top(Push(0,\Lambda)),\Lambda), Pop(Push(Succ(0),\Lambda))) \rightarrow^* Push(0,\Lambda).$$

Under the assumption that *Nat* and *Stack* are type constants and that

$$\begin{array}{lcl} \mathcal{E}\ 0 &=& Nat, \\ \mathcal{E}\ Succ &=& Nat \rightarrow Nat, \\ \mathcal{E}\ L &=& Stack, \\ \mathcal{E}\ Push &=& Nat \rightarrow Stack \rightarrow Stack, \\ \mathcal{E}\ Top &=& Stack \rightarrow Nat \\ \mathcal{E}\ Pop &=& Stack \rightarrow Stack \\ \mathcal{E}\ Alternate &=& Stack \rightarrow Stack \rightarrow Stack. \end{array}$$

the presented rewrite rules are typeable.

7.6 A type check algorithm for TRSs

In this section we present a type check algorithm, as first presented in [9], that, when applied to a TRS and an environment determines whether this TRS is typeable with respect to the environment.

The goal of the type check algorithm presented below is to determine whether a type assignment can be constructed such that all the conditions of Definitions 7.9 and 7.15 are satisfied. The main function of the algorithm, called *TypeTRS*, expects a TRS as well as an environment as parameters. It returns a boolean that indicates whether the construction of the type assignment was successful.

It is easy to prove that the algorithm presented here is correct and complete:

Theorem 7.20 *i) If t is typeable with respect to $\mathcal{E}$, then TypeTerm $t\,\mathcal{E}$ returns $pp_{Env}\,t$.*
ii) If TypeTerm $t\,\mathcal{E}$ returns the pair $\langle \Gamma, A \rangle$, then $pp_{Env}\,t = \langle \Gamma, A \rangle$.
*iii) There is a type assignment with respect to $\mathcal{E}$ for the TRS **R**, if and only if TypeRules **R** $\mathcal{E}$.*

Proof: By straightforward induction on the structure of terms and rewrite rules. ■

The algorithm does not perform any consistency check on its input so it assumes the input to be correct according to Definitions 7.1 and 7.2. Moreover, all possible error messages and error handling cases are omitted, and the algorithm *TypeRules* returns only **true** for rewrite systems that are typeable. It could easily be extended to an algorithm that rejects untypable rewrite rules. Notice that, below, a TRS is a pair of rules and term; as in Λ^{N} and Λ^{NR}, the term is there in order for the TRS to become a program rather than a collection of procedures.

The type of a symbol is either an instance of the type for that symbol given by the environment (in case of a symbol) or that type itself (in case of a defined symbol). The distinction between the two is determined by the function *TypeTerm*.

```
$TypeTerm\ x\quad \mathcal{E} \rightarrow fresh$
$TypeTerm\ t_1t_2\ \mathcal{E} \rightarrow S\,\langle \Gamma_1 \cup \Gamma_2, \varphi\rangle$
    where $\varphi = fresh$
          $\langle \Gamma_1, B\rangle = TypeTerm\ t_1\ \mathcal{E}$
          $\langle \Gamma_2, A\rangle = TypeTerm\ t_2\ \mathcal{E}$
          $S = unify\ B\ (A\rightarrow\varphi)$

$TypeTerm\ F\ \mathcal{E} \rightarrow Freeze\,(\mathcal{E}\ F)$,          if this is defining occurrence of $F$
                      $FreshInstance\,(\mathcal{E}\ F)$,  otherwise
```

Notice that the call '*Freeze* $(\mathcal{E}\ F)$' is needed to avoid simply producing $\mathcal{E}\ F$, since it would mean that the type variables in the environment change because of unification. However, the defining symbol of a rewrite rule can only be typed with *one* type, so any substitution resulting from a unification is forbidden to change this type. We can ensure this by using 'non-unifiable' type variables; the non-specified function *Freeze* replaces all type variables by non-unifiable type variables. The unification algorithm should be extended in such a way that all the type variables that are not new (so they appear in some environment type) are recognized, so that it refuses to substitute these variables by other types.

TypeRule takes care of checking the safeness constraint as given in Definition 7.15, by checking if the unification of left and right-hand sides of a rewrite rule has changed the left-hand side context.

```
$TypeRule\ (l \rightarrow r)\ \mathcal{E} \rightarrow (S_2\,(S_1\,\Gamma_l)) = \Gamma_l$
    where $S_2 = UnifyContexts\ (S_1\,\Gamma_l)\ (S_1\,\Gamma_r)$,
          $S_1 = unify\ A\ B$,
          $\langle \Gamma_l, A\rangle = TypeTerm\ l\ \mathcal{E}$,
          $\langle \Gamma_r, B\rangle = TypeTerm\ r\ \mathcal{E}$
$TypeRules\ [\,]\quad \mathcal{E} \rightarrow$ true
$TypeRules\ [\mathbf{r} \mid \mathbf{R}]\ \mathcal{E} \rightarrow (TypeRule\ \mathbf{r}\ \mathcal{E})\ \&\ (TypeRules\ \mathbf{R}\ \mathcal{E})$
```

and the procedure that type checks the program:

```
$TypeTRS\ \langle \mathbf{R} : t\rangle\ \mathcal{E} \rightarrow TypeTerm\ t\ \mathcal{E}$, if $TypeRules\ \mathbf{R}\ \mathcal{E}$
```

8 Other approaches to type systems

In this section, we will have a brief look at some alternatives to the Curry-types approach.

8.1 Church's typed Lambda Calculus

An other approach to dealing with types in programming is that of *typed* systems. These differ from *type assignment* systems in that they do not deal with combining terms and types, but rather 'decorate' terms with types. The name most commonly associated with these systems is that of A. Church.

For example, the typed variant of Curry type assignment is defined as follows. First, the set of types in this system is exactly like that of Definition 2.1. But terms are defined in a different way.

Definition 8.1 (Typed Lambda terms) $M ::= x \mid (M_1 \cdot M_2) \mid (\lambda x^A . M)$

As before, we will write $\lambda x^A.y^B.M$ for $\lambda x^A.\lambda y^B.M$.

Definition 8.2 *Church's typed system* is defined by the following derivation rules.

$$(Ax): \frac{}{\Gamma, x{:}A \vdash_{\text{CH}} x : A}$$

$$(\to I): \frac{\Gamma, x{:}A \vdash_{\text{CH}} M : B}{\Gamma \vdash_{\text{CH}} \lambda x^A.M : A \to B}$$

$$(\to E): \frac{\Gamma \vdash_{\text{CH}} M_1 : A \to B \quad \Gamma \vdash_{\text{CH}} M_2 : A}{\Gamma \vdash_{\text{CH}} M_1 M_2 : B}$$

Exercise 8.3 Check the following statements:

i) $\emptyset \vdash_{\text{CH}} \lambda x^A.x : A \to A$.
ii) $\emptyset \vdash_{\text{CH}} \lambda x^A.y^B.x : A \to B \to A$.
iii) $\emptyset \vdash_{\text{CH}} \lambda x^{A \to B \to C}.y^{A \to B}.z^A.xz(yz) : (A \to B \to C) \to (A \to B) \to A \to C$.
iv) $\emptyset \vdash_{\text{CH}} \lambda b^A.c^B.c : A \to B \to B$.
v) $\emptyset \vdash_{\text{CH}} \lambda b^{B \to A}.c^B.(\lambda y^A.c)(bc) : (B \to A) \to B \to B$.
vi) $\emptyset \vdash_{\text{CH}} \lambda b^{B \to A}.\lambda c^B.(\lambda x^B.y^A.x)c(bc) : (B \to A) \to B \to B$.
vii) $\emptyset \vdash_{\text{CH}} (\lambda a^{B \to A \to B}.b^{B \to A}.c^B.ac(bc))(\lambda x^B.y^A.x) : (B \to A) \to B \to B$.

The whole problem of finding a type for a term becomes much more easy in this system, which explains why typed systems have been so widely studied.

Since all terms contain some kind of type information, the set of types that can be derived for terms is small (only *one* for a term without free variables). So notions like polymorphism have no place here, and these systems are therefore more studied in the context of *strongly typed programming languages* and *theorem provers*.

8.2 System F: The Polymorphic Lambda Calculus

In this section, we will focus on the System F approach to polymorphism in its untyped variant. System F (also known as the Polymorphic Lambda Calculus) was invented independently by Jean-Yves Girard [?] and John C. Reynolds [?]. It was originally introduced as an extension of the *typed* LC, which adds two new constructs to the syntax. In terms of logical proofs (which motivated Girard), these correspond to the $\forall\mathcal{I}$ and $\forall\mathcal{E}$ natural deduction rules. In terms of a programming language (as studied by Reynolds), they explicitly represent polymorphism in the language.

As a typed calculus, the variables of a term each carry a type in the syntax, and from these it is possible to derive the unique type of any particular term deterministically. In this section, we will briefly visit the untyped variant.

Definition 8.4 (UNTYPED SYSTEM F) *i*) $\mathcal{T}_\forall$, the set of *polymorphic types*, ranged over by $A, B, \ldots$, is defined over a set of *type-variables* $\Phi = \{\varphi_1, \varphi_2, \varphi_3, \ldots\}$ by:

$$A \ ::= \ \varphi \mid A \rightarrow B \mid \forall \varphi. A$$

ii) *Type assignment* is defined by the following natural deduction system.

$$(Ax): \ \dfrac{}{\Gamma \vdash_\forall x : A} \ (x{:}A \in \Gamma)$$

$$(\forall I): \ \dfrac{\Gamma \vdash_\forall M : A}{\Gamma \vdash_\forall M : \forall \varphi. A} \ (*)$$

$$(\forall E): \ \dfrac{\Gamma \vdash_\forall M : \forall \varphi. A}{\Gamma \vdash_\forall M : A[B/\varphi]}$$

$$(\rightarrow I): \ \dfrac{\Gamma, x{:}A \vdash_\forall M : B}{\Gamma \vdash_\forall \lambda x. M : A \rightarrow B}$$

$$(\rightarrow E): \ \dfrac{\Gamma \vdash_\forall M_1 : A \rightarrow B \quad \Gamma \vdash_\forall M_2 : A}{\Gamma \vdash_\forall M_1 M_2 : B}$$

$(*)$ If φ does not occur free in Γ.

This system is an extension (with general quantification) of the ML-style of polymorphism.

All terms typeable in this system are strongly normalisable (for a proof, see [?]), but not all strongly normalisable terms are typeable (see [?] for an example).

Type assignment in this system is not decidable [?]. It is also not clear if the system has the principal type property, since, for example, there is no known way to obtain the types $(\forall \varphi. \varphi) \rightarrow (\forall \varphi. \varphi)$ and $(\forall \varphi. \varphi \rightarrow \varphi) \rightarrow (\forall \varphi. \varphi \rightarrow \varphi)$ – both types for the λ-term $\lambda x. xx$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}\forall\varphi.\varphi \vdash_\forall x : \forall\varphi.\varphi}}{x{:}\forall\varphi.\varphi \vdash_\forall x : \varphi' \rightarrow \varphi'} \quad \dfrac{\dfrac{}{x{:}\forall\varphi.\varphi \vdash_\forall x : \forall\varphi.\varphi}}{x{:}\forall\varphi.\varphi \vdash_\forall x : \varphi'}}{x{:}\forall\varphi.\varphi \vdash_\forall xx : \varphi'}}{x{:}\forall\varphi.\varphi \vdash_\forall xx : \forall\varphi'.\varphi'}}{\emptyset \vdash_\forall \lambda x.xx : (\forall\varphi.\varphi) \rightarrow (\forall\varphi'.\varphi')}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}\forall\varphi.\varphi\rightarrow\varphi \vdash_\forall x : \forall\varphi.\varphi\rightarrow\varphi}}{x{:}\forall\varphi.\varphi\rightarrow\varphi \vdash_\forall x : (\varphi' \rightarrow \varphi') \rightarrow \varphi' \rightarrow \varphi'} \quad \dfrac{\dfrac{}{x{:}\forall\varphi.\varphi\rightarrow\varphi \vdash_\forall x : \forall\varphi.\varphi\rightarrow\varphi}}{x{:}\forall\varphi.\varphi\rightarrow\varphi \vdash_\forall x : \varphi' \rightarrow \varphi'}}{x{:}\forall\varphi.\varphi\rightarrow\varphi \vdash_\forall xx : \varphi' \rightarrow \varphi'}}{x{:}\forall\varphi.\varphi\rightarrow\varphi \vdash_\forall xx : \forall\varphi'.\varphi' \rightarrow \varphi'}}{\emptyset \vdash_\forall \lambda x.xx : (\forall\varphi.\varphi\rightarrow\varphi) \rightarrow (\forall\varphi'.\varphi' \rightarrow \varphi')t}$$

from a unique type (see [?]). Notice that there exists no type A derivable for $\lambda x.xx$ such that both types can be obtained from A by substitution.

Notice that we can also derive (where $\Gamma = \{x{:}\forall\varphi.\varphi\}$):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_\forall x : \forall\varphi.\varphi}}{\Gamma \vdash_\forall x : (\varphi' \rightarrow \varphi') \rightarrow \varphi' \rightarrow \varphi'} \quad \dfrac{\dfrac{}{\Gamma \vdash_\forall x : \forall\varphi.\varphi}}{\Gamma \vdash_\forall x : \varphi' \rightarrow \varphi'}}{\Gamma \vdash_\forall xx : \varphi' \rightarrow \varphi'}}{\Gamma \vdash_\forall xx : \forall\varphi'.\varphi' \rightarrow \varphi'}}{\emptyset \vdash_\forall \lambda x.xx : (\forall\varphi.\varphi) \rightarrow (\forall\varphi'.\varphi' \rightarrow \varphi')}$$

This system can be extended with *subtyping* and *bounded second-order polymorphism*. As first proposed by Cardelli and Wegner [?], it has been widely studied as a core calculus for type systems with subtyping. Curien and Ghelli [?] proved the partial correctness of a recursive procedure for computing minimal types of $F_\leq$ terms and showed that the termination of this procedure is equivalent to the termination of its major component, a procedure for checking the subtype relation between $F_\leq$ types. This procedure was thought to terminate on all inputs, but the discovery of a subtle bug in a 'proof' of this claim put doubts on the question of the decidability of subtyping, and hence of typechecking. This question was answered negatively by Pierce [?] by showing that the subtype relation of $F_\leq$ is undecidable.

8.3 Featherweight Java

In this subsection we will focus on Featherweight Java (FJ) [?], a restriction of Java defined by removing all but the most essential features of the full language; Featherweight Java bears a similar relation to Java as LC does to languages such as ML [?] and Haskell [32]. We illustrate the expressive power of our calculus by showing that it is Turing complete through an embedding of Combinatory Logic (CL) – and thereby also LC.

As in other class-based object-oriented languages, FJ defines *classes*, which represent abstractions encapsulating both data (stored in *fields*) and the operations to be performed on that data (encoded as *methods*). Sharing of behaviour is accomplished through the *inheritance* of fields and methods from parent classes. Computation is mediated by *instances* of these classes (called *objects*), which interact with one another by *calling* (also called *invoking*) methods on each other and accessing each other's (or their own) fields. We have removed cast expressions since, as the authors of [?] themselves point out, the presence of *downcasts* is unsound[2]; for this reason we call our calculus $\text{FJ}^{\not c}$. We also leave the constructor method as implicit.

As is usual, we distinguish the class name `Object` (which denotes the root of the class inheritance hierarchy in all programs) and the self variable `this`[3], used to refer to the receiver object in method bodies.

Definition 8.5 ($\text{FJ}^{\not c}$ SYNTAX) $\text{FJ}^{\not c}$ programs P consist of a *class table* $\mathcal{CT}$, comprising the *class declarations*, and an *expression* e to be run (corresponding to the body of the `main` method in a real Java program). They are defined by the grammar:

$$\begin{array}{rcl}
e & ::= & x \mid \texttt{this} \mid \texttt{new C}(\vec{e}) \mid e.f \mid e.m(\vec{e}) \\
fd & ::= & \texttt{C}\, f; \\
md & ::= & \texttt{D}\ m(\texttt{C}_1\ x_1,\ \ldots,\ \texttt{C}_n\ x_n)\ \{\texttt{return}\ e;\} \\
cd & ::= & \texttt{class C extends C}'\ \{\vec{fd}\ \vec{md}\} \qquad (\texttt{C} \neq \texttt{Object}) \\
\mathcal{CT} & ::= & \vec{cd} \\
P & ::= & (\mathcal{CT}, e)
\end{array}$$

Notice that the language is first-order.

From this point, for readability, all the concepts defined are program dependent (or more precisely, parametric on the class table); however, since a program is essentially a fixed entity, it will be left as an implicit parameter in the definitions that follow. We also only consider programs which conform to some sensible well-formedness criteria: no cycles in the inheritance hierarchy, and fields and methods in any given branch of the inheritance

[2]In the sense that typeable expressions can get stuck at runtime.

[3]Not a variable in the traditional sense, since it is not used to express a position in the method's body where a parameter can be passed.

hierarchy are uniquely named. An exception is made to allow the redeclaration of methods, providing that only the *body* of the method is allowed to differ from the previous declaration (in the parlance of class-based OO, this is called *method override*).

We define the following functions to look up elements of class definitions.

Definition 8.6 (LOOKUP FUNCTIONS) The following lookup functions are defined to extract the names of fields and bodies of methods belonging to (and inherited by) a class.

i) The following functions retrieve the name of a class or field from its definition:

$$\begin{aligned} \textsc{cn}(\texttt{class C extends D } \{\overrightarrow{fd}\ \overrightarrow{md}\}) &= \texttt{C} \\ \textsc{fn}(\texttt{C}\ f) &= f \end{aligned}$$

ii) By abuse of notation, we will treat the *class table*, $\mathcal{CT}$, as a partial map from class names to class definitions:

$$\mathcal{CT}(\texttt{C}) = cd \text{ if } \textsc{cn}(cd) = \texttt{C} \text{ and } cd \in \mathcal{CT}$$

iii) The list of fields belonging to a class C (including those it inherits) is given by the function $\mathcal{F}$, which is defined as follows:

$$\begin{aligned} \mathcal{F}(\texttt{Object}) &= \epsilon \\ \mathcal{F}(\texttt{C}) &= \mathcal{F}(\texttt{C}')\cdot\vec{f} \text{ if } \mathcal{CT}(\texttt{C}) = \texttt{class C extends C}' \ \{\overrightarrow{fd}\ \overrightarrow{md}\} \\ &\qquad \text{and } \textsc{fn}(fd_i) = f_i \text{ for all } i \in \overline{n}. \end{aligned}$$

iv) The function *Mb*, given a class name C and method name *m*, returns a tuple $(\vec{x},e)$, consisting of a sequence of the method's formal parameters and its body:

$$\begin{aligned} Mb(\texttt{C},m) &= (\vec{x},e) && \text{if } \mathcal{CT}(\texttt{C}) = \texttt{class C extends C}'\ \{\overrightarrow{fd}\ \overrightarrow{md}\}\text{, and there exist} \\ &&& \texttt{C}_0,\vec{\texttt{C}} \text{ such that } \texttt{C}_0\ m(\texttt{C}_1\ x_1,\ldots,\texttt{C}_n\ x_n)\ \{\texttt{return } e;\} \in \overrightarrow{md}. \\ Mb(\texttt{C},m) &= Mb(\texttt{C}',m) && \text{if } \mathcal{CT}(\texttt{C}) = \texttt{class C extends C}'\ \{\overrightarrow{fd}\ \overrightarrow{md}\}\text{, and there are no} \\ &&& \texttt{C}_0,\ \vec{\texttt{C}},\ \vec{x},\ e \text{ such that } \texttt{C}_0\ m(\texttt{C}_1\ x_1,\ldots,\texttt{C}_n\ x_n)\ \{\texttt{return } e;\} \in \overrightarrow{md}. \end{aligned}$$

v) The function VARS returns the set of variables used in an expression.

We impose the additional criterion that well-formed programs satisfy the following property:

$$\text{if } Mb(\texttt{C},m) = (\vec{x},e_b) \text{ then } \textsc{vars}(e_b) \setminus \{\texttt{this}\} \subseteq \{x_1,\ldots,x_n\}$$

As is clear from the definition of classes, if `class C extends D` $\{\vec{f}\ \vec{m}\}$ is a class declaration, then `new C`$(\vec{e})$ creates an object where the expressions e_i is attached to the field f_i.

As for term rewriting, *substitution* of expressions for variables is the basic mechanism for reduction in our calculus: when a method is invoked on an object (the *receiver*) the invocation is replaced by the body of the method that is called, and each of the variables is replaced by a corresponding argument.

Definition 8.7 (REDUCTION) The reduction relation $\rightarrow$ is the smallest relation on expressions satisfying:

$$\begin{aligned} \texttt{new C}(\vec{e}).f_i &\rightarrow e_i && \text{for class name C with } \mathcal{F}(\texttt{C}) = \vec{f} \text{ and } i \in \overline{n}, \\ \texttt{new C}(\vec{e}).m(\overrightarrow{e'_n}) &\rightarrow e^{\mathsf{S}} && \text{for class name C and method } m \text{ with } Mb(\texttt{C},m) = (\vec{x},e), \\ &&& \text{where } \mathsf{S} = \{\ \texttt{this} \mapsto \texttt{new C}(\vec{e}),\ \mathsf{x}_1 \mapsto e'_1,\ \ldots,\ \mathsf{x}_n \mapsto e'_n\ \} \end{aligned}$$

We add the usual congruence rules for allowing reduction in subexpressions, and the reflexive and transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$.

Type assignment in FJ$^{\not{c}}$ is a relatively easy affair, and more or less guided by the class hierarchy.

Definition 8.8 ((Nominal) type assignment for FJ$^{\not{c}}$)

$$\begin{array}{rl}
(\texttt{var}): & \dfrac{}{\Gamma, x{:}\texttt{C} \vdash x : \texttt{C}} \\[2ex]
(\texttt{fld}): & \dfrac{\Gamma \vdash e : \texttt{D}}{\Gamma \vdash e.f : \texttt{C}} \ (\mathcal{FT}(\mathcal{EC},\texttt{D},f) = \texttt{C}) \\[2ex]
(\texttt{invk}): & \dfrac{\Gamma \vdash e : \texttt{C} \quad \Gamma \vdash e_i : \texttt{C}_i \ \ (\forall i \in \underline{n})}{\Gamma \vdash e.m(\vec{e}) : \texttt{D}} \ (\mathcal{MT}(\mathcal{EC},\texttt{C},m) = \vec{\texttt{C}} \rightarrow \texttt{D}) \\[2ex]
(\texttt{sub}): & \dfrac{\Gamma \vdash e : \texttt{C}'}{\Gamma \vdash e : \texttt{C}} \ (\texttt{C}' \lhd: \texttt{C}) \\[2ex]
(\texttt{new}): & \dfrac{\Gamma \vdash e_i : \texttt{C}_i \qquad (\forall i \in \underline{n})}{\Gamma \vdash \texttt{new C}(\vec{e}) : \texttt{C}} \ (\mathcal{F}(\mathcal{EC},\texttt{C}) = \vec{f} \ \& \ \mathcal{FT}(\mathcal{EC},\texttt{C},f_i) = \texttt{C}_i \ (\forall i \in \underline{n}))
\end{array}$$

where $(\texttt{var})$ is applicable to `this` as well and the subtype relation $\lhd:$ is defined as the transitive closure of the of class extension.

To emphasise the expressive power of FJ$^{\not{c}}$, we will now encode Combinatory Logic.

Definition 8.9 The encoding of Combinatory Logic (CL) into the FJ$^{\not{c}}$ program OOCL (Object-Oriented Combinatory Logic) is defined using the execution context given by:

```
class Combinator extends Object {
      Combinator app(Combinator x) { return this; } }
class K extends Combinator {
      Combinator app(Combinator x) { return new K1(x); } }
class K1 extends K {
      Combinator x;
      Combinator app(Combinator y) { return this.x; } }
class S extends Combinator {
      Combinator app(Combinator x) { return new S1(x); } }
class S1 extends S {
      Combinator x;
      Combinator app(Combinator y) { return new S2(this.x, y); } }
class S2 extends S1 {
      Combinator y;
      Combinator app(Combinator z) {
          return this.x.app(z).app(this.y.app(z)); } }
```

and the function $\lceil\cdot\rfloor$ which translates terms of CL into FJ$^{\not{c}}$ expressions, and is defined as follows:

$$\begin{array}{rcl}
\lceil x \rfloor & = & x \\
\lceil t_1 t_2 \rfloor & = & \lceil t_1 \rfloor\texttt{.app(}\lceil t_2 \rfloor\texttt{)} \\
\lceil \mathsf{K} \rfloor & = & \texttt{new K()} \\
\lceil \mathsf{S} \rfloor & = & \texttt{new S()}
\end{array}$$

We can easily verify that the reduction behaviour of OOCL mirrors that of CL.

Theorem 8.10 *If t_1, t_2 are terms of* CL *and $t_1 \rightarrow^* t_2$, then $\llbracket t_1 \rrbracket \rightarrow^* \llbracket t_2 \rrbracket$ in* OOCL.

Proof: By induction on the definition of reduction in CL; we only show the case for S:

$$\begin{array}{lr}
\llbracket \mathsf{S}\, t_1\, t_2\, t_3 \rrbracket & \triangleq \\
\texttt{((new S().app(}\llbracket t_1 \rrbracket\texttt{)).app(}\llbracket t_2 \rrbracket\texttt{)).app(}\llbracket t_3 \rrbracket\texttt{)} & \rightarrow \\
\texttt{((new S}_1\texttt{(}\llbracket t_1 \rrbracket\texttt{)).app(}\llbracket t_2 \rrbracket\texttt{)).app(}\llbracket t_3 \rrbracket\texttt{)} & \rightarrow \\
\texttt{(new S}_2\texttt{(this.x,y)).app(}\llbracket t_3 \rrbracket\texttt{)}\quad [\texttt{this} \mapsto \texttt{new S}_1\texttt{(}\llbracket t_1 \rrbracket\texttt{)},\ \texttt{y} \mapsto \llbracket t_2 \rrbracket] & = \\
\texttt{(new S}_2\texttt{(new S}_1\texttt{(}\llbracket t_1 \rrbracket\texttt{).x,}\llbracket t_2 \rrbracket\texttt{)).app(}\llbracket t_3 \rrbracket\texttt{)} & \rightarrow \\
\texttt{(new S}_2\texttt{(}\llbracket t_1 \rrbracket\texttt{,}\llbracket t_2 \rrbracket\texttt{).app(}\llbracket t_3 \rrbracket\texttt{)} & \rightarrow \\
\texttt{this.x.app(z).app(this.y.app(z))}\quad [\texttt{this} \mapsto \texttt{new S}_2\texttt{(}\llbracket t_1 \rrbracket\texttt{,}\llbracket t_2 \rrbracket\texttt{)},\ \texttt{z} \mapsto \llbracket t_3 \rrbracket] & = \\
\texttt{((new S}_2\texttt{(}\llbracket t_1 \rrbracket\texttt{,}\llbracket t_2 \rrbracket\texttt{).x.app(}\llbracket t_3 \rrbracket\texttt{)).app(new S}_2\texttt{(}\llbracket t_1 \rrbracket\texttt{.}\llbracket t_2 \rrbracket\texttt{).y.app(}\llbracket t_3 \rrbracket\texttt{))} & \rightarrow^* \\
\texttt{(}\llbracket t_1 \rrbracket\texttt{.app(}\llbracket t_3 \rrbracket\texttt{)).app((}\llbracket t_2 \rrbracket\texttt{).app(}\llbracket t_3 \rrbracket\texttt{))} & \triangleq \\
\llbracket t_1\, t_3\, (t_2\, t_3) \rrbracket &
\end{array}$$

The case for K is similar, and the rest is straightforward. ■

Given the Turing completeness of CL, this result shows that FJ$^{\not{c}}$ is also Turing complete.

The nominal type system for Java is the accepted standard; many researchers are looking for more expressive type systems, that deal with intricate details of object oriented programming, and in particular with side effects. We briefly study here a *functional* system, that allows for us to show a preservation result.

Definition 8.11 (FUNCTIONAL TYPE ASSIGNMENT FOR FJ$^{\not{c}}$)

- Functional types for FJ are defined by:

  $$A, B \ ::= \ \texttt{C} \mid \phi \mid \langle f_1{:}A,\ \ldots,\ f_n{:}A,\ m_1{:}\overrightarrow{A} \rightarrow B,\ \ldots,\ m_k{:}\overrightarrow{C} \rightarrow D \rangle \ (n+k \geq 1)$$

  We will write R for record types, ℓ for aribtrary labels, $\langle \ell{:}A \rangle \in R$ when $\ell{:}A$ occurs in R, and assume that all labels are distinct in records.
- A context is a mapping from term variables (including `this`) to types.
- Functional type assignment is defined through:

$$(\texttt{var}) : \dfrac{}{\Gamma, x{:}A \vdash x : A}$$

$$(\texttt{invk}) : \dfrac{\Gamma \vdash e : \langle m{:}(A_1,\ldots,A_n) \to B\rangle \quad \Gamma \vdash e_1 : A_1 \ \cdots \ \Gamma \vdash e_n : A_n}{\Gamma \vdash e.m\,(\overrightarrow{e_n}) : B}$$

$$(\texttt{fld}) : \dfrac{\Gamma \vdash e : \langle f{:}A\rangle}{\Gamma \vdash \texttt{e}.f : A}$$

$$(\texttt{newM}) : \dfrac{\texttt{this}{:}C, x_1{:}A_1,\ldots,x_n{:}A_n \vdash e_b : B \quad \Gamma \vdash \texttt{new C}(\vec{e}) : C}{\Gamma \vdash \texttt{new C}(\vec{e}) : \langle m{:}(A_1,\ldots,A_n) \to B\rangle} \ (\mathit{Mb}(\texttt{C},m) = (\vec{x},e_b))$$

$$(\texttt{newF}) : \dfrac{\Gamma \vdash e_1 : A_1 \quad \cdots \quad \Gamma \vdash e_n : A_n}{\Gamma \vdash \texttt{new C}(\overrightarrow{e_n}) : \langle f_i{:}A_i\rangle} \ (f_i \in \mathcal{F}(\texttt{C}))$$

$$(\texttt{newO}) : \dfrac{\Gamma \vdash e_1 : A_1 \quad \cdots \quad \Gamma \vdash e_n : A_n}{\Gamma \vdash \texttt{new C}(\overrightarrow{e_n}) : \texttt{C}} \ (\mathcal{F}(\texttt{C}) = \overrightarrow{f_n})$$

$$(\texttt{proj}) : \dfrac{\Gamma \vdash e : R}{\Gamma \vdash e : \langle \ell{:}A\rangle} \ (\ell{:}A \in R)$$

$$(\texttt{rec}) : \dfrac{\Gamma \vdash e : \langle \ell_1{:}A_1\rangle \quad \cdots \quad \Gamma \vdash e : \langle \ell_n{:}A_n\rangle}{\Gamma \vdash e : \langle \ell_1{:}A_1,\ \ldots,\ \ell_n{:}A_n\rangle}$$

Using a modified notion of unification, it is possible to define a notion of principal pair for FJ$^{\not{c}}$ expressions, and show completeness.

The elegance of this functional approach is that we can now link types assigned in functional languages to types assignable to object-oriented programs. To show type preservation, we need to define what the equivalent of Curry's types are in terms of our FJ$^{\not{c}}$ types. To this end, we define the following translation of Curry types.

Definition 8.12 (Type Translation) The function $[\![\cdot]\!]$, which transforms Curry types[4], is defined as follows:

$$\begin{array}{rcl}[\![\varphi]\!] & = & \varphi \\ [\![A \to B]\!] & = & \langle \mathsf{app}{:}([\![A]\!]) \to [\![B]\!]\rangle\end{array}$$

It is extended to contexts as follows: $[\![\Gamma]\!] = \{x{:}[\![A]\!] \mid x{:}A \in \Gamma\}$.

We can now show the type preservation result.

Theorem 8.1 (Preservation of Types) *If* $\Gamma \vdash_{\text{CL}} t{:}A$ *then* $[\![\Gamma]\!] \vdash [\![t]\!] : [\![A]\!]$.

Proof: By induction on the definition of derivations. The cases for (Ax) and $\to E$ are trivial. For the rules (K) and (S), take the following derivation schemas for assigning the translation of the respective Curry type schemes to the OOCL translations of K and S.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{this}{:}\langle \texttt{x}{:}B,\texttt{y}{:}C\rangle \vdash \texttt{this} : \langle \texttt{x}{:}B,\texttt{y}{:}C\rangle}\,(\texttt{var})}{\texttt{this}{:}\langle \texttt{x}{:}B\texttt{,y}{:}C\rangle \vdash \texttt{this} : \langle \texttt{x}{:}B\rangle}\,(\texttt{proj})}{\texttt{this}{:}\langle \texttt{x}{:}B,\texttt{y}{:}C\rangle \vdash \texttt{this.x} : B}\,(\texttt{fld}) \qquad \dfrac{\dfrac{}{\texttt{this}{:}\texttt{K},\texttt{x}{:}B \vdash \texttt{x} : B}\,(\texttt{var})}{\texttt{this}{:}\texttt{K},\texttt{x}{:}B \vdash \texttt{new K}_1\texttt{(x)} : \langle \texttt{x}{:}B\rangle}\,(\texttt{newF})}{\begin{array}{c}\texttt{this}{:}\texttt{K},\texttt{x}{:}B \vdash \texttt{K}_1\texttt{(x)} : \langle \mathsf{app}{:}(C) \to B\rangle \\ \vdots\end{array}}\,(\texttt{newM}) \qquad \dfrac{}{\varnothing \vdash \texttt{K} : \texttt{K}}\,(\texttt{obj})}{\varnothing \vdash \texttt{K} : \langle \mathsf{app}{:}(B) \to \langle \mathsf{app}{:}(C) \to B\rangle\rangle}\,(\texttt{newM})$$

and

[4]Note we have *overloaded* the notation $[\![\cdot]\!]$, which we also use for the translation of CL terms to FJ$^{\not{c}}$ expressions.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \texttt{this} : \langle \texttt{x}{:}\langle \texttt{app}{:}(B) \to \langle \texttt{app}{:}(C) \to D\rangle\rangle\rangle}\,(\texttt{var})}{\Pi \vdash \texttt{this.x} : \langle \texttt{app}{:}(B) \to \langle \texttt{app}{:}(C) \to D\rangle\rangle}\,(\texttt{fld}) \quad \dfrac{}{\Pi \vdash \texttt{z} : B}\,(\texttt{var})}{\Pi \vdash \texttt{this.x.app(z)} : \langle \texttt{app}{:}(C) \to D\rangle}\,(\texttt{newM}) \quad \overset{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \texttt{this} : \langle \texttt{y}{:}\langle \texttt{app}{:}(B) \to C\rangle\rangle}\,(\texttt{var})}{\Pi \vdash \texttt{this.y} : \langle \texttt{app}{:}(B) \to C\rangle}\,(\texttt{fld}) \quad \dfrac{}{\Pi \vdash \texttt{z} : B}\,(\texttt{var})}{\Pi \vdash \texttt{this.y.app(z)} : C}\,(\texttt{invk})}{\vdots}}{\Pi \vdash \texttt{this.x.app(z).app(this.y.app(z))} : D}\,(\texttt{invk}) \;\; \vdots \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Pi' \vdash \texttt{this} : \langle \texttt{x}{:}B_1\rangle}\,(\texttt{var})}{\Pi' \vdash \texttt{this.x} : B_1}\,(\texttt{fld})}{\Pi' \vdash \texttt{S}_2\texttt{(this.x,y)} : \langle \texttt{x}{:}B_1\rangle}\,(\texttt{newF}) \quad \overset{\dfrac{\dfrac{}{\Pi' \vdash \texttt{y} : B_2}\,(\texttt{var})}{\Pi' \vdash \texttt{S}_2\texttt{(this.xy)} : \langle \texttt{y}{:}B_2\rangle}\,(\texttt{newF})}{\vdots}}{\Pi' \vdash \texttt{S}_2\texttt{(this.x,y)} : \langle \texttt{x}{:}B_1\rangle \cap \langle \texttt{y}{:}B_2\rangle}\,(\texttt{join})}{\Pi' \vdash \texttt{S}_2\texttt{(this.x,y)} : \langle \texttt{app}{:}(B) \to D\rangle}\,(\texttt{newM}) \quad \overset{\dfrac{\dfrac{}{\texttt{this:S,x:}B_1 \vdash \texttt{x} : B_1}\,(\texttt{var})}{\texttt{this:S,x:}B_1 \vdash \texttt{S}_1\texttt{(x)} : \langle \texttt{x}{:}B_1\rangle}\,(\texttt{newF})}{\vdots}}{\texttt{this:S,x:}\sigma_1 \vdash \texttt{S}_1\texttt{(x)} : \langle \texttt{app}{:}(B_2) \to \langle \texttt{app}{:}(B) \to D\rangle\rangle}\,(\texttt{newM}) \quad \dfrac{}{\emptyset \vdash \texttt{S} : \texttt{S}}\,(\texttt{obj})}{\emptyset \vdash \texttt{S} : \langle \texttt{app}{:}(B_1) \to \langle \texttt{app}{:}(B_2) \to \langle \texttt{app}{:}(B) \to D\rangle\rangle\rangle}\,(\texttt{newM})$$

(where $B_1 = \langle \texttt{app}{:}(B) \to \langle \texttt{app}{:}(C) \to D\rangle\rangle$, and $B_2 = \langle \texttt{app}{:}(B) \to C\rangle$, $\Pi' = \texttt{this:}\langle \texttt{x:}B_1\rangle\texttt{,y:}B_2$, and $\Pi = \texttt{this:}\langle \texttt{x:}B_1\texttt{,y:}B_2\rangle\texttt{,z:}B$). ■

Furthermore, since Curry's well-known translation of the simply typed LC into CL preserves typeability, we also construct a type-preserving encoding of LC into $\text{FJ}^{\mathcal{C}}$.

9 Basic extensions to the type language

In this section we will briefly discusses a few basic extensions that must be made to obtain a traditional programming language, i.e. to add those type features that are consider basic: *data structures*, and *recursive types*[5].

9.1 Data structures

Introducing *products* and *sums* (or *disjoint union*) to our type language is straightforward. The grammar of types is extended as follows:

$$A,B \;::=\; \cdots \mid A \times B \mid A + B$$

The type $A \times B$ denotes a way of building a *pair* out of two components (left and right) with types A and B. The type $A + B$ describes *disjoint union* either via *left injection* applied to a value of type A, or *right injection* applied to a value of type B.

Like many simple language features, products and sums can be viewed either as new language constructs or simply as new constants, together with some syntactic structure.

Definition 9.1 (PAIRING) We extend the calculus with the following constructors

$$E \;::=\; \ldots \mid \langle E_1,E_2\rangle \mid \textsf{left}\ (E) \mid \textsf{right}\ (E)$$

with their type assignment rules:

[5]This section is in part based on [46]

$$(\mathsf{Pair}) : \frac{\Gamma \vdash E_1 : A \quad \Gamma \vdash E_2 : B}{\Gamma \vdash \langle E_1,E_2 \rangle : A \times B}$$

$$(\mathsf{left}) : \frac{\Gamma \vdash E : A \times B}{\Gamma \vdash \mathsf{left}\ (E) : A}$$

$$(\mathsf{right}) : \frac{\Gamma \vdash E : A \times B}{\Gamma \vdash \mathsf{right}\ (E) : B}$$

The reduction rules that come with these constants are:

$$\begin{array}{rcl} \mathsf{left}\ \langle E_1,E_2 \rangle & \to & E_1 \\ \mathsf{right}\ \langle E_1,E_2 \rangle & \to & E_2 \end{array}$$

We could be tempted to add the rule

$$\langle \mathsf{left}\ (E), \mathsf{right}\ (E) \rangle \ \to\ E$$

as well, but a difficulty with this in combination with the two projection rules is that it forms Klop's famous 'Surjective Pairing' example [36]; this cannot be expressed in LC because when added to LC, the Church-Rosser property no longer holds.

Definition 9.2 ((DISJOINT) UNION) We extend the calculus with the following constants

$$E ::= \ldots \mid \mathsf{case}\ (E_1,E_2,E_3) \mid \mathsf{inj}{\cdot}l\ (E) \mid \mathsf{inj}{\cdot}r\ (E)$$

with their type assignment rules:

$$(\mathsf{case}) : \frac{\Gamma \vdash E_1 : A + B \quad \Gamma \vdash E_2 : A{\to}C \quad \Gamma \vdash E_3 : B{\to}C}{\Gamma \vdash \mathsf{case}\ (E_1,E_2,E_3) : C}$$

$$(\mathsf{inj}{\cdot}l) : \frac{\Gamma \vdash E : A}{\Gamma \vdash \mathsf{inj}{\cdot}l\ (E) : A + B}$$

$$(\mathsf{inj}{\cdot}r) : \frac{\Gamma \vdash E : B}{\Gamma \vdash \mathsf{inj}{\cdot}r\ (E) : A + B}$$

Notice that the additional syntactic structure as added to the programming language acts as a marker, so that it is always possible to decide which part of the composite type was actually derived.

The reduction rules that come with these constants are:

$$\begin{array}{rcl} \mathsf{case}\ (\mathsf{inj}{\cdot}l\ (E_1),E_2,E_3) & \to & E_2\ E_1 \\ \mathsf{case}\ (\mathsf{inj}{\cdot}r\ (E_1),E_2,E_3) & \to & E_3\ E_1 \end{array}$$

Notice that application is used on the right-hand side of these rules.

9.2 Recursive types: the equi-recursive approach

A type built out of products, sums, and base types can only describe structures of finite size, and we cannot describe *lists*, *trees*, or other data structures of (potential) unbounded size. For this, some form of recursive types is needed. As a matter of fact, the informal definition

"a list is either empty or a pair of an element and a list"

is recursive.

Several computer programming languages provide a *unit* type to specify the argument type of a function that does not require arguments, and then we write $E : A$ rather than $E : unit{\rightarrow}A$. In the functional programming languages Haskell [32], and Clean [12], the *unit* type is called $()$ and its only value is also $()$, reflecting the 0-tuple interpretation. In ML, the type is called *unit* but the value is written as $()$. Using this approach here, we add the rule

$$(unit) : \frac{}{\Gamma \vdash () : unit}$$

Using pairing, we can express lists of type B via the equation

$$A \;=\; unit + (B \times A);$$

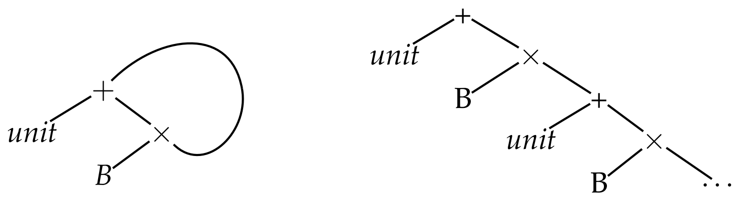
This is indeed a formalisation of the informal definition above. The most obvious way of introducing recursive types into a type system is to ensure that such a recursive equation admits a solution. This can be done by extending the grammar of types with:

$$A,B \;=\; \cdots \mid X \mid \mu X.A$$

and then the type described by the above equation is

$$\mu X.unit + (B \times X).$$

which corresponds to the graphs:



We can see these types as descriptions for infinite trees, where sub-trees are shaped like the tree itself, and we can generate these infinite trees by *unfolding* the recursive definition. Two recursive types A and B are said to be the same when their infinite unfoldings coincide; we then write $A =_\mu B$. Conditions on recursive types rule out meaningless types, such as $\mu X.X$, which (infinite) unfolding isn't well defined.

Two equal types can be used interchangeably: this is formalised by introducing a new typing rule:

$$(\mu) : \frac{\Gamma \vdash E : A}{\Gamma \vdash E : B} \; (A =_\mu B)$$

This rule is not syntax-directed (i.e. E does not change), so it can be applied at any point in a derivation.

Example 9.3 Assuming numbers, the type I for them, and pre-fix addition we can express the function that calculates the length of a list by:

$$LL \;=\; \textsf{Fix}\; ll.\lambda list.\textsf{case}\; (list,\lambda x.0,\lambda x.+\;1\;(ll(\textsf{right}\;x)))$$

Notice that now

$$\begin{array}{ll}
(\mathsf{Fix}\ ll.\lambda list.\mathsf{case}\ (list,\lambda x.0,\lambda x.+\,1\ (ll(\mathsf{right}\ x))))\ (\mathsf{inj}{\cdot}r\langle a,b\rangle) & \to\\
(\lambda list.\mathsf{case}\ (list,\lambda x.0,\lambda x.+\,1\ (LL\ (\mathsf{right}\ x))))\ (\mathsf{inj}{\cdot}r\langle a,b\rangle) & \to\\
\mathsf{case}\ (\mathsf{inj}{\cdot}r\langle a,b\rangle,\lambda x.0,\lambda x.+\,1\ (LL\ (\mathsf{right}\ x))) & \to\\
(\lambda x.+\,1\ (LL\ (\mathsf{right}\ x)))\ \langle a,b\rangle & \to\\
+\,1\ (LL\ (\mathsf{right}\ \langle a,b\rangle)) & \to\\
+\,1\ (LL\ b) &
\end{array}$$

Writing $[\phi]$ for $\mu X.unit + \phi{\times}X$ (so $[\phi] = unit + \phi{\times}[\phi]$), we can derive (hiding obsolete statements in contexts):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash + : I{\to}I{\to}I} \quad \dfrac{}{\vdash 1 : I}}{\vdash +\,1 : I{\to}I} \quad \dfrac{\dfrac{}{ll{:}[\phi]{\to}I \vdash ll : [\phi]{\to}I} \quad \dfrac{\dfrac{}{x{:}\phi{\times}[\phi] \vdash x : \phi{\times}[\phi]}}{x{:}\phi{\times}[\phi] \vdash \mathsf{right}\ x : [\phi]}}{ll{:}[\phi]{\to}I, x{:}\phi{\times}[\phi] \vdash ll(\mathsf{right}\ x) : I}}{ll{:}[\phi]{\to}I, x{:}\phi{\times}[\phi] \vdash +\,1\ (ll(\mathsf{right}\ x)) : I}}{ll{:}[\phi]{\to}I \vdash \lambda x.+\,1\ (ll(\mathsf{right}\ x)) : (\phi{\times}[\phi]){\to}I}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{list{:}[\phi] \vdash list : [\phi]}}{list{:}[\phi] \vdash list : unit + \phi{\times}[\phi]}\,(\mu) \quad \dfrac{\dfrac{}{x{:}unit \vdash 0 : I}}{\vdash \lambda x.0 : unit{\to}I} \quad \vdots}{ll{:}[\phi]{\to}I, list{:}[\phi] \vdash \mathsf{case}\ (list,\lambda x.0,\lambda x.+\,1\ (ll(\mathsf{right}\ x))) : I}}{ll{:}[\phi]{\to}I \vdash \lambda list.\mathsf{case}\ (list,\lambda x.0,\lambda x.+\,1\ (ll(\mathsf{right}\ x))) : [\phi]{\to}I}}{\vdash \mathsf{Fix}\ ll.\lambda list.\mathsf{case}\ (list,\lambda x.0,\lambda x.+\,1\ (ll(\mathsf{right}\ x))) : [\phi]{\to}I} \quad \dfrac{\dfrac{\dfrac{\dfrac{\nabla}{\vdash a : \phi} \quad \dfrac{\nabla}{\vdash b : [\phi]}}{\vdash \langle a,b\rangle : \phi{\times}[\phi]}}{\vdash \mathsf{inj}{\cdot}r\langle a,b\rangle : unit + \phi{\times}[\phi]}}{\vdash \mathsf{inj}{\cdot}r\langle a,b\rangle : [\phi]}\,(\mu)}{\vdash (\mathsf{Fix}\ ll.\lambda list.\mathsf{case}\ (list,\lambda x.0,\lambda x.+\,1\ (ll(\mathsf{right}\ x))))\ (\mathsf{inj}{\cdot}r\langle a,b\rangle) : I}$$

This approach to recursive types is known as the *equi-recursive* approach [1, 28], because equality modulo infinite unfolding is placed at the heart of the type system. One of its strong points is to not require any explicit type annotations or declarations, so that full type inference is preserved. For this reason, it is exploited, for instance, in the object-oriented subsystem of Objective Caml [47]. Its main disadvantage is that, in the presence of equi-recursive types, apparently meaningless programs have types. For instance, self-application $\lambda x.xx$ has the type $\mu X.X{\to}\phi$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}\mu X.X{\to}\phi \vdash x : \mu X.X{\to}\phi}}{x{:}\mu X.X{\to}\phi \vdash x : (\mu X.X{\to}\phi){\to}\phi}\,(\mu X.X{\to}\phi = (\mu X.X{\to}\phi){\to}\phi) \quad \dfrac{}{x{:}\mu X.X{\to}\phi \vdash x : \mu X.X{\to}\phi}}{x{:}\mu X.X{\to}\phi \vdash xx : \phi}}{\vdash \lambda x.xx : (\mu X.X{\to}\phi){\to}\phi}}{\vdash \lambda x.xx : \mu X.X{\to}\phi}$$

Exercise 9.1 Find a type for $(\lambda x.xx)(\lambda x.xx)$.

9.3 The iso-recursive approach

In the *iso-recursive* approach to recursive types, the above is not possible. In this alternative approach, the equation

$$A \;=\; unit + (B \times A);$$

is viewed as unsatisfiable. Instead, the only way to express lists is to let the user declare a new type constructor *List* (which we write as [ ]), that satisfies

$$[\phi] \;=\; unit + (\phi \times [\phi])$$

Or, more generally, recursive (data) types are defined via:

$$C\,\vec{\varphi} \;=\; A_C[\vec{\varphi}]$$

where C is the user-defined *type constructor*, defined over a number of type variables $\vec{\varphi}$, and $A_C[\vec{\varphi}]$ can refer to C, making the definition recursive, as well as to the type variables. Declarations of iso-recursive types can in fact be *mutually* recursive: every equation can refer to a type constructor introduced by any other equation. Now $C\vec{\varphi}$ and $A_C[\vec{\varphi}]$ are distinct types, but it is possible to convert one into the other via *folding* and *unfolding*. To this effect, the syntax is extended by

$$E \;::=\; \cdots \mid \mathsf{fold}_C(E) \mid \mathsf{unfold}_C(E)$$

as well as the reduction rule

$$\mathsf{unfold}_C(\mathsf{fold}_C(E)) \;\rightarrow\; E$$

and the type assignment rules:

$$(\mathsf{fold}_C):\ \frac{\Gamma \vdash E : A_C[\vec{\varphi}]}{\Gamma \vdash \mathsf{fold}_C\ (E) : C\,\vec{\varphi}}\ (\vec{\varphi} \notin \Gamma)$$

$$(\mathsf{unfold}_C):\ \frac{\Gamma \vdash E : C\,\vec{\varphi}}{\Gamma \vdash \mathsf{unfold}_C\ (E) : A_C[\vec{\varphi}]}\ (\vec{\varphi} \notin \Gamma)$$

We will often omit the type-subscript C. Converting $C\,\vec{\varphi}$ to its unfolding $A_C[\vec{\varphi}]$ – or folding $A_C[\vec{\varphi}]$ to $C\,\vec{\varphi}$ – requires an explicit use of fold or unfold, that is, an explicit syntax in the calculus, making a recursive type-conversion only possible on *call*, i.e. if a fold or unfold call is present in the program. This is contrary to the equi-recursive approach, where the conversion is silent, and not represented in the syntax.

Remember that we deal with recursive types, of the shape $\mu X.C[X]$ so C is some type that contains X; for example, lists of type B are represented by $\mu X.unit + B \times X$. So we can define $[B] = unit + B \times [B]$ as the equation that has $\mu X.unit + B \times X$ as its solution.

The fold rule now expresses: if I have derived that a term E has type $unit + B \times [B]$ (typically by deriving either $unit$ and using (inj_l) or deriving $B \times [B]$ and using (inj_r)), then I can fold this information up, and say that E has type $[B]$ as well. This implies that type $[B]$ gets 'constructed' for E only if either the type $unit$ or the type $B \times [B]$ is derived for E.

For unfold, it works the other way around: if we have derived that E has type $[B]$, then I can unfold that information, as say that E has type $unit + B \times [B]$ (this is typically for used for a variable x, where $x : [B]$ is assumed); I then have access to the types $unit$ and $B \times [B]$, and can do a case analysis.

Notice that it is possible to derive unfold directly after fold, but that would be a waste of effort; however, when showing subject reduction such a derivation can be constructed. We therefore also have the reduction rule $unfold(fold(E)) -> E$.

A term like $\lambda x.xx$ is no longer typeable; instead, the only version of that term typeable now with $\mu X.X\rightarrow\varphi$ is $\mathsf{fold}(\lambda x.(\mathsf{unfold}\ x)\,x)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash x : \mu X.X\rightarrow\varphi}}{\Gamma \vdash \mathsf{unfold}(x) : (\mu X.X\rightarrow\varphi)\rightarrow\varphi} \quad \overline{\Gamma \vdash x : \mu X.X\rightarrow\varphi}}{\Gamma \vdash \mathsf{unfold}(x)x : \varphi}}{\vdash \lambda x.\mathsf{unfold}(x)x : (\mu X.X\rightarrow\varphi)\rightarrow\varphi}}{\vdash \mathsf{fold}(\lambda x.\mathsf{unfold}(x)x) : \mu X.X\rightarrow\varphi}$$

For the list type constructor declared as above, the empty list is written

$\textsf{fold}\ (\textsf{inj}{\cdot}l\ (\ ))$

A list l of type $List\,\varphi$ is deconstructed by

$\textsf{case}\ (\textsf{unfold}\ l,\ \lambda n.\ldots,\ \lambda c.\textsf{let hd} = \textsf{left}\ c\ \texttt{in}\ \textsf{let tl} = \textsf{right}\ c\ \texttt{in}\ \ldots)$

Common use is to fold when constructing data and to unfold when deconstructing it. As can be seen from this example, having explicit (un)folding gives a complicated syntax.

Example 9.4 In this setting, the (silent) μ-conversion in the definition of LL in Example 8.3 are now made explicit, and LL becomes

$$LL\ =\ \textsf{Fix}\ ll.\lambda list.\textsf{case}\ (\textsf{unfold}(list),\lambda x.0,\lambda x.\ +1\ (ll(\textsf{right}\ x)))$$

Notice that now

$$\begin{array}{ll}
(\textsf{Fix}\ ll.\lambda list.\textsf{case}\ (\textsf{unfold}(list),\lambda x.0,\lambda x.\ +1\ (ll(\textsf{right}\ x))))\ (\textsf{fold}(\textsf{inj}{\cdot}r\langle a,b\rangle)) & \to \\
(\lambda list.\textsf{case}\ (\textsf{unfold}(list),\lambda x.0,\lambda x.\ +1\ (LL\ (\textsf{right}\ x))))\ (\textsf{fold}(\textsf{inj}{\cdot}r\langle a,b\rangle)) & \to \\
\textsf{case}\ (\textsf{unfold}(\textsf{fold}(\textsf{inj}{\cdot}r(\langle a,b\rangle))),\lambda x.0,\lambda x.\ +1\ (LL\ (\textsf{right}\ x))) & \to \\
\textsf{case}\ (\textsf{inj}{\cdot}r(\langle a,b\rangle),\lambda x.0,\lambda x.\ +1\ (LL\ (\textsf{right}\ x))) & \to \\
(\lambda x.\ +1\ (LL\ (\textsf{right}\ x)))\ \langle a,b\rangle & \to \\
+1\ (LL\ (\textsf{right}\ \langle a,b\rangle)) & \to \\
+1\ (LL\ b) &
\end{array}$$

Exercise 9.5 Give the derivation for

$$\vdash (\textsf{Fix}\ ll.\lambda list.\textsf{case}\ (\textsf{unfold}(list),\lambda x.0,\lambda x.\ +1\ (ll(\textsf{right}\ x))))\ (\textsf{fold}(\textsf{inj}{\cdot}r\langle a,b\rangle)) : I$$

9.4 Algebraic datatypes

In ML and Haskell, structural products and sums are defined via iso-recursive types, yielding so-called *algebraic data types* [13]. The idea is to avoid requiring both a (type) name and a (field or tag) number, as in $\textsf{fold}\ (\textsf{inj}{\cdot}1\ ())$. Instead, it would be desirable to mention a single name, as in [ ] for the empty list. This is permitted by *algebraic data type* declarations. An algebraic data type constructor C is introduced via a *record* type definition:

$$C\,\vec{\varphi}\ =\ \Pi_{i=1}^{k}\ \ell_i : A_i[\vec{\varphi}]\quad \text{(short for } \ell_1 : A_1[\vec{\varphi}] \times \cdots \times \ell_k : A_k[\vec{\varphi}])$$

or the or *variant* type definition:

$$C\,\vec{\varphi}\ =\ \Sigma_{i=1}^{k}\ \ell_i : A_i[\vec{\varphi}]\quad \text{(short for } \ell_1 : A_1[\vec{\varphi}] + \cdots + \ell_k : A_k[\vec{\varphi}])$$

The record *labels* ℓ_i used in all algebraic data type declarations must be pairwise distinct, so that every record label can be uniquely associated with a type constructor C and with an index i.

The implicit type of the label ℓ_i is $A_i[\vec{\varphi}]{\to}C\,\vec{\varphi}$; we can in fact also allow the label to be parameter less, as in the definition

$$Bool\ =\ True + False$$

The *record* type definition

$$C\,\vec{\varphi}\ =\ \Pi_{i=1}^{k}\ \ell_i : A_i[\vec{\varphi}]$$

introduces the functions ℓ_i for $1 \leq i \leq k$ and build_C, with the following rules:

$$(\mathit{label}): \frac{\Gamma \vdash E : C\,\vec{\phi}}{\Gamma \vdash \ell_i\ (E) : A_i[\vec{\phi}]}\ (\exists 1 \leq i \leq k)$$

$$(\mathsf{build}_C): \frac{\Gamma \vdash E_1 : A_1[\vec{\phi}] \quad \cdots \quad \Gamma \vdash E_k : A_k[\vec{\phi}]}{\Gamma \vdash \mathsf{build}_C\ E_1 \ldots E_k : C\,\vec{\phi}}$$

so the labels act as projection functions into the product type.

In this setting, pairing can be expressed via the product type

$$\langle\rangle\ \phi_1\ \phi_2\ =\ \mathsf{left} : \phi_1 \times \mathsf{right} : \phi_2$$

and the rules

$$\frac{\Gamma \vdash E : \langle\rangle\ \phi_1\ \phi_2}{\Gamma \vdash \mathsf{left}\ (E) : \phi_1} \quad \frac{\Gamma \vdash E : \langle\rangle\ \phi_1\ \phi_2}{\Gamma \vdash \mathsf{right}\ (E) : \phi_2} \quad \frac{\Gamma \vdash E_1 : \phi_1 \quad \Gamma \vdash E_2 : \phi_2}{\Gamma \vdash \mathsf{build}_{\langle\rangle}\ E_1\ E_2 : \langle\rangle\ \phi_1\ \phi_2}$$

Of course an in-fix notation would create better readability: $\Gamma \vdash \langle E_1, E_2\rangle : \langle \phi_1, \phi_2\rangle$.

For readability, we normally write ℓ for $\ell\ ()$ (so when E is empty in $\ell\ E$), so the label needs no arguments.

The *variant* type definition

$$C\,\vec{\phi}\ =\ \Sigma_{i=1}^{k}\ \ell_i : A_i[\vec{\phi}]$$

introduces the functions ℓ_i (with $1 \leq i \leq k$) and case_C, typeable via the rules:

$$(\mathit{label}): \frac{\Gamma \vdash E : A_i[\vec{\phi}]}{\Gamma \vdash \ell_i\ E : C\,\vec{\phi}}\ (\exists 1 \leq i \leq k)$$

$$(\mathsf{case}_C): \frac{\Gamma \vdash E : C\,\vec{\phi} \quad \Gamma \vdash E_1 : A_1[\vec{\phi}] \to C \quad \cdots \quad \Gamma \vdash E_k : A_k[\vec{\phi}] \to C}{\Gamma \vdash \mathsf{case}_C\ (E,\ E_1,\ \ldots,\ E_k) : C}$$

(Notice that the latter is a generalised case of the rule presented above.)

For readability, we write $\mathsf{case}\ E\ [\ \ell_1 : E_1 \cdots \ell_k : E_k\ ]$ for $\mathsf{case}_C(E, E_1, \ldots, E_k)$ when $k > 0$, and $C\,\vec{\phi} = \Sigma_{i=1}^{k}\ \ell_i : A_i[\vec{\phi}]$, thus avoiding to label case.

We can now give the type declaration for lists as

$$[\phi]\ =\ [\,] : \mathit{unit}\ +\ \mathsf{Cons} : \phi \times [\phi]$$

This gives rise to the rules

$$([\,]): \frac{}{\Gamma \vdash [\,] : [\phi]}$$

$$(\mathsf{Cons}): \frac{\Gamma \vdash E : \phi \times [\phi]}{\Gamma \vdash \mathsf{Cons}\ E : [\phi]}$$

$$(\mathsf{case}): \frac{\Gamma \vdash E_1 : [\phi] \quad \Gamma \vdash E_2 : [\,] \to \phi' \quad \Gamma \vdash E_3 : (\phi \times [\phi]) \to \phi'}{\Gamma \vdash \mathsf{case}_{[\phi]}\ (E_1,\ E_2,\ E_3) : \phi'}$$

Notice that here Cons and $[\,]$ act as fold, and the rule (case) as unfold; also, we could have used $\Gamma \vdash E_2 : \phi'$ in the last rule.

In this setting, our example becomes:

$$(\mathsf{Fix}\ ll\,.\,\lambda list.\mathsf{case}_{[\phi]}\ (list,\ \lambda x.0,\ \lambda x.\ +1\ (ll(\mathsf{right}\ x))\ ))\ (\mathsf{Cons}\langle a,b\rangle)$$

or

$$(\mathsf{Fix}\ ll\,.\,\lambda list.\mathsf{case}\ (list,\ Nil : \lambda x.0,\ \mathsf{Cons} : \lambda x.\ +1\ (ll(\mathsf{right}\ x))\ ))\ (\mathsf{Cons}\langle a,b\rangle)$$

This yields concrete syntax that is more pleasant, and more robust, than that obtained when viewing structural products and sums and iso-recursive types as two orthogonal language features. This explains the success of algebraic data types.

Exercise 9.6 Give the derivation for

$$\mathsf{case}\ (list,\ [\,] : \lambda x.0,\ \mathsf{Cons} : \lambda x.\ +1\ (ll(\mathsf{right}\ x))\ )(\mathsf{Cons}\langle a,b\rangle) : I$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{list{:}[\phi] \vdash list : [\phi]}}{list{:}[\phi] \vdash list : unit + \phi\times[\phi]}\,(\mu) \quad \dfrac{\dfrac{}{x{:}unit \vdash 0 : I}}{\vdash \lambda x.0 : unit \to I} \quad \begin{array}{c}\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash + : I\to I\to I} \quad \dfrac{}{\vdash 1 : I}}{\vdash +1 : I\to I} \quad \dfrac{\dfrac{}{ll{:}[\phi]\to I \vdash ll : [\phi]\to I} \quad \dfrac{\dfrac{}{x{:}\phi\times[\phi] \vdash x : \phi\times[\phi]}}{x{:}\phi\times[\phi] \vdash \mathsf{right}\ x : [\phi]}}{ll{:}[\phi]\to I, x{:}\phi\times[\phi] \vdash ll(\mathsf{right}\ x) : I}}{ll{:}[\phi]\to I, x{:}\phi\times[\phi] \vdash +\ 1\ (ll(\mathsf{right}\ x)) : I}}{ll{:}[\phi]\to I \vdash \lambda x.+\ 1\ (ll(\mathsf{right}\ x)) : (\phi\times[\phi])\to I}\\ \vdots\end{array}}{ll{:}[\phi]\to I, list{:}[\phi] \vdash \mathsf{case}\ (list, \lambda x.0, \lambda x.+\ 1\ (ll(\mathsf{right}\ x))) : I}}{ll{:}[\phi]\to I \vdash \lambda list.\mathsf{case}\ (list, \lambda x.0, \lambda x.+\ 1\ (ll(\mathsf{right}\ x))) : [\phi]\to I}}{\vdash \mathsf{Fix}\ ll.\lambda list.\mathsf{case}\ (list, \lambda x.0, \lambda x.+\ 1\ (ll(\mathsf{right}\ x))) : [\phi]\to I} \quad \dfrac{\dfrac{\dfrac{\begin{array}{c}\boxed{\quad}\\ \vdash a : \phi\end{array} \quad \begin{array}{c}\boxed{\quad}\\ \vdash b : [\phi]\end{array}}{\vdash \langle a,b\rangle : \phi\times[\phi]}}{\vdash \mathsf{Cons}\langle a,b\rangle : unit + \phi\times[\phi]}}{\vdash \mathsf{Cons}\langle a,b\rangle : [\phi]}\,(\mu)}{\vdash (\mathsf{Fix}\ ll.\lambda list.\mathsf{case}\ (list, \lambda x.0, \lambda x.+\ 1\ (ll(\mathsf{right}\ x))))\ (\mathsf{Cons}\langle a,b\rangle) : I}$$

10 The intersection type assignment system

In this section we will present a notion of intersection type assignment, and discuss some of its main properties. The system presented here is one out of a family of intersection systems [15, 17, 18, 11, 16, 19, 3, 5], all more or less equivalent; we will use the system of [3] here, because it is the most intuitive.

Intersection types are an extension of Curry types by adding an extra type constructor '$\cap$', that enriches the notion of type assignment in a dramatic way. In fact, type assignment is now closed for $=_\beta$, which immediately implies that it is undecidable.

We can recover from the undecidability by limiting the structure of types, an approach that is used in [6, 33], the trivial being to do without intersection types at all, and fall back to Curry types.

10.1 Intersection types

Intersection types are defined by extending Curry with the type contructor '$\cap$'.

Definition 10.1 (Strict types) *i*) $\mathcal{T}_s$, the set of *strict types*, is defined by:

$$A \ ::= \ \varphi \mid ((A_1 \cap \cdots \cap A_n) \to A),\ (n \geq 0)$$

The set $\mathcal{T}$ of strict intersection types is defined by:

$$\mathcal{T} \ = \ \{(A_1 \cap \cdots \cap A_n) \mid n \geq 0\ \&\ \forall 1 \leq i \leq n\ [A_i \in \mathcal{T}_s]\}$$

ii) On $\mathcal{T}$, the relation $\leq$ is defined as the smallest relation satisfying:

$$\begin{array}{rcll} \forall 1 \leq i \leq n \; [A_1 \cap \cdots \cap A_n & \leq & A_i] & (n \geq 1) \\ \forall 1 \leq i \leq n \; [A \leq A_i] & \Rightarrow & A \leq A_1 \cap \cdots \cap A_n & (n \geq 0) \\ A \leq B \leq C & \Rightarrow & A \leq C & \end{array}$$

iii) We define the relation $\sim$ by:

$$\begin{array}{rcl} A \leq B \leq A & \Rightarrow & A \sim B \\ A \sim C \;\&\; B \sim D & \Rightarrow & A \to B \sim C \to D \end{array}$$

We will work with types modulo $\sim$.

As usual in the notation of types, right-most, outermost brackets will be omitted, and, as in logic, '$\cap$' binds stronger than '$\to$', so $C \cap D \to C \to D$ stands for $((C \cap D) \to (C \to D))$.

We will write $\cap_n A_i$ for $A_1 \cap \cdots \cap A_n$, and use $\top$ to represent an intersection over zero elements: if $n = 0$, then $\cap_n A_i = \top$, so, in particular, $\top$ does not occur in an intersection subtype. Moreover, intersection type schemes (so also $\top$) occur in strict types only as subtypes at the left hand side of an arrow type scheme.

Exercise 10.2 ([5]) *For all $A, B \in \mathcal{T}$, $A \leq B$ if and only if there are A_i $(i \in \underline{n}), B_j$ $(j \in \underline{m})$ such that $A = \cap_n A_i$, $B = \cap_m B_j$, and $\{B_j \; (j \in \underline{m})\} \subseteq \{A_i \; (i \in \underline{n})\}$.* ■

Notice that, by definition, in $\cap_n A_i$, all A_i are strict; sometimes we will deviate from this by writing $A \cap B$ also for A and B not in $\mathcal{T}_s$.

Definition 10.3 (Contexts) *i*) A *statement* is an expression of the form $M : A$, where M is the *subject* and A is the *predicate* of $M : A$.

ii) A *context* Γ is a set of statements with (distinct) variables as subjects.

iii) The relations $\leq$ and $\sim$ are extended to contexts by:

$$\begin{array}{l} \Gamma \leq \Gamma' \iff \forall x{:}A' \in \Gamma' \; \exists x{:}A \in \Gamma \; [A \leq A'] \\ \Gamma \sim \Gamma' \iff \Gamma \leq \Gamma' \leq \Gamma. \end{array}$$

10.2 Intersection type assignment

This type assignment system will derive judgements of the form $\Gamma \vdash_\cap M : A$, where Γ is a context and A a type.

Definition 10.4 *i*) *Strict type assignment* and *strict derivations* are defined by the following natural deduction system (where all types displayed are strict, except A in the derivation rules $(\to I)$ and $(\to E)$):

$$(Ax): \frac{}{\Gamma, x{:}\cap_n A_i \vdash_\cap x : A_i} \; (n \geq 1)$$

$$(\cap I): \frac{\Gamma \vdash_\cap M : A_1 \quad \cdots \quad \Gamma \vdash_\cap M : A_n}{\Gamma \vdash_\cap M : \cap_n A_i} \; (n \geq 0)$$

$$(\to I): \frac{\Gamma, x{:}A \vdash_\cap M : B}{\Gamma \vdash_\cap \lambda x . M : A \to B}$$

$$(\to E): \frac{\Gamma \vdash_\cap M : A \to B \quad \Gamma \vdash_\cap N : A}{\Gamma \vdash_\cap MN : B}$$

We write $\Gamma \vdash_\cap M : A$, if this is derivable from Γ using a strict derivation.

Notice that in $\Gamma \vdash_\cap M : A$ the context can contain types that are not strict, and that $\Gamma \vdash_\cap M : A$ is only defined for $A \in \mathcal{T}$.

For this notion of type assignment, the following properties hold:

Exercise 10.5 *i)* $\Gamma \vdash_\cap MN : A \iff \exists B\, [\Gamma \vdash_\cap M : B \to A \;\&\; \Gamma \vdash_\cap N : B]$.
ii) $\Gamma \vdash_\cap \lambda x.M : A \iff \exists C \in \mathcal{T}, D \in \mathcal{T}_s\, [A = C \to D \;\&\; \Gamma \backslash x, x{:}C \vdash_\cap M : D]$.
iii) $\Gamma \vdash_\cap M : A \iff x{:}B \in \Gamma \mid x \in fv(M) \vdash_\cap M : A$.

Example 10.6 In this system, we can derive both $\emptyset \vdash_\cap (\lambda xyz.xz(yz))(\lambda ab.a) : \top \to A \to A$ and $\emptyset \vdash_\cap \lambda yz.z : \top \to A \to A$ (where $\Gamma = x{:}A \to \top \to A, z{:}A$).

$$\dfrac{\dfrac{\overline{z{:}A \vdash_\cap z : A}}{\emptyset \vdash_\cap \lambda z.z : A \to A}}{\emptyset \vdash_\cap \lambda yz.z : \top \to A \to A}$$

and

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash_\cap x : A \to \top \to A} \quad \overline{\Gamma \vdash_\cap z : A}}{\Gamma \vdash_\cap xz : \top \to A} \qquad \overline{\Gamma \vdash_\cap yz : \top}}{\Gamma \vdash_\cap xz(yz) : A}}{x{:}A \to \top \to A \vdash_\cap \lambda z.xz(yz) : A \to A}}{x{:}A \to \top \to A \vdash_\cap \lambda yz.xz(yz) : \top \to A \to A}}{\emptyset \vdash_\cap \lambda xyz.xz(yz) : (A \to \top \to A) \to \top \to A \to A} \quad \dfrac{\dfrac{\overline{a{:}A \vdash_\cap a : A}}{a{:}A \vdash_\cap \lambda b.a : \top \to A}}{\emptyset \vdash_\cap \lambda ab.a : A \to \top \to A}}{\emptyset \vdash_\cap (\lambda xyz.xz(yz))(\lambda ab.a) : \top \to A \to A}$$

Notice that, by using $\Gamma = x{:}A \to \top \to A, y{:}B, z{:}A$ in the derivation above, we could as well have derived $\emptyset \vdash_\cap (\lambda xyz.xz(yz))(\lambda ab.a) : B \to A \to A$.

10.3 Subject reduction and normalisation

That subject reduction holds in this system is not difficult to see. The proof follows very much the same lines as the one given in Theorem 2.6, and will follow below; first we give an intuitive argument. Suppose there exists a type assignment for the redex $(\lambda x.M)N$, so there are a context Γ and a type A such that there is a derivation for $\Gamma \vdash_\cap (\lambda x.M)N : A$. Then by $(\to E)$ there is a type $\cap_n B_i$ such that there are derivations $\Gamma \vdash_\cap \lambda x.M : \cap_n B_i \to A$ and $\Gamma \vdash_\cap N : \cap_n B_i$. Since $(\to I)$ should be the last step performed in the derivation for $\Gamma \vdash_\cap \lambda x.M : \cap_n B_i \to A$, there is also a derivation for $\Gamma, x{:}\cap_n B_i \vdash_\cap M : A$. Since $(\cap I)$ must have been the last step performed in the derivation for $\Gamma \vdash_\cap N : \cap_n B_i$, for every $1 \leq i \leq n$, there exists a derivation for $\Gamma \vdash_\cap N : B_i$. Then a derivation for $\Gamma \vdash_\cap M[N/x] : A$ can be obtained by replacing for $1 \leq i \leq n$ the sub-derivation $\Gamma, x{:}\cap_n B_i \vdash_\cap x : B_i$ by the derivation for $N{:}B_i$.

The problem to solve in a proof for closure under β-equality is then that of β-expansion: suppose we have derived $\Gamma \vdash_\cap M[N/x] : A$ and also want to derive $\Gamma \vdash_\cap (\lambda x.M)N : A$. If the term-variable x occurs in M, then the term N is a subterm of $M[N/x]$, so N is typed in the derivation for $\Gamma \vdash_\cap M[N/x] : A$. It may be that in this derivation the subterm N is typed with several different types, say A_i $(i \in \underline{n})$, so, for $1 \leq i \leq n$, $\Gamma \vdash_\cap N : A_i$. Then in Curry's system M can not be typed using the same types, since then the context would contain more than one type for x, which is not allowed. In the intersection system a term-variable can have different types within a derivation, combined in an intersection, and the term M can then be typed by $\Gamma, x{:}\cap_n A_i \vdash_\cap M : A$, and from this we get, by rule $(\to I)$, $\Gamma \vdash_\cap \lambda x.M : \cap_n A_i \to A$. Since, for every $1 \leq i \leq n$, $\Gamma \vdash_\cap N : A_i$, by rule $(\cap I)$ we also have $\Gamma \vdash_\cap N : \cap_n A_i$. Then, using $(\rightarrow E)$, the redex can be typed.

The next exercise states that type assignment is closed for '$\leq$'.

Exercise 10.7 If $\Gamma \vdash_\cap M : A$ and $A \leq B$, and $\Gamma' \leq \Gamma$, then $\Gamma' \vdash_\cap M : B$.

We will show now show formally that '$\vdash_\cap$' is closed for '$=_\beta$'. First, a substitution lemma is proved. Notice that, unlike for Curry's system, the implication holds in both directions.

Lemma 10.8 $\exists C\ [\Gamma, x{:}C \vdash_\cap M : A \ \&\ \Gamma \vdash_\cap N : C] \iff \Gamma \vdash_\cap M[N/x] : A$.

Proof: By induction on M. Only the case $A \in \mathcal{T}_s$ is considered.

$$\begin{array}{llll}
(M \equiv x): & (\Rightarrow): & \exists C\ [\Gamma, x{:}C \vdash_\cap x : A \ \&\ \Gamma \vdash_\cap N : C] & \Rightarrow\ (Ax) \\
& & \exists A_i\ (i \in \underline{n}), j \in \underline{n}\ [A = A_j \ \&\ \Gamma \vdash_\cap N : \cap_n A_i] & \Rightarrow\ (9.7) \\
& & \Gamma \vdash_\cap x[N/x] : A_j. &
\end{array}$$

$(\Leftarrow)$: $\Gamma \vdash_\cap x[N/x] : A \Rightarrow \Gamma, x{:}A \vdash_\cap x : A \ \&\ \Gamma \vdash_\cap N : A$.

$(M \equiv y \neq x)$: $(\Rightarrow)$: $\exists C\ [\Gamma, x{:}C \vdash_\cap y : A \ \&\ \Gamma \vdash_\cap N : C] \Rightarrow \Gamma \vdash_\cap y[N/x] : A$.

$(\Leftarrow)$: $\Gamma \vdash_\cap y[N/x] : A \Rightarrow \Gamma \vdash_\cap y : A \ \&\ \Gamma \vdash_\cap N : \top$.

$$\begin{array}{lll}
(M \equiv \lambda y.M'): \quad (\iff): & \exists C\ [\Gamma, x{:}C \vdash_\cap \lambda y.M' : A \ \&\ \Gamma \vdash_\cap N : C] & \iff (\rightarrow I) \\
& \exists C, A', B'\ [\Gamma, x{:}C, y{:}A' \vdash_\cap M' : B' \ \&\ A = A' \rightarrow B' \ \&\ \Gamma \vdash_\cap N : C] & \iff (IH) \\
& \exists A', B'\ [\Gamma, y{:}A' \vdash_\cap M'[N/x] : B' \ \&\ A = A' \rightarrow B'] & \iff (\rightarrow I) \\
& \Gamma \vdash_\cap \lambda y.M'[N/x] : A. &
\end{array}$$

$$\begin{array}{ll}
(M \equiv M_1 M_2): \quad (\iff): \quad \Gamma \vdash_\cap M_1 M_2[N/x] : A & \iff (\rightarrow E) \\
\exists B\ [\Gamma \vdash_\cap M_1[N/x] : B \rightarrow A \ \&\ \Gamma \vdash_\cap M_2[N/x] : B] & \iff (IH) \\
\exists C_1, C_2, B\ [\Gamma, x{:}C_i \vdash_\cap M_1 : B \rightarrow A \ \&\ \Gamma \vdash_\cap N : C_1 \ \&\ \Gamma, x{:}C_2 \vdash_\cap M_2 : B \ \&\ \Gamma \vdash_\cap N : C_2] & \\
& \iff (C = C_1 \cap C_2) \ \&\ (\cap I) \ \&\ (9.7) \\
\exists C\ [\Gamma, x{:}C \vdash_\cap M_1 M_2 : A \ \&\ \Gamma \vdash_\cap N : C]. & \blacksquare
\end{array}$$

Corollary 10.9 $M =_\beta N \Rightarrow (\Gamma \vdash_\cap M : A \iff \Gamma \vdash_\cap N : A)$, so the following rule is admissible in '$\vdash_\cap$':

$$(=_\beta): \frac{\Gamma \vdash_\cap M : A}{\Gamma \vdash_\cap N : A}\ (M =_\beta N)$$

Proof: By induction on the definition of '$=_\beta$'. The only part that needs attention is that of a redex, $\Gamma \vdash_\cap (\lambda x.M)N : A \iff \Gamma \vdash_\cap M[N/x] : A$, where $A \in \mathcal{T}_s$; all other cases follow by straightforward induction. To conclude, notice that, if $\Gamma \vdash_\cap (\lambda x.M)N : A$, then, by $(\rightarrow E)$ and $(\rightarrow I)$, there exists C such that $\Gamma, x{:}C \vdash_\cap M : A$ and $\Gamma \vdash_\cap N : C$. The result follows then from Lemma 9.8. $\blacksquare$

In fact, interpreting a term M by its set of assignable types $\mathcal{T}(M) = \{A \mid \exists \Gamma\ [\Gamma \vdash_\cap M : A]\}$ gives a *semantics* for M, and a *filter model* for the Lambda Calculus (for details, see [11, 3, 5]).

Types are not invariant by η-reduction. For example, notice that $\lambda xy.xy \rightarrow_\eta \lambda x.x$; we can derive $\emptyset \vdash_\cap \lambda xy.xy : (\varphi_1 \rightarrow \varphi_2) \rightarrow \varphi_1 \cap \varphi_3 \rightarrow \varphi_2$, but not $\emptyset \vdash_\cap \lambda x.x : (\varphi_1 \rightarrow \varphi_2) \rightarrow \varphi_1 \cap \varphi_3 \rightarrow \varphi_2$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \varphi_1 \rightarrow \varphi_2, y : \varphi_1 \cap \varphi_3 \vdash_\cap x : \varphi_1 \rightarrow \varphi_2} \qquad \dfrac{}{x : \varphi_1 \rightarrow \varphi_2, y : \varphi_1 \cap \varphi_3 \vdash_\cap y : \varphi_1}\ (\varphi_1 \cap \varphi_3 \leq \varphi_1)}{x : \varphi_1 \rightarrow \varphi_2, y : \varphi_1 \cap \varphi_3 \vdash_\cap xy : \varphi_2}}{x : \varphi_1 \rightarrow \varphi_2 \vdash_\cap \lambda y.xy : \varphi_1 \cap \varphi_3 \rightarrow \varphi_2}}{\emptyset \vdash_\cap \lambda xy.xy : (\varphi_1 \rightarrow \varphi_2) \rightarrow \varphi_1 \cap \varphi_3 \rightarrow \varphi_2}$$

We cannot derive $\emptyset \vdash_\cap \lambda x.x : (\varphi_1 \rightarrow \varphi_2) \rightarrow \varphi_1 \cap \varphi_3 \rightarrow \varphi_2$, since we cannot transform the type

$\varphi_1 \rightarrow \varphi_2$ into $\varphi_1 \cap \varphi_3 \rightarrow \varphi_2$ using $\leq$. There exists intersection systems that do allow this (see, for example, [3]).

The intersection type assignment system allows for a very nice characterisation, through assignable types, of normalisation, head-normalisation, and strong normalisation [11, 5].

Theorem 10.10 *i) There are $\top$-free Γ, A such that $\Gamma \vdash_\cap M : A$ if and only if M has a normal form.*
ii) There are $\Gamma, A \in \mathcal{T}_s$ such that $\Gamma \vdash_\cap M : A$ if and only if M has a head normal form.
iii) If M is strongly normalisable, if and only if there are Γ and A such that $\Gamma \vdash_{\not\omega} M : A$.

Because all these properties can be reduced to the *halting problem*, type assignment with intersection types is undecidable. It is possible to define a notion of principal pair for lambda terms using intersection types [50, 4, 5]. A semi-algorithm is defined in [49]; if a term has a principal pair is, of course, undecidable.

11 A brief overview of the Haskell type system

(This section is in part inspired by, based on, and copied from [31] and [52]; we will assume the reader to be familiar with Haskell as a programming language).

Haskell is a functional programming language where types are omnipresent, and play an important role for program correctness and code generation. Because Haskell is purely functional, all computations are done via the evaluation of *expressions* to yield *values*; every value has an associated type. Examples of expressions include atomic values such as the integer `5`, the character `'a'`, and the function `\x -> x+1` (Haskell's notation for $\lambda x.x + 1$), as well as structured values such as the list `[1,2,3]` and the pair `('b',4)`.

Just as expressions denote values, type expressions are syntactic terms that denote type values (or just types). Examples of type expressions are the atomic types `Integer` (infinite-precision integers), `Char` (characters), the function type `Integer -> Integer` (functions mapping Integer to Integer), as well as the structured types `[Integer]` (homogeneous lists of integers) and `(Char,Integer)` (pairs of character and integer). All Haskell values are "first class" in that they may be passed as arguments to functions, returned as results, placed in data structures, etc.

Typings are written as *Value* : : *Type*, as in

```
5              :: Integer
`a`            :: Char
\x -> x+1      :: Integer -> Integer
[1,2,3]        :: [Integer]
('b',4)        :: (Char,Integer)
```

The type system also ensures that user-supplied type signatures are correct. In fact, Haskell's type system is powerful enough to avoid writing any type signatures at all; the type system incorporated into the compiler infers the correct types; however, in type inference it will use Milner's approach to type recursive definitions, and Mycroft's when a type is supplied.

Haskell's type system uses the shallow polymorphism we have seen before for ML, but since programs are written in plain ascii, type variables are written using normal font, as in `a`: also, since the quantifier cannot be represented, $\forall$`a.Type` is written as `Type` (i.e. all variables are implicitly universally quantified (in a shallow fashion).

Functions are in Haskell defined as term rewriting systems, using pattern matching; rewrite rules are matched top-down, and definitions can be partial, in that the alternatives need not cover all possibilities: this gives rise to run-time errors.

```
head          :: [a] -> a
head (x:xs)   = x

tail          :: [a] -> [a]
tail (x:xs)   = xs
```

The kind of patterns used allow for matching only against *data constructors*, which constitutes a *function-constructor* system. We have seen above that pattern matching creates problems for subject reduction, so it is not evident that this property holds for Haskell. However, this property in shown to hold for the polymorphic rewriting calculus [39], a variant of the λ-calculus that adds pattern matching [45], but restricts those to a function-constructor system, so is likely to hold also for Haskell. Also the principal type property is claimed in [31].

11.1 Data types

Haskell allows for user-defined types using the `data` tag; for example, the boolean type can be defined via

```
data Bool     = False | True
```

which corresponds to the variant type definition

$$Bool \;=\; True + False$$

that we saw above. The word `data` is a tag for the compiler to make sure the definition concerns a type. Another example:

```
data Point a  = Pt a a
```

Here `Point` is a unary *type constructor*, and `Pt` a binary *data constructor*, that constructs an object of type `Point`; likewise, `Bool` above is nullary type constructor, and `True` and `False` are nullary data constructors that construct an object of type `Bool`; in fact, we can even define

```
data Point a  =  Point a a
```

so then `Point` is both a type as a data constructor (this does not create problems). It is possible to label the fields:

```
data Point a  = Point {pointx, pointy :: a}
```

or

```
data FloatPoint       = Point {pointx, pointy :: Float}
```

The normal (lazy) reduction strategy can be influenced by annotating the type definition, as in

```
data RealFloat        => Complex a = + !a !a
```

This definition marks the two components, the real and imaginary parts, of the complex number as being strict, i.e., they will be reduced to values before being placed in the structure. This is a more compact representation of complex numbers, and comes at the expense of making a complex number with an undefined component, `+ 1` $\perp$ for example, totally undefined ($\perp$). As there is no real need for partially defined complex numbers, it makes sense to use strictness flags to achieve a more efficient representation.

Types can also be recursive, as in the type of binary trees:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

This defines a polymorphic binary tree type whose elements are either leaf nodes containing a value of type `a`, or internal nodes ("branches") containing (recursively) two sub-trees.

Notice that this type declaration corresponds to the type definition

$$Tree\ \phi \ = \ Leaf\ \phi + Branch\ (Tree\ \phi)\ (Tree\ \phi)$$

As mentioned in the previous section, we now have the following types for `Branch` and `Leaf`:

$$\begin{aligned} \texttt{Branch} &: \forall\phi.(\texttt{Tree}\ \phi) \to (\texttt{Tree}\ \phi) \to (\texttt{Tree}\ \phi) \\ \texttt{Leaf} &: \forall\phi.\phi \to (\texttt{Tree}\ \phi) \end{aligned}$$

Type constructors and data constructors need not be pre-fix:

```
data [a] = [] | a : [a]
```

Here the type constructor is `[]`, and `:` is a data constructor[6]. The types of the data constructors are as expected:

$$\begin{aligned} \texttt{[]} &: \forall\phi.\texttt{[}\phi\texttt{]} \\ \texttt{:} &: \forall\phi.\phi \to \texttt{[}\phi\texttt{]} \to \texttt{[}\phi\texttt{]} \end{aligned}$$

For convenience, Haskell provides a way to define type synonyms, i.e. names for commonly used types. Type synonyms are created using a type declaration.

```
type String   = [Char]
type Person   = (Name, Address)
type Name     = String
data Address  = None | Addr String
```

Notice that `None` has type `Address`.

Although formally defined and treated as such, not all data types used in Haskell are defined like this. For example, we would like to define

```
data Int      = -65532 | ... | -1 | 0 | 1 | ... | 65532
data Integer  = ... -2 | -1 | 0 | 1 | 2 ...
```

but this is in practice not feasible. Notice that the first is a finite declaration (albeit large, which would involve a large look-up table to be used at compile time), but that the second is infinite; there is no way we can incorporate this. The solution to this conundrum is to use *implicit* type assignment, by splitting the syntax of Haskell into syntactic categories: then the compiler - or rather, the parser - will detect if a sequence of characters consists entirely of numbers, and thus assign it type `Integer`, or is a single character between quotes which is then assigned the type `Char`, etc.

It is also possible to define a type whose representation is identical to an existing one but which has a separate identity in the type system. In Haskell, the newtype declaration creates a new type from an existing one. For example, natural numbers can be represented by the type Integer using the following declaration:

```
newtype Natural = MakeNatural Integer
```

This creates an entirely new type, `Natural`, whose only constructor contains a single `Integer`, and the constructor `MakeNatural` converts between a `Natural` and an `Integer`.

[6]In Haskell, when using an infix constructor in a prefix position - as `:` here, used with zero arguments - backquotes have to be placed, as in `` `:` ``; since we are not writing a program here, but talk about the various constructors, we need not bother with these details; we let the font of the symbols do the talking.

11.2 Errors

To reason correctly on the running of programs, the value representing a non-terminating program, as well as a run-time error[7], is $\perp$, pronounced *bottom*. Since this value can be produced by *any* program, or better, in any result type, $\perp$ is assumed to have all types: $\perp : A$, for al types A, which we can generalise to $\perp : \forall\phi.\phi$. This implies that $\forall\phi.\phi$ is the type for the meaningless computation, which corresponds to what we have seen before: notice that

$$Y(\lambda x.x) \;=\; (\lambda m.m(Ym))(\lambda x.x) \;\to_\beta\; (\lambda x.x)(Y(\lambda x.x)) \;\to\; Y(\lambda x.x)$$

which clearly does not terminate - it does not even have a head-normal form. We can type $Y(\lambda x.x)$ (which corresponds to $\mathsf{Fix}\ x.x$) as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash Y : \forall\phi.(\phi\to\phi)\to\phi}}{\vdash Y : (\phi\to\phi)\to\phi}\,(\forall E) \quad \dfrac{\dfrac{}{x{:}\phi \vdash x : \phi}}{\vdash \lambda x.x : \phi\to\phi}\,(\to I)}{\vdash Y(\lambda x.x) : \phi}\,(\to E)}{\vdash Y(\lambda x.x) : \forall\phi.\phi}\,(\forall I)$$

which supports the idea that $\forall\phi.\phi$ is the non-informative type, i.e. is the natural type for the program that contains no information, i.e. $\perp$.

Haskell has a built-in function called `error` whose type is `String->a` (or $\forall\phi.$`String`$\to\phi$ in our notation), i.e. it is a polymorphic function, capable of returning any type at all. Since $\perp$ is the one value "shared" by all types, semantically this is the value always returned by `error`. However, we can expect that a reasonable implementation will print the string argument to error for diagnostic purposes, as a side effect to the calculation. Thus this function is useful when we wish to terminate a program when something has "gone wrong." For example, the actual definition of `head` is:

```
head (x:xs)    = x
head []        = error "head{PreludeList}: head []"
```

11.3 Classes

Overloading in Haskell is modelled via the concept of *classes*. For example, take the definition of the function `elem` which tests for membership in a list:

```
elem x []     = False
elem x (y:ys) = x==y || (elem x ys)
```

which 'should' have type `a -> [a] -> Bool`, but this is not the case: `==` does not have type `a -> a -> Bool`, since we cannot define equality in every type (for example, we cannot decide if two functions are the same, or, in general, if two programs (λ-terms) are). So equality should only be defined for certain types (`Integer`, `Char`, for example), and only for those types, and for all of them, will `elem` be well defined.

This is solved by introducing classes. For example, a type class containing an equality operator is defined by:

```
class Eq a where
  (==)         :: a -> a -> Bool
```

Here `Eq` is the name of the class being defined, and `==` is the single operation in the class. This declaration may be read "a type is an instance of the class `Eq` if there is an (overloaded)

[7]There exists notions of type assignment that separate non-termination from run-time errors.

operation ==, of the appropriate type, defined on it." That a type a must be an instance of the class Eq is written Eq a; this is not a type expression, but a constraint on a type, and is called a context. Contexts are placed at the front of type expressions. For example, the effect of the above class declaration is to assign the following type to ==:

```
(==)  :: (Eq a) => a -> a -> Bool
```

This should be read, "For every type a that is an instance of the class Eq, == has type a -> a -> Bool." This is the type that would be used for == in the elem example, and indeed the constraint imposed by the context propagates to the principal type for elem:

```
elem  :: (Eq a) => a -> [a] -> Bool
```

This is read "every type a that is an instance of the class Eq, elem has type a->[a]->Bool." This expresses the fact that elem is not defined on all types, just those for which we know how to compare elements for equality.

To specify which types are instances of the class Eq, and the actual behaviour of == on each of those types is done via an *instance declaration*. For example:

```
instance Eq Integer where
  x == y       = x `integerEq` y
```

The definition of == is called a *method*. Here the function integerEq is the primitive function that compares integers for equality. Given this declaration, we can now compare fixed precision integers for equality using ==. Similarly:

```
instance Eq Float where
  x == y       = x `floatEq` y
```

allows us to compare floating point numbers using ==. In general any valid expression is allowed on the right-hand side, just as for any other function definition, giving potentially completely different implementations of equality; it is of course the responsibility of the programmer to make sure that the code is correct.

The Haskell Prelude contains many type classes. In fact, a class Eq is defined that contains more methods than the one above:

```
class Eq a where
  (==), (/=)  :: a -> a -> Bool
  x /= y      = not (x == y)
```

Notice that now only the code of the method == needs to be specified for any instance of this class.

Haskell also supports a notion of class extension. For example, the class Ord inherits all of the operations in Eq, but in addition has a set of comparison operations and minimum and maximum functions:

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>)        :: a -> a -> Bool
  max, min                    :: a -> a -> a
```

Eq is a superclass of Ord, and Ord is a subclass of Eq, and any type which is an instance of Ord is also an instance of Eq.

11.4 Monads

A *monad* is a rather difficult notion that comes from category theory, but has a nice application in the context of referentially transparent languages [41, 42, 52]: they can be used to deal without the normal excessive overhead with error handling, count of operations, execution traces, etc. Wadler [52] gives a detailed list of examples that illustrate the usefulness of monads: we will focus on his example of exception handling here.

Take an evaluator that acts on terms and calculates a composed division:

```
data Term     = Con Integer | Div Term Term
```

The basic evaluator is defined by:

```
eval                 :: Term -> Integer
eval (Con a)         = a
eval (Div t u)       = (eval t) / (eval u)
```

where / is appropriately defined as division over `Integer`, which is clearly a partial function; since `eval` does not incorporate error handling, the result of `eval error` is undefined.

In a pure language, exception handling may be mimicked by introducing a type to represent computations that may raise an exception.

```
data M a             = Raise Exception | Return a
type Exception       = String
```

A value of type `M a` either has the form `Raise e`, where `e` is an exception, or `Return a`, where `a` is a value of type `a`[8].

It is straightforward, but cumbersome, to adapt the evaluator to this representation.

```
eval                 :: Term -> M Integer
eval (Con a)         = Return a
eval (Div t u)       = case eval t of
                           Raise e       -> Raise e
                           Return a      ->
                             case eval u of
                               Raise e   -> Raise e
                               Return b  ->
                                 if b == 0
                                    then Raise "divide by zero"
                                    else Return (a / b)
```

At each call of the evaluator, the form of the result must be checked: if an exception was raised it is re-raised, and if a value was returned it is processed.

What sort of operations are required on the type `M`?[9] First, we need a way to turn a value into the computation that returns that value and does nothing else.

```
return :: a -> M a
```

Second, we need a way to apply a function of type `a -> M b` to a computation of type `M a`.

```
(>>=) :: M a -> (a -> M b) -> M b
```

The Monad class defines two basic operators: >>= (bind) and `return`.

```
infixl 1 >>, >>=
class Monad m where
      (>>=)   :: m a -> (a -> m b) -> m b
      (>>)    :: m a -> m b -> m b
```

[8]By (Haskell) convention, `a` will be used both as a type variable, as in `M a`, and as a variable ranging over values of that type, as in `Return a`.

[9]We will often write expressions in the form `m >>= \a.n`, which can be seen as `let a = m in n`, where `m` and `n` are expressions, and `a` is a variable.

```
return  :: a -> m a
fail    :: String -> m a

m >> k  = m >>= \_ -> k
```

The bind operations, `>>` and `>>=`, combine two monadic values while the `return` operation injects a value into the monad (container). The signature of `>>=` helps us to understand this operation: `ma >>= \_ -> mb` combines a monadic value `ma` containing values of type `a` and a function which operates on a value `v` of type `a`, returning the monadic value `mb`. The result is to combine `ma` and `mb` into a monadic value containing `b`. The `>>` function is used when the function does not need the value produced by the first monadic operator. The precise meaning of binding depends, of course, on the monad.

The `do` syntax provides a simple shorthand for chains of monadic operations. The essential translation of `do` is captured in the following two rules:

```
do e1 ; e2          = e1 >> e2
do p <- e1; e2      = e1 >>= \p -> e2
```

When the pattern in this second form of `do` is refutable, pattern match failure calls the `fail` operation; this may raise an error. Thus the more complete translation is

```
do p <- e1; e2      = e1 >>= (\v ->
                              case v of
                                p -> e2
                                _ -> fail "s")
```

where `"s"` is a string identifying the location of the `do` statement for possible use in an error message.

These operations must satisfy three laws:

(*Left return*) : Compute the value `a`, bind `b` to the result, and compute `n`. The result is the same as `n` with value `a` substituted for variable `b`.

```
return a >>= \b -> n = n[a/b]
```

(*Right return*) : Compute `m`, bind the result to `a`, and return `a`. The result is the same as `m`.

```
m >>= \a -> return a = m
```

(*Associative*) : Compute `m`, bind the result to `a`, compute `n`, bind the result to `b`, compute `o`. The order of parentheses in such a computation is irrelevant.

```
m >>= \a -> (n >>= \b -> o) = (m >>= \a -> n) >>= \b -> o
```

The scope of the variable `a` includes `o` on the left but excludes `o` on the right, so this law is valid only when `a` does not appear free in `o`.

Or, formulated differently:

```
return a >>= k              = k a
m >>= return                = m
xs >>= return . f           = fmap f xs
m >>= (\x -> k x >>= h)     = (m >>= k) >>= h
```

Haskell's monads allow for another solution for the exception handling of `eval`. The original evaluator has the type `Term -> Integer`, while above its type took the form `Term -> M Integer`. In general, a function of type `a -> b` is replaced by a function of type `a -> M b`. This can be read as a function that accepts an argument of type `a` and returns a result of type `b`, with a possible additional effect captured by `M`. This effect may be to act on state, generate output, raise an exception, etc.

A *monad* is a triple `(M, return, >>=)` consisting of a type constructor `M` and two operations of the given polymorphic types.

Using monads, the evaluator can be rewritten as:

```
eval                  :: Term -> M Integer
eval (Con a)          = return a
eval (Div t u)        = (eval t) >>= \a  ->  (eval u) >>= \b ->  return (a / b)
```

The type `Term -> M Integer` indicates that the evaluator takes a term and performs a computation yielding an `Integer`. To compute `Con a`, just return `a`. To compute `Div t u`, first compute `t`, bind `a` to the result, then compute `u`, bind `b` to the result, and then return `a / b`.

In the exception monad, a computation may either raise an exception or return a value.

```
data M a              = Raise Exception | Return a
type Exception        = String

return                :: a -> M a
return a              = Return a

(>>=)                 :: M a -> (a -> M b) -> M b
m >>= k               = case m of
                          Raise e   -> Raise e
                          Return a  -> k a

raise                 :: Exception    -> M a
raise e               = Raise e
```

The call `return` a simply returns the value `a`. The call `m >>= k` examines the result of the computation `m`: if it is an exception it is re-raised, otherwise the function `k` is applied to the value returned. Finally, the call raise `e` raises the exception `e`. To add error handling to the monadic evaluator, take the monad as above. Then just replace `return (a / b)` by

```
if b == 0
   then raise divide by zero
   else return (a / b)
```

This is similar to the change that would be required in an impure language.

11.5 I/O

The I/O system in Haskell is purely functional, yet has all of the expressive power found in imperative programming languages. Haskell's I/O system is built around a monad. Every I/O action returns a value. In the type system, the return value is 'tagged' with IO type, distinguishing actions from other values. For example, the type of the function `getChar` and `putChar` are:

```
getChar       :: IO Char
putChar       :: Char -> IO ()
```

The `IO` constructor works as a *cast*, forcing the type `Char` to one that becomes a monad, enabling the additional functionality.

Actions are sequenced using `do`, which we have seen can be encoded via `>>=` (or '`bind`'). Here is a simple program to read and then print a character:

```
main  :: IO ()
main  = do c <- getChar
```

```
             putChar c
```

For example, in the I/O monad, an action such as `'a' <- getChar` will call `fail` if the character typed is not `'a'`. This, in turn, terminates the program since in the I/O monad fail calls error.

The function getLine:

```
getLine       :: IO String
getLine       = do c <- getChar
                if c == '\n'
                   then return ""
                   else do l <- getLine
                           return (c:l)
```

Notice how 'imperative' this code is: Haskell is functional, and using monads allows for imperative features.

Notice that a function such as `f :: Int -> Int -> Int` cannot do any I/O since `IO` does not appear in the returned type.

I/O actions are ordinary Haskell values: they may be passed to functions, placed in structures, and used as any other Haskell value.

To deal with exceptional conditions such as 'file not found' within the I/O monad, a handling mechanism is used. No special syntax or semantics are used; exception handling is part of the definition of the I/O sequencing operations.

Errors are encoded using a special data type, `IOError`. This type represents all possible exceptions that may occur within the I/O monad. This is an abstract type: no constructors for `IOError` are available to the user. Predicates allow `IOError` values to be queried. For example, the function

```
isEOFError    :: IOError -> Bool
```

determines whether an error was caused by an end-of-file condition. By making `IOError` abstract, new sorts of errors may be added to the system without a noticeable change to the data type. The function `isEOFError` is defined in a separate library, `IO`, and must be explicitly imported into a program. An exception handler has type `IOError -> IO a`. The catch function associates an exception handler with an action or set of actions:

```
catch         :: IO a -> (IOError -> IO a) -> IO a
```

(Notice that `catch` is the `>>=` of the I/O monad.) The arguments to `catch` are an action and a handler. If the action succeeds, its result is returned without invoking the handler. If an error occurs, it is passed to the handler as a value of type `IOError` and the action associated with the handler is then invoked. For example, this version of `getChar` returns a newline when an error is encountered:

```
getChar'      :: IO Char
getChar'      = getChar `catch` (\e -> return '\n')
```

For convenience, Haskell provides a default exception handler at the topmost level of a program that prints out the exception and terminates the program.

References

[1] M. Abadi and M.P. Fiore. Syntactic Considerations on Recursive Types. In *Proceedings 11th Annual IEEE Symp. on Logic in Computer Science, LICS'96, New Brunswick, NJ, USA, 27–30 July 1996*, pages 242–252. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[2] Z.M. Ariola and M. Felleisen. The Call-By-Need Lambda Calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.

[3] S. van Bakel. Complete restrictions of the Intersection Type Discipline. *Theoretical Computer Science*, 102(1):135–163, 1992.

[4] S. van Bakel. Principal type schemes for the Strict Type Assignment System. *Journal of Logic and Computation*, 3(6):643–670, 1993.

[5] S. van Bakel. Intersection Type Assignment Systems. *Theoretical Computer Science*, 151(2):385–435, 1995.

[6] S. van Bakel. Rank 2 Intersection Type Assignment in Term Rewriting Systems. *Fundamenta Informaticae*, 2(26):141–166, 1996.

[7] S. van Bakel and M. Fernández. Normalisation Results for Typeable Rewrite Systems. *Information and Computation*, 2(133):73–116, 1997.

[8] S. van Bakel and M. Fernández. Normalisation, Approximation, and Semantics for Combinator Systems. *Theoretical Computer Science*, 290:975–1019, 2003.

[9] S. van Bakel, S. Smetsers, and S. Brock. Partial Type Assignment in Left Linear Applicative Term Rewriting Systems. In J.-C. Raoult, editor, *Proceedings of CAAP'92. 17th Colloqium on Trees in Algebra and Programming,* Rennes, France, volume 581 of *Lecture Notes in Computer Science*, pages 300–321. Springer Verlag, 1992.

[10] H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.

[11] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[12] T. Brus, M.C.J.D. van Eekelen, M.O. van Leer, and M.J. Plasmeijer. Clean - A Language for Functional Graph Rewriting. In *Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture,* Portland, Oregon, USA, volume 274 of *Lecture Notes in Computer Science*, pages 364–368. Springer Verlag, 1987.

[13] R.M. Burstall, D.B. MacQueen, and D.T. Sannella. Hope: An Experimental Applicative Language. In *Conference Record of the 1980 LISP Conference*, pages 136–143. ACM Press, 1980.

[14] A. Church. A Note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936.

[15] M. Coppo and M. Dezani-Ciancaglini. An Extension of the Basic Functionality Theory for the λ-Calculus. *Notre Dame journal of Formal Logic*, 21(4):685–693, 1980.

[16] M. Coppo, M. Dezani-Ciancaglini, F. Honsell, and G. Longo. Extended type structures and filter lambda models. In G. Lolli, G. Longo, and A. Marcja, editors, *Logic Colloquium 82*, pages 241–262, Amsterdam, the Netherlands, 1984. North-Holland.

[17] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and λ-calculus semantics. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry, Essays in combinatory logic, lambda-calculus and formalism*, pages 535–560. Academic press, New York, 1980.

[18] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

[19] M. Coppo, M. Dezani-Ciancaglini, and M. Zacchi. Type Theories, Normal Forms and D_∞-Lambda-Models. *Information and Computation*, 72(2):85–116, 1987.

[20] H.B. Curry. Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics*, 52:509–536, 789–834, 1930.

[21] H.B. Curry. Functionality in Combinatory Logic. In *Proc. Nat. Acad. Sci. U.S.A*, volume 20, pages 584–590, 1934.

[22] H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[23] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings* 9th *ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[24] L.M.M. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, Department of Computer Science, Edinburgh, 1985. Thesis CST-33-85.

[25] N. Dershowitz and J.P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 245–320. North-Holland, 1990.

[26] M. Dezani-Ciancaglini and J.R. Hindley. Intersection types for combinatory logic. *Theoretical Computer Science*, 100:303–324, 1992.

[27] F. Dupont. *Langage fonctionnels et parallélisme. Une réalisation Pour le système CAML*. PhD thesis, École Polytechnique, Palaiseau, France, July 1990.

[28] V. Gapeyev, M. Levin, and B. Pierce. Recursive Subtyping Revealed. In *International Conference on Functional Programming (ICFP), Montreal, Canada*, 2000.

[29] J.-Y. Girard. The System F of Variable Types, Fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[30] J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[31] P. Hudak, J. Peterson, and J Fasel. A Gentle Introduction to Haskell 98. Copyright ©1999 Paul Hudak, John Peterson and Joseph Fasel, 1999.

[32] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell. *ACM SIGPLAN Notices*, 27(5):1–64, 1992.

[33] A. Kfoury and J. Wells. Principality and decidable type inference for finite-rank intersection types. In *Proceedings of the 26th ACM Symposium on the Principles of Programming Languages (POPL '99)*, pages 161–174, 1999.

[34] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. A proper extension of ML with an effective type-assignment. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages,* San Diego, California, pages 58–69, 1988.

[35] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. ML Typability is Dexptime-Complete. In A. Arnold, editor, *Proceedings of CAAP'90. 15th Colloquim on Trees in Algebra and Programming,* Copenhagen, Denmark, volume 431 of *Lecture Notes in Computer Science*, pages 206–220. Springer Verlag, 1990.

[36] J.W. Klop. Term Rewriting Systems: a tutorial. *EATCS Bulletin*, 32:143–182, 1987.

[37] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–116. Clarendon Press, 1992.

[38] J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. In M. Dezani-Ciancaglini, S. Ronchi Della Rocca, and M Venturini Zilli, editors, *A Collection of contributions in honour of Corrado Böhm*, pages 279–308. Elsevier, 1993.

[39] L. Liquori and B. Wack. The Polymorphic Rewriting-calculus [Type Checking vs. Type Inference]. *Electronic Notes in Theoretical Computer Science*, 117:89–111, 2005.

[40] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[41] E. Moggi. Computational Lambda-calculus and Monads. In *Proceedings of Logic in Computer Science (LICS)*, pages 14–23, 1989.

[42] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93:55–92, 1991.

[43] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the International Symposium on Programming,* Toulouse, France, volume 167 of *Lecture Notes in Computer Science*, pages 217–239. Springer Verlag, 1984.

[44] F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Aanalysis*. Springer Verlag, 1999.

[45] V. van Oostrom. Lambda Calculus with Patterns. Technical Report IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, 1990.

[46] F. Pottier. A modern eye on ML type inference - Old techniques and recent developments, September 2005.

[47] D. Remy and J. Vouillon. Objective ML: An Effective Object-Oriented Extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[48] J.A. Robinson. A Machine-Oriented Logic Based on Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[49] S. Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59:181–209, 1988.

[50] S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28:151–169, 1984.

[51] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, 1985.

[52] P. Wadler. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer Verlag, 1995.

[53] J.B. Wells. The essence of principal typings. In *Proceedings of ICALP'92. 29th International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer Verlag, 2002.