Type Systems for Programming Languages

Course notes

Steffen van Bakel

Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2BZ, UK
s.vanbakel@imperial.ac.uk

Spring 2001
(revised Autumn 2022)

These notes accompany the course Type Systems for Programming Languages, given to students 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.

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 also makes *separate compilation* of program modules possible.

• Type systems warn the programmer in an early stage (at compile time) if a program contains *severe errors*. If a program is type-error free, it is safe to run: "*Typed programs cannot go wrong*" (Milner [43]). The meaning of this well-known quotation is the following: *a compile-time analysis of a program filters out certain errors in programs which might occur at run-time*, like applying a function defined on integers to a character.

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 studies systems that define type assignment in the context of functional languages. Because of the strong relation between the Lambda Calculus and functional programming, type assignment systems are normally defined and studied in the context of the 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 *pattern matching*, 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 having a look at intersection types and semantics.

Contents

1 The Lambda Calculus

The Lambda Calculus [15, 10] is a formalism, developed in the 1930s, 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 [53], Haskell [33], and CaML [27]. It gets its name from the Greek character λ (lambda). Church defined the λ-calculus as a way of formally defining *computation*, i.e. to start a (mathematical) process that in a finite number of steps produces a result. He thereby focussed on the normal mathematical notation of functions, and analysed what is needed to come to a notion of computation using that.

1.1 λ-terms

The set of *λ-terms*, ranged over by M, is constructed from a set of term-variables and two operations, *application* and *abstraction*. It is formally defined by:

Definition 1.1 (λ-TERMS) Let $\mathcal{V} = \{x, y, z, x_1, x_2, x_3, \ldots\}$ be a set of term-variables. Λ, the set of λ-terms is defined by the following grammar:

$$\begin{array}{rcccccc} M, N & ::= & x & \mid & (\lambda x.M) & \mid & (MN) \\ & & \textit{variable} & & \textit{abstraction} & & \textit{application} \end{array}$$

We say that M in $(M \cdot N)$ appears in *function position* and that N is an *argument*.

Or, in an inference system:

$$\frac{}{x \in \Lambda}\,(x \in \mathcal{V}) \qquad \frac{M \in \Lambda}{(\lambda x.M) \in \Lambda}\,(x \in \mathcal{V}) \qquad \frac{M \in \Lambda \qquad N \in \Lambda}{(MN) \in \Lambda}$$

The operation of application takes two terms M and N, and produces a new term, the application of M to N. You can see the term M as a function and N 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 an anonymous function; you can read $\lambda x.M$ as 'given the operand x, this function returns M'. In normal mathematics, functions are defined as in $sq\,x = x^2$; then we can use sq as a *name* for the square function, as in $sq\,3$. In the λ-calculus, we write $\lambda x.x^2$ rather than $sq\,x = x^2$, and $(\lambda x.x^2)\,3$ rather than $sq\,3$, so function definition and application are not separated.

Leftmost, outermost brackets can be omitted, so $MN\,(PQ)$ stands for $((MN)\,(PQ))$. Also, repeated abstractions can be abbreviated, so $\lambda xyz.M$ stands for $(\lambda x.(\lambda y.(\lambda z.M)))$.

As we will see below, computation in the λ-calculus is expressed through *term substitution*, where a term takes the position of a variable. Not all variables can replaced like this. Those x that occur underneath the scope of a corresponding λx (so occur in a subterm starting with that abstraction) are called *bound* and cannot be replaced; those that do not, are called *free* and can be replaced. These notions are formally defined by.

Definition 1.2 (FREE AND BOUND VARIABLES) The set of *free* variables of a term M ($fv\,(M)$) and its *bound* variables ($bv\,(M)$) are defined by:

$$\begin{array}{lcllcl} fv\,(x) & = & \{x\} & \qquad bv\,(x) & = & \emptyset \\ fv\,(\lambda y.M) & = & fv\,(M)\backslash\{y\} & \qquad bv\,(\lambda y.M) & = & bv\,(M) \cup \{y\} \\ fv\,(MN) & = & fv\,(M) \cup fv\,(N) & \qquad bv\,(MN) & = & bv\,(M) \cup bv\,(N) \end{array}$$

We write $x \notin M$ for $x \notin fv\,(M) \cup bv\,(M)$.

We call M *closed* if M contains no free variables, *i.e.* $fv\,(M) = \emptyset$.

To avoid ambiguity, we will assume that bound and free variables are always different (this is called *Barendregt's convention*), renaming bound variables when necessary (see below).

In calculating using functions, we need the operation of substitution (normally not formally specified), to express that the parameter replaces the variable in a function, as in $(\lambda x.x^2)3$ becomes 3^2. This feature, replacing all free occurrences of x in a term M by the term N, denoted by $M[N/x]$, is at the basis of the computational step of the λ-calculus.

Definition 1.3 The substitution of the term variable x by the term N is defined inductively over the structure of terms by:

$$\begin{array}{rcll} x[N/x] & = & N & \\ y[N/x] & = & y & (y \neq x) \\ (PQ)[N/x] & = & P[N/x]\,Q[N/x] & \end{array} \qquad \begin{array}{rcll} (\lambda y.M)[N/x] & = & \lambda y.(M[N/x]) & (y \neq x) \\ (\lambda x.M)[N/x] & = & \lambda x.M & \end{array}$$

We will see below the problem this definition gives in terms of inadvertently binding free variables, and how to avoid that from happening.

1.2 β and α-conversion

On Λ, the basic computational step is that of effecting the replacement of a bound variable in an abstraction by the parameter of the application. The notion of computation is defined as a relation '$\rightarrow_\beta$' on terms that specifies that, if $M \rightarrow_\beta N$, then M executes 'in one step' to N.

Definition 1.4 (β-CONVERSION) *i*) The β-reduction rule is defined by:

$$(\lambda x.M)\,N \;\rightarrow_\beta\; M\,[N/x]$$

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

ii) The binary *one-step* reduction relation '$\rightarrow_\beta$' on λ-terms (as usual, we will write $M \rightarrow_\beta N$ rather than $\langle M,N \rangle \in \rightarrow_\beta$) is defined by adding the 'contextual closure' rules:

$$M \rightarrow_\beta N \;\Rightarrow\; \begin{cases} \lambda x.M \;\rightarrow_\beta\; \lambda x.N \\ \;\;PM \;\rightarrow_\beta\; PN \\ \;\;MP \;\rightarrow_\beta\; NP \end{cases}$$

iii) The reduction relation '$\rightarrow_\beta^*$' (sometimes also written as '$\twoheadrightarrow_\beta$'), called β-reduction, is defined as the reflexive, transitive closure of '$\rightarrow_\beta$':

$$\begin{array}{rcl} M \rightarrow_\beta N & \Rightarrow & M \rightarrow_\beta^* N \\ & M \rightarrow_\beta^* M & \\ M \rightarrow_\beta^* N \wedge N \rightarrow_\beta^* P & \Rightarrow & M \rightarrow_\beta^* P \end{array}$$

iv) '$=_\beta$' is the equivalence relation generated by '$\rightarrow_\beta^*$':

$$\begin{array}{rcl} M \rightarrow_\beta^* N & \Rightarrow & M =_\beta N \\ M =_\beta N & \Rightarrow & N =_\beta M \\ M =_\beta N \wedge N =_\beta P & \Rightarrow & M =_\beta P \end{array}$$

We can define these relations also using inference rules as in Figure 1.

This notion of reduction is actually directly based on function application in mathematics. To illustrate this, take again function f defined by $f x = x^2$, 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.

Example 1.5 We show some examples of reduction and equality. Notice that we have both:

$$
\begin{array}{lll}
(\beta): \dfrac{}{(\lambda x.M)\,N \to_\beta M\,[N/x]} & (\mathit{Inherit}_r): \dfrac{M \to_\beta N}{M \to_\beta^* N} & (\mathit{Inherit}_{eq}): \dfrac{M \to_\beta^* N}{M =_\beta N} \\
(\mathit{Appl\text{-}L}): \dfrac{M \to_\beta N}{MP \to_\beta NP} & (\mathit{Refl}): \dfrac{}{M \to_\beta^* M} & (\mathit{Symm}): \dfrac{M =_\beta N}{N =_\beta M} \\
(\mathit{Appl\text{-}R}): \dfrac{M \to_\beta N}{PM \to_\beta PN} & (\mathit{Trans}_r): \dfrac{M \to_\beta^* N \quad N \to_\beta^* P}{M \to_\beta^* P} & (\mathit{Trans}_{eq}): \dfrac{M =_\beta N \quad N =_\beta P}{M =_\beta P} \\
(\mathit{Abstr}): \dfrac{M \to_\beta N}{\lambda x.M \to_\beta \lambda x.N} & &
\end{array}
$$

Figure 1. Definition of '$\to_\beta$', '$\to_\beta^*$', and '$=_\beta$' through inference rules

$$
\begin{array}{ll}
(\lambda xyz.xz\,(yz))\,(\lambda ab.a) & (\lambda uv.uv)\,(\lambda c.c)\,(\lambda yz.z) \\
\quad \to_\beta \ \lambda yz.(\lambda ab.a)\,z\,(yz) & \quad \to_\beta \ (\lambda v.(\lambda c.c)\,v)\,(\lambda yz.z) \\
\quad \to_\beta \ \lambda yz.(\lambda b.z)\,(yz) & \quad \to_\beta \ (\lambda c.c)\,(\lambda yz.z) \\
\quad \to_\beta \ \lambda yz.z & \quad \to_\beta \ \lambda yz.z
\end{array}
$$

so we have both

$$
\begin{array}{rcl}
(\lambda xyz.xz\,(yz))\,(\lambda ab.a) & \to_\beta^* & \lambda yz.z \\
(\lambda uv.uv)\,(\lambda c.c)\,(\lambda yz.z) & \to_\beta^* & \lambda yz.z
\end{array}
$$

and thereby

$$
(\lambda xyz.xz\,(yz))\,(\lambda ab.a) \ =_\beta \ (\lambda uv.uv)\,(\lambda c.c)\,(\lambda yz.z)
$$

It is important to note that reduction as defined above will break Barendregt's convention. Take for example the term $(\lambda xy.xy)\,(\lambda xy.xy)$, which adheres to the convention. Reducing this term using the above definition for term substitution would give:

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

Notice that this term no longer adheres to the 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. When, however, we continue with the reduction, we get

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

We would now be forced to accept that *both* ys are bound by the innermost λy, which would be wrong. The problem is that the free occurrence of y becomes bound during substitution; this is called a *variable capture*.

At the 'human' level, to not have to worry about variable capture, we use a notion of equivalence (or rather, *convergence*) on terms; it considers terms equivalent that can be obtained from each other by renaming bound variables. It corresponds to the mathematical idea that the functions $f\,x = x^2$ and $g\,y = y^2$ are identical. Essentially, this relation, called *α-conversion*, is defined through

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

extending it to all terms much in the spirit of β-conversion. We can avoid the capture problem above by α-converting the term $\lambda y.(\lambda xy.xy)y$ to $\lambda y.(\lambda xz.xz)y$ before performing the reduction.

So the replacement of a variable x by a term N should perhaps be defined by:

$$
\begin{array}{rcll}
x\,[N/x] & = & N & \\
y\,[N/x] & = & y & (y \neq x) \\
(\lambda y.M)\,[N/x] & = & \lambda y.(M\,[N/x]) & (y \notin fv\,(N)\ \&\ y \neq x) \\
(PQ)\,[N/x] & = & P\,[N/x]\,Q\,[N/x] & \\
(\lambda y.M)\,[N/x] & = & \lambda z.(M\,[z/y])\,[N/x] & (y \in fv\,(N)\ \&\ y \neq x, z\ \mathit{fresh}) \\
(\lambda x.M)\,[N/x] & = & \lambda x.M &
\end{array}
$$

a rather complicated definition, and the one used in implementation (rather CPU-heavy because of the additional 'renaming' $[z/y]$); notice the α-conversion taking place in the fourth line. Using this definition, we get:

$$
\begin{array}{rcll}
(\lambda xy.xy)\,(\lambda xy.xy) & \to & (\lambda y.xy)\,[(\lambda xy.xy)/x] & = \ \lambda y.(\lambda xy.xy)\,y \\
& \to & \lambda y.(\lambda y.xy)\,[y/x] & =_\alpha \ \lambda y.(\lambda z.xz)\,[y/x] \\
& = & \lambda y.(\lambda z.yz) & = \ \lambda yz.yz
\end{array}
$$

a reduction that prevents capture, but does not preserve the convention.

In practice, we abstract away from these difficulties: we will assume that all terms satisfy Barendregt's convention and use α-conversion freely to preserve the convention during reduction. We therefore will be using the definition:

Definition 1.6 (Term substitution) The substitution of the term variable x by the term N is defined inductively over the structure of terms by:

$$
\begin{array}{rcll@{\qquad}rcl}
x\,[N/x] & = & N & & (PQ)\,[N/x] & = & P\,[N/x]\,Q\,[N/x] \\
y\,[N/x] & = & y & (y \neq x) & (\lambda y.M)\,[N/x] & = & \lambda y.(M\,[N/x])
\end{array}
$$

By Barendregt's convention, in $(\lambda x.M)\,N$ not only can we assume that x does not occur free in N, but also that all bound variables in M are not free in N; therefore, in $(\lambda y.M)\,[N/x]$, we can assume the same, and do not need to check if variable capture will occur. This implies that we reduce the above term as follow:

$$
\begin{array}{rcll}
(\lambda xy.xy)\,(\lambda xy.xy) & \to & (\lambda z.xz)\,[(\lambda xy.xy)/x] & \\
& \to & \lambda z.(\lambda xy.xy)\,z & = \ \lambda zy.zy
\end{array}
$$

or

$$
\begin{array}{rcll}
(\lambda xy.xy)\,(\lambda xy.xy) & \to & (\lambda y.xy)\,[(\lambda xz.xz)/x] & \\
& \to & \lambda y.(\lambda xz.xz)\,y & = \ \lambda yz.yz
\end{array}
$$

(notice that $\lambda zy.zy =_\alpha \lambda yz.yz$).

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, as in mathematics, α-conversion plays no active role; terms are considered modulo α-conversion.

In implementations of the λ-calculus, it is normal that a *strategy* is used, that systematically picks a redex amongst those available. The one that stands, for example, at the basis of reduction in Haskell [33], is that of *lazy* reduction, whereas for languages based on ML (see Section 5) call-by-value is used.

Definition 1.7 (Reduction strategies) *i*) We define *head reduction* '$\to_{\mathrm{H}}$' as a restriction of '$\to_\beta$' by:

$$
\frac{}{(\lambda x.M)\,N \to_{\mathrm{H}} M\,[N/x]} \qquad \frac{M \to_{\mathrm{H}} N}{\lambda x.M \to_{\mathrm{H}} \lambda x.N} \qquad \frac{M \to_{\mathrm{H}} N}{MP \to_{\mathrm{H}} NP}
$$

ii) *Call-by-name* reduction (CBN, also known as *lazy* reduction) is defined through:

$$
\frac{}{(\lambda x.M)\,N \to_{\mathrm{N}} M\,[N/x]} \qquad \frac{M \to_{\mathrm{N}} N}{MP \to_{\mathrm{N}} NP}
$$

iii) We consider all abstractions and variables *values*, and use V to denote those[1]. *Call-by-value* reduction (CBV) is defined through:

$$\frac{}{(\lambda x.M)\,V \rightarrow_{\mathrm{V}} M[V/x]} \qquad \frac{M \rightarrow_{\mathrm{V}} N}{MP \rightarrow_{\mathrm{V}} NP} \qquad \frac{M \rightarrow_{\mathrm{V}} N}{VM \rightarrow_{\mathrm{V}} VN}$$

iv) *Normal order* reduction is defined through:

$$\frac{}{(\lambda x.M)\,N \rightarrow_{\mathrm{N}} M\,[N/x]} \qquad \frac{M \rightarrow_{\mathrm{N}} N}{MP \rightarrow_{\mathrm{N}} NP} \qquad \frac{M \rightarrow_{\mathrm{N}} N}{PM \rightarrow_{\mathrm{N}} PN}\,(P \textit{ contains no redex}) \qquad \frac{M \rightarrow_{\mathrm{N}} N}{\lambda x.M \rightarrow_{\mathrm{N}} \lambda x.N}$$

v) *Applicative order* reduction is defined through:

$$\frac{}{(\lambda x.M)\,N \rightarrow_{\mathrm{A}} M\,[N/x]}\,(M,N \textit{ contain no redex}) \qquad \frac{M \rightarrow_{\mathrm{A}} N}{MP \rightarrow_{\mathrm{A}} NP}$$

$$\frac{M \rightarrow_{\mathrm{A}} N}{PM \rightarrow_{\mathrm{A}} PN}\,(P \textit{ contains no redex}) \qquad \frac{M \rightarrow_{\mathrm{A}} N}{\lambda x.M \rightarrow_{\mathrm{A}} \lambda x.N}$$

Another way of defining some of these strategies is through the notion of evaluation contexts.

Definition 1.8 *i*) *Evaluation contexts* are defined as terms with a single hole $\lceil\,\rfloor$ by:

$$\mathsf{C} \ ::= \ \lceil\,\rfloor \mid \mathsf{C}M \mid M\mathsf{C} \mid \lambda x.\mathsf{C}$$

We write $\mathsf{C}\lceil M\rfloor$ for the term obtained from the context C by replacing its hole $\lceil\,\rfloor$ with M, allowing variables to be captured.

ii) Then one-step β reduction can be defined through:

$$\mathsf{C}\lceil(\lambda x.M)\,N\rfloor \ \rightarrow \ \mathsf{C}\lceil M[N/x]\rfloor$$

for any evaluation context.

iii) CBN *evaluation contexts* are defined through:

$$\mathsf{C}_{\mathrm{N}} \ ::= \ \lceil\,\rfloor \mid \mathsf{C}_{\mathrm{N}}M$$

The *Call-by-name* (CBN) reduction strategy $\rightarrow_{\beta}{}^{\mathrm{N}}$ is defined through:

$$\mathsf{C}_{\mathrm{N}}\lceil(\lambda x.M)N\rfloor \ \rightarrow \ \mathsf{C}_{\mathrm{N}}\lceil M[N/x]\rfloor$$

iv) Call-by-value *evaluation contexts* are defined through:

$$\mathsf{C}_{\mathrm{V}} \ ::= \ \lceil\,\rfloor \mid \mathsf{C}_{\mathrm{V}}M \mid V\mathsf{C}_{\mathrm{V}}$$

The *Call-by-value* (CBV) reduction strategy $\rightarrow_{\beta}{}^{\mathrm{V}}$ is defined through:

$$\mathsf{C}_{\mathrm{V}}\lceil(\lambda x.M)\,V\rfloor \ \rightarrow \ \mathsf{C}_{\mathrm{V}}\lceil M[V/x]\rfloor$$

Notice that CBN and CBV are called strategies since in both there can only ever be one redex to contract.

Notice that for all these notions, reduction is limited by omitting contextual rules, so in CBN-reduction it is impossible to reduce under abstraction or in the right-hand side of an application; a redex will only be contracted if it occurs at the start of the term.

CBV-reduction does the same, until the term it is reducing becomes a value; it then runs the argument until that becomes a value, thus creating a redex where the right-hand term is a value, and contracts that redex. Some of these notions do not reduce redexes that occur *inside* abstractions (whereas, for example, head reduction does); for those notions, an abstraction is considered to be 'finished' (as are numbers, for example, and variables) and are therefore called values.

The following terms will reappear frequently in these notes:

[1] We could add numbers, characters, etc to these.

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

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

Definition 1.9 (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 '$\rightarrow_\beta^*$' as a 'many step' execution. We can then view the relation '$=_\beta$' as 'executing the same function'.

This notion of reduction satisfies the 'Church-Rosser Property' or confluence:

Property 1.10 (CHURCH-ROSSER) *i) If $M \rightarrow_\beta^* N$ and $M \rightarrow_\beta^* P$, then there exists a term Q such that $N \rightarrow_\beta^* Q$ and $P \rightarrow_\beta^* Q$.*

ii) If $M =_\beta N$ and $M \rightarrow_\beta^* P$, then there exists Q such that $P \rightarrow_\beta^* Q$ and $N \rightarrow_\beta^* Q$.

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

(This is, for obvious reasons, also called the *diamond property*). So diverging computations can always be joined.

Although the λ-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 the λ-calculus. In particular, it is possible to have non-terminating terms.

Example 1.11 *i*) Take $(\lambda x.xx)\,(\lambda x.xx)$. This term reduces as follows:

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

so this term reduces only to itself.

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

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

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 $MN =_\beta N$: we will come back to this in Section 4.

Theorem 1.12 (FIXED-POINT THEOREM) *For every term M there exists a term N (the fixed-point of M), such that $MN =_\beta N$.*

Proof: The proof is given by taking $N = YM$, where $Y = \lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx))$.

$$
\begin{aligned}
YM &\triangleq (\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx)))\,M \\
&\rightarrow_\beta (\lambda x.M(xx))\,(\lambda x.M(xx)) \\
&\rightarrow_\beta M((\lambda x.M(xx))(\lambda x.M(xx)))
\end{aligned}
$$

and

$$
\begin{aligned}
M(YM) &\triangleq M((\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx)))\,M) \\
&\rightarrow_\beta M((\lambda x.M(xx))(\lambda x.M(xx)))
\end{aligned}
$$

So in particular, $MN \triangleq M(YM) =_\beta YM \triangleq N$.

So Y is a fixed-point constructor, i.e. given M, YM is a fixed-point of M, i.e. $M(YM) =_\beta YM$. There are many different fixed-point constructors.

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

Definition 1.13 *i*) A term is *in normal form* if it does not contain a redex. Terms in normal form can be defined by:

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

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

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

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

iv) We call a term without head-normal form *meaningless* (it can never interact with any context).

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

If a term has a head-normal form, then head-reduction on that term is terminating.

Example 1.14 *i*) The term $\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx))$ contains an occurrence 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, so the first term has a head normal form. Notice that it is not in normal form, since 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 normalisable. Also, we have contracted all possible redexes, so the original term is strongly normalisable.

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 normalisable.

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

1.3 Approximation semantics

There are various ways of giving meaning to the λ-calculus. Some people just consider the reduction rules, and then speak of '*operational semantics*', others consider Set Theory the only real setting for semantics and use '*denotational semantics*', often using Scott domains [30].

Here we will use a more light-weight approach, and define a semantics that is denotational in character, but uses the reduction system for its definition. It uses a notion of approximant for λ-terms, as was first presented by Wadsworth [54], which is defined using the notion of terms in $\Lambda\bot$-normal form. It is based on an extension of the syntax of terms by adding the term constant $\bot$, which operationally stands for 'unknown', 'meaningless', or 'no information'. It is used to mask sub-terms, typically those containing redexes, and allows to focus on the 'stable part' of a term, the part that will/can no longer change as a result of reduction.

Definition 1.15 *i*) The set $\Lambda\bot$ of *$\Lambda\bot$-terms* is defined by adding the term constant $\bot$:

$$M,N \ ::= \ x \mid \bot \mid \lambda x.M \mid MN$$

ii) The notion of reduction '$\rightarrow_\bot$' is defined as '$\rightarrow_\beta$', extended by:

$$\begin{array}{rcl} \lambda x.\bot & \rightarrow_\bot & \bot \\ \bot M & \rightarrow_\bot & \bot \end{array}$$

iii) The set of *normal forms for elements of $\Lambda\bot$, with respect to* '$\rightarrow_\bot$', is the set $\mathcal{A}$ of *$\lambda\bot$-normal forms* or *approximate normal forms*, ranged over by A, defined by:

$$\mathsf{A} \ ::= \ \bot \mid \lambda x.\mathsf{A} \ (\mathsf{A} \neq \bot) \mid x\mathsf{A}_1 \cdots \mathsf{A}_n \ (n \geq 0)$$

Remark that approximate normal forms are defined in much the same way as terms in normal form in Definition 1.13, but taking care of the case that $\lambda x.\bot$ is considered to be a redex. It is easy to show that the approximate normal forms are the normal forms with respect to '$\rightarrow_\bot$'-reduction.

As can be seen from the second part, the reduction system is set up, as before, to reduce terms to redex-free expressions, but the presence of $\bot$ changes the behaviour of terms quite dramatically. In the reduction system, it acts a a 'sink hole' for terms; any applicative term starting with $\bot$ will run to $\bot$: this is because, given that $\bot$ represents that fact that the first term is 'unknown', anything can happen: replacing $\bot$ by an appropriate term, all arguments can be discarded and therefore we can only safely assume that 'unknown' will be produced for the application. Similarly, any function that returns 'unknown' will have to act as 'unknown' itself.

We will now define the notion of *approximant*. These are $\lambda\bot$-normal forms, redex-free terms that can contain $\bot$, and are used to be able to repesent finite parts of possibly infinitely large λ-terms in head-normal form.

Definition 1.16 (Approximants) *i*) The partial order $\sqsubseteq \subseteq (\Lambda\bot)^2$ is defined as the smallest pre-order (i.e. reflexive and transitive relation) such that:

$$\begin{array}{rclrcl} \bot & \sqsubseteq & M & M \sqsubseteq M' & \Rightarrow & \lambda x.M \sqsubseteq \lambda x.M' \\ x & \sqsubseteq & x & M_1 \sqsubseteq M_1' \wedge M_2 \sqsubseteq M_2' & \Rightarrow & M_1M_2 \sqsubseteq M_1'M_2' \end{array}$$

ii) For $\mathsf{A} \in \mathcal{A}$, $M \in \Lambda$, if $\mathsf{A} \sqsubseteq M$, then A is called a *direct approximant* of M.

iii) The set of *approximants of* M, $\mathcal{A}(M)$, is defined through:

$$\mathcal{A}(M) \ \triangleq \ \{\, \mathsf{A} \in \mathcal{A} \mid \exists M' \in \Lambda\, (M \rightarrow_\beta^* M' \wedge \mathsf{A} \sqsubseteq M') \,\}.$$

Notice that if A is a direct approximant of M, then A and M have the same structure, except in places where A contains $\bot$ (notice that M is a pure λ-term); also M might contain redexes, but A will then have $\bot$ in that (or larger) location. The set of approximants of a term is built up using reduction, and after each step constructing the approximants that are below each resulting term.

Example 1.17 We have seen that

$$(\lambda xyz.xz(yz))\,(\lambda ab.a) \ \rightarrow_\beta \ \lambda yz.(\lambda ab.a)z(yz) \ \rightarrow_\beta \ \lambda yz.(\lambda b.z)\,(yz) \ \rightarrow_\beta \ \lambda yz.z$$

and the direct approximants of these terms are, respectively, $\bot$, $\bot$, $\bot$, and $\{\,\bot, \lambda yz.z\,\}$, so

$$\mathcal{A}((\lambda xyz.xz(yz))\,(\lambda ab.a)) \ = \ \{\,\bot,\ \lambda yz.z\,\}$$

Likewise,

$$(\lambda xyz.xz(yz))\,a\,(\lambda cd.c) \;\to_\beta\; (\lambda yz.az(yz))\,(\lambda cd.c) \;\to_\beta\; \lambda z.az((\lambda cd.c)\,z) \;\to_\beta\; \lambda z.az(\lambda d.z)$$

with direct approximants, respectively,

$$\bot,\ \bot,\ \{\bot,\ \lambda z.a\bot\bot,\ \lambda z.az\bot\},\ \{\bot,\ \lambda z.a\bot\bot,\ \lambda z.az\bot,\ \lambda z.a\bot(\lambda d.z),\ \lambda z.az(\lambda d.z)\}$$

Also:

$$\begin{array}{rcl}
\mathcal{A}(\lambda x.x) & = & \{\bot,\ \lambda x.x\} \\
\mathcal{A}(\lambda x.xx) & = & \{\bot,\ \lambda x.x\bot,\ \lambda x.xx\} \\
\mathcal{A}(\lambda x.x((\lambda y.yy)(\lambda y.yy))) & = & \{\bot,\ \lambda x.x\bot\} \\
\mathcal{A}(\lambda xyz.xz(yz)) & = & \{\bot,\ \lambda xyz.x\bot\bot,\ \lambda xyz.x\bot(y\bot),\ \lambda xyz.x\bot(yz), \\
 & & \quad \lambda xyz.xz\bot,\ \lambda xyz.xz(y\bot),\ \lambda xyz.xz(yz)\} \\
\mathcal{A}(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))) & = & \{\bot,\ \lambda f.f\bot,\ \lambda f.f(f\bot),\ \lambda f.f(f(f\bot)),\ \ldots\}
\end{array}$$

The following properties of approximants hold:

Proposition 1.18 *i) If* $\mathsf{A} \in \mathcal{A}(xM_1\cdots M_n)$, $\mathsf{A} \neq \bot$ *and* $\mathsf{A}' \in \mathcal{A}(N)$, *then also* $\mathsf{A}\mathsf{A}' \in \mathcal{A}(xM_1\cdots M_nN)$.

ii) If $\mathsf{A} \in \mathcal{A}(Mz)$ *and* $z \notin fv(M)$, *then either* $\mathsf{A} = \bot$, *or:*

- $\mathsf{A} \equiv \mathsf{A}'z$, $z \notin fv(\mathsf{A}')$, *and* $\mathsf{A}' \in \mathcal{A}(M)$, *or*
- $\lambda x.\mathsf{A} \in \mathcal{A}(M)$.

iii) If $\mathsf{A} \sqsubseteq M$ *and* $M \to_\beta^* N$, *then* $\mathsf{A} \sqsubseteq N$.

iv) If $\mathsf{A} \in \mathcal{A}(M)$ *and* $M \to_\beta^* N$, *then also* $\mathsf{A} \in \mathcal{A}(N)$.

v) If $\mathsf{A} \in \mathcal{A}(N)$ *and* $M \to_\beta^* N$, *then also* $\mathsf{A} \in \mathcal{A}(M)$.

We leave the proof of these properties as an exercise.

The following definition introduces an operation of *join* on $\Lambda\bot$-terms.

Definition 1.19 *i*) On $\Lambda\bot$, the partial mapping *join*, $\sqcup : \Lambda\bot \times \Lambda\bot \to \Lambda\bot$, is defined by:

$$\begin{array}{rclcrcl}
\bot \sqcup M \equiv M \sqcup \bot & \equiv & M & \qquad & (\lambda x.M) \sqcup (\lambda x.N) & \equiv & \lambda x.(M \sqcup N) \\
x \sqcup x & \equiv & x & & (M_1M_2) \sqcup (N_1N_2) & \equiv & (M_1 \sqcup N_1)\,(M_2 \sqcup N_2)
\end{array}$$

ii) If $M \sqcup N$ is defined, then M and N are called *compatible*.

So compatible terms are equal, except for positions where one contains $\bot$ and the other does not; the join of (compatible) terms is then built out of these two terms, choosing the non-$\bot$ part each time (if possible; $\bot$ might appear in both terms in the same position).

Notice that, e.g., $(\lambda x.x) \sqcup (\lambda yz.yz)$ is undefined, so a join of two arbitrary terms does not always exist. However, that is only an issue if the function should be total, which is not the case here; our definition will only be used on terms that are compatible.

The following lemma shows that '$\sqcup$' acts as least upper bound of compatible terms.

Lemma 1.20 *If* $M_1 \sqsubseteq M$, *and* $M_2 \sqsubseteq M$, *then* $M_1 \sqcup M_2$ *is defined, and:*

$$M_1 \sqsubseteq M_1 \sqcup M_2, M_2 \sqsubseteq M_1 \sqcup M_2, \textit{ and } M_1 \sqcup M_2 \sqsubseteq M.$$

Notice that $M_1 \sqsubseteq (M_1 \sqcup M_2) \sqsubseteq (M_1 \sqcup M_2 \sqcup M_3) \sqsubseteq \cdots$, so it is natural to consider $\bot$ to be the empty join, i.e. if $M \equiv M_1 \sqcup \cdots \sqcup M_n$, and $n = 0$, then $M \equiv \bot$.

$\mathcal{A}(\cdot)$ can be used to define a semantics for the Lambda Calculus.

Theorem 1.21 *If* $M =_\beta N$, *then* $\mathcal{A}(M) = \mathcal{A}(N)$.

Proof: By induction on the definition of '$=_\beta$', using part (*iv*) and (*v*) of Proposition 1.18.

In fact, we can show that overlaying all approximants for M creates $BT(M)$, the *Böhm tree* of M, a tree that represents the (possible infinite) normal form of M (see [10]).

Notice that the reverse of this property does not hold. We have $\mathcal{A}((\lambda x.xx)\,(\lambda x.xx)) = \{\bot\} = \mathcal{A}((\lambda x.xxx)\,(\lambda x.xxx))$, but these terms are not related through $=_\beta$. So this semantics is not *fully abstract*.

1.4 Making substitution explicit

In implementations (and semantics) of the λ-calculus, implicit substitution $[N/x]$ on terms creates particular problems; remark that in $M[N/x]$, the substitution is assumed to be instantaneous, irrespective of where x occurs in M. Of course, in an implementation, substitution during execution of a program comes with a cost; many approaches to implement substitution efficiently exist, varying from string reduction, λ-graphs, and Krivine's machine [40].

Normally, a calculus of *explicit substitutions* [12, 1, 42, 41], where substitution is a part of the syntax of terms, is considered better suited for an accurate account of the substitution process and its implementation. There are many variants of such calculi; the one we look at here is $\lambda\mathbf{x}$, the calculus of *explicit substitution with explicit names*, defined by Bloo and Rose [12]. $\lambda\mathbf{x}$ gives a better account of substitution as it integrates substitutions as first class citizens by extending the syntax with the construct $M\langle x := N\rangle$, decomposes the process of inserting a term into atomic actions, and explains in detail how substitutions are distributed through terms to be eventually evaluated at the variable level.

Definition 1.22 (Explicit λ-calculus $\lambda\mathbf{x}$ cf. [12]) *i*) The syntax of the *explicit lambda calculus* $\lambda\mathbf{x}$ is defined by:

$$M, N \;::=\; x \mid \lambda x.M \mid MN \mid M\langle x := N\rangle$$

The notion of bound variable of Definition 1.2 is extended by: occurrences of x in M are bound in $M\langle x := N\rangle$ (and by Barendregt's convention, then x cannot appear in N).

ii) The reduction relation '$\rightarrow_{\mathrm{x}}$' on terms in $\lambda\mathbf{x}$ is defined by the following rules:

$$\begin{array}{rcll}
(\lambda x.M)\,N & \rightarrow & M\langle x := N\rangle & \\
(MN)\langle x := L\rangle & \rightarrow & (M\langle x := L\rangle)\,(N\langle x := L\rangle) & \\
(\lambda y.M)\langle x := L\rangle & \rightarrow & \lambda y.(M\langle x := L\rangle) & \\
x\langle x := L\rangle & \rightarrow & L & \\
M\langle x := L\rangle & \rightarrow & M & (x \notin fv(M))
\end{array}
\qquad
M \rightarrow N \;\Rightarrow\; \left\{\begin{array}{rcl}
\lambda x.M & \rightarrow & \lambda x.N \\
ML & \rightarrow & NL \\
LM & \rightarrow & LN \\
M\langle x := L\rangle & \rightarrow & N\langle x := L\rangle \\
L\langle x := M\rangle & \rightarrow & L\langle x := N\rangle
\end{array}\right.$$

Notice that we allow reduction *inside* the substitution.

iii) We write '$\rightarrow_{:=}$' if the rule (β) does not get applied in the reduction.

Note that the rule $M\langle x := N\rangle\langle y := L\rangle \rightarrow M\langle y := L\rangle\langle x := N\langle y := L\rangle\rangle$ is not part of the reduction rules: its addition would lead to undesired non-termination.

As for '$\rightarrow_\beta$', reduction using '$\rightarrow_{\mathrm{x}}$' is confluent. It is easy to show that '$\rightarrow_{:=}$' is (on its own) a terminating reduction, *i.e.* there are no infinite reduction sequences in '$\rightarrow^*_{:=}$'. Thereby, if no reduction starting from M terminates, then every reduction sequence in '$\rightarrow_{\mathrm{x}}$' starting from M has an infinite number of β-steps.

The rules express reduction as a term rewriting system (see Section 6). Explicit substitution describes explicitly the process of executing a β-reduction, *i.e.* expresses syntactically the computation as a succession of atomic, constant-time steps, where the implicit substitution of the β-reduction step is split into several steps. Therefore, the following is easy to show:

Proposition 1.23 (λx implements β-reduction) *i)* $M \rightarrow_\beta N \Rightarrow M \rightarrow^*_{\mathrm{x}} N$.
ii) $M \in \Lambda \wedge M \rightarrow^*_{\mathrm{x}} N \Rightarrow \exists L \in \Lambda\,(N \rightarrow^*_{:=} L \wedge M \rightarrow^*_\beta L)$.

* 1.5 Example: a numeral system

We will now show the expressivity of the λ-calculus 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 1.24 We define the *booleans* True and False as:

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

Then the *conditional* is defined as:

$$\mathsf{Cond} \;=\; \lambda btf.btf$$

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

Definition 1.25 The operation of *pairing* of terms is defined via

$$\langle M,N \rangle \;=\; \lambda z.zMN$$

(or *pair* $= \lambda xyz.zxy$) and the first and second projection functions are defined by

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

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 1.26 The (Scott) *Numerals* are defined by:

$$\begin{array}{rcl}\lceil\!\lceil 0 \rfloor\!\rfloor & = & \mathbf{K} \\ \mathsf{Succ} & = & \lambda nxy.yn \\ \mathsf{Pred} & = & \langle \mathbf{K},\mathbf{I} \rangle \quad = \quad \lambda p.p\mathbf{KI}\end{array}$$

So, for example,

$$\begin{array}{rcl}\lceil\!\lceil 3 \rfloor\!\rfloor & = & \lceil\!\lceil S(S(S(0))) \rfloor\!\rfloor \\ & = & \mathsf{Succ}\ (\mathsf{Succ}\ (\mathsf{Succ}\ \lceil\!\lceil 0 \rfloor\!\rfloor)) \\ & = & (\lambda nxy.yn)((\lambda nxy.yn)((\lambda nxy.yn)\mathbf{K})) \\ & \to_\beta & \lambda ab.b((\lambda nxy.yn)((\lambda nxy.yn)\mathbf{K})) \\ & \to_\beta & \lambda ab.b(\lambda cd.d((\lambda nxy.yn)\mathbf{K})) \\ & \to_\beta & \lambda ab.b(\lambda cd.d(\lambda xy.y\mathbf{K}))\end{array}$$

Notice that $\lceil\!\lceil 0 \rfloor\!\rfloor = \mathsf{True}$. It is now easy to check that

$$\mathsf{Cond}\ \lceil\!\lceil n \rfloor\!\rfloor\ f\ g \;\to_\beta^*\; \begin{cases} f & (\lceil\!\lceil n \rfloor\!\rfloor = 0) \\ g\ \lceil\!\lceil n-1 \rfloor\!\rfloor & (\text{otherwise}) \end{cases}$$

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 1.27 Addition and multiplication are now defined by:

$$\begin{array}{rcl}\mathsf{Add} & = & \lambda nm.\mathsf{Cond}\ (\mathsf{IsZero}\ n)\ m\ (\mathsf{Succ}\ (\mathsf{Add}\ (\mathsf{Pred}\ n)\ m)) \\ \mathsf{Mult} & = & \lambda nm.\mathsf{Cond}\ (\mathsf{IsZero}\ n)\ \mathsf{Zero}\ (\mathsf{Add}\ (\mathsf{Mult}\ (\mathsf{Pred}\ n)\ m)\ m)\end{array}$$

Notice that these definitions are recursive; we will see in Section 4 that we can express recursion in the λ-calculus.

There are alternative ways of encoding numbers in the λ-calculus, like the *Church Numerals* $\underline{n} = \lambda fn.f^n x$ which would give an elegant encoding of addition and multiplication that do not depend on recursion, but has a very complicated definition of predecessor.

Exercises

Exercise 1.28 Write the following terms in the original, full notation:

i) $\lambda xyz.xzyz$.

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

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

Remove the omissible brackets in the following terms:

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

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

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

Exercise 1.29 *Assume that* $x \notin fv\,(L)$ *and* $x \neq y$. *Show that* $M\,[N/x]\,[L/y] = M\,[L/y]\,[N\,[L/y]/x]$.

Exercise 1.30 *Establish for each of the following terms if they have a head-normal form, or even a normal form.*

i) $(\lambda xyz.xz(yz))\,(\lambda ab.a)$.

ii) $(\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx)))\,(\lambda a.a)$.

iii) $(\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx)))\,(\lambda ab.a)$.

iv) $(\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx)))\,(\lambda ab.b)$.

v) $(\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx)))\,(\lambda ab.ab)$.

* *Exercise 1.31* *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* $M_i \rightarrow^*_\beta M_{i+1}$, *or* $M_{i+1} \rightarrow^*_\beta M_i$.

Exercise 1.32 *Show that* $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ *is a fixed-point constructor for the Lambda Calculus, i.e. show that, for all terms* M,

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

Exercise 1.33 *Show that* $(\lambda xy.y(xxy))(\lambda xy.y(xxy))$ *is a fixed-point constructor for the Lambda Calculus.*

Exercise 1.34 *Show that all terms in normal form are in head-normal form.*

Exercise 1.35 *Show that, if* M *has a normal form, it is unique (hint: use the Church-Rosser property).*

* *Exercise 1.36* *i) If* $A \in \mathcal{A}\,(xM_1 \cdots M_n)$, $A \neq \bot$ *and* $A' \in \mathcal{A}\,(N)$, *then also* $AA' \in \mathcal{A}\,(xM_1 \cdots M_n N)$.

ii) If $A \in \mathcal{A}\,(Mz)$ *and* $z \notin fv\,(M)$, *then either* $A = \bot$, *or:*

- $A \equiv A'z$, $z \notin fv\,(A')$, *and* $A' \in \mathcal{A}\,(M)$, *or*
- $\lambda x.A \in \mathcal{A}\,(M)$.

* *Exercise 1.37* *i) If* $A \sqsubseteq M$ *and* $M \rightarrow^*_\beta N$, *then* $A \sqsubseteq N$.

ii) If $A \in \mathcal{A}\,(M)$ *and* $M \rightarrow^*_\beta N$, *then also* $A \in \mathcal{A}\,(N)$.

iii) If $A \in \mathcal{A}\,(N)$ *and* $M \rightarrow^*_\beta N$, *then also* $A \in \mathcal{A}\,(M)$.

* *Exercise 1.38* *If* $M_1 \sqsubseteq M$, *and* $M_2 \sqsubseteq M$, *then* $M_1 \sqcup M_2$ *is defined, and:*

$$M_1 \sqsubseteq M_1 \sqcup M_2, M_2 \sqsubseteq M_1 \sqcup M_2, \textit{ and } M_1 \sqcup M_2 \sqsubseteq M.$$

* *Exercise 1.39* *Verify that* Cond True $M\ N \rightarrow M$ *and* Cond False $M\ N \rightarrow N$.

* *Exercise 1.40* *Verify that* First $\langle M, N \rangle \rightarrow_\beta M$ *and* Second $\langle M, N \rangle \rightarrow_\beta N$.

* *Exercise 1.41* *Show that* Pred (Succ n) $\rightarrow_\beta n$; *can we show the same for* Succ (Pred n)?

2 The Curry type assignment system

In this section, we will present the basic notion of type assignment for the λ-calculus, as first studied by H.B. Curry in [22] (see also [23]). 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 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 Curry's 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 of type A and returning a result of type B.

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 it has type B.

Curry's system is formulated by a system of *derivation rules* that act as description of building stones that are used to build derivations; the two observations made 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 that given the assumptions in Γ, A is the type for the term M.

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:

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

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 $\Gamma \cup \{x{:}A\}$ where either $x{:}A \in \Gamma$ or x does not occur in Γ, and $x{:}A$ for $\emptyset, x{:}A$. We write $x \in \Gamma$ if there exists A such that $x{:}A \in \Gamma$, and $x \notin \Gamma$ if this is not the case.

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 $(A \rightarrow B) \rightarrow C \rightarrow D$ stands for $((A \rightarrow B) \rightarrow (C \rightarrow D))$.

We will now give the definition of Curry type assignment.

Definition 2.2 (cf. [22, 23]) *i*) *Curry type assignment* and *derivations* are defined by the following derivation rules (that define a natural deduction system).

$$(Ax): \frac{}{\Gamma, x{:}A \vdash_{\mathrm{C}} x : A} \qquad (\rightarrow I): \frac{\Gamma, x{:}A \vdash_{\mathrm{C}} M : B}{\Gamma \vdash_{\mathrm{C}} \lambda x.M : A \rightarrow B}\ (x \notin \Gamma) \qquad (\rightarrow E): \frac{\Gamma \vdash_{\mathrm{C}} M_1 : A \rightarrow B \quad \Gamma \vdash_{\mathrm{C}} M_2 : A}{\Gamma \vdash_{\mathrm{C}} M_1 M_2 : B}$$

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.

We will extend Barendregt's convention to judgements $\Gamma \vdash M : A$ (in all notions of type assignment we consider here) by demanding that in $\Gamma \vdash M : A$ no term variable bound in M can occur in Γ; this implies that the side-condition on rule $(\rightarrow I)$ can be omitted.

Example 2.3 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

$$\frac{\dfrac{}{\Gamma, x{:}A \rightarrow B \vdash_{\mathrm{C}} x : A \rightarrow B} \qquad \dfrac{}{\Gamma, x{:}A \vdash_{\mathrm{C}} x : A}}{\Gamma, x{:}? \vdash_{\mathrm{C}} xx : B}$$

for certain A and B. So we need to find a solution for $A \rightarrow B = A$, and this is impossible given our definition of types. For this reason, the term $\lambda x.xx$ is not typeable, and neither is $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. In fact, the procedure pp_{c} that tries to construct a type for terms as defined below (see Definition 2.14) will fail on xx.

Also, since $\bot$ is not mentioned in the type assignment rules, any approximant containing $\bot$ will not be typeable.

2.2 Subject Reduction

The following theorem 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 if 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 as an abstract interpretation of M.

Theorem 2.4 (Soundness) *If $\Gamma \vdash_{\mathrm{c}} M : A$ and $M \to_\beta^* N$, then $\Gamma \vdash_{\mathrm{c}} N : A$.*

Notice that this result states that if a derivation exists for the first result, one exist for the second. In the proof for the result, we will reason over the structure of the given derivation (the first), and show that a derivation exists for the second statement by constructing it.

Before coming to the proof of this result, first we illustrate it by the following:

Example 2.5 Suppose first that $\Gamma \vdash_{\mathrm{c}} (\lambda x.M)N : A$; since this is derived by $(\to E)$, there exists B such that $\Gamma \vdash_{\mathrm{c}} \lambda x.M : B\to A$ and $\Gamma \vdash_{\mathrm{c}} N : B$. Then $(\to I)$ 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 $\Gamma \vdash_{\mathrm{c}} N : B$, so the full derivation looks like the left-hand derivation below. 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 $\Gamma,x{:}B \vdash_{\mathrm{c}} x : B$ (consisting of just rule (Ax)) by the derivation for $\Gamma \vdash_{\mathrm{c}} N : B$, as in the right-hand derivation.

$$
\dfrac{
 \dfrac{\begin{array}{c}\overline{\Gamma,x{:}B \vdash x : B}\;(Ax)\\ \boxed{\mathcal{D}_1}\\ \Gamma,x{:}B \vdash M : A\end{array}}{\Gamma \vdash \lambda x.M : B\to A}\;(\to I)
 \qquad
 \begin{array}{c}\boxed{\mathcal{D}_2}\\ \Gamma \vdash N : B\end{array}
}{\Gamma \vdash (\lambda x.M)\,N : A}\;(\to E)
\qquad\qquad
\begin{array}{c}\boxed{\mathcal{D}_2}\\ \Gamma \vdash_{\mathrm{c}} N : B\\ \boxed{\mathcal{D}_1[N/x]}\\ \Gamma \vdash_{\mathrm{c}} M[N/x] : A\end{array}
$$

Notice that we then also need to systematically replace x by N throughout $\mathcal{D}_1$.

We will need the following result below:

Lemma 2.6 (*Free variables*): *If $\Gamma \vdash_{\mathrm{c}} M : A$ and $x \in fv(M)$, then there exists B such that $x{:}B \in \Gamma$.*

(*Weakening*): *If $\Gamma \vdash_{\mathrm{c}} M : A$, and Γ' is such that, for all $x{:}B \in \Gamma'$ either $x{:}B \in \Gamma$ or x does not occur free or bound in M, then $\Gamma' \vdash_{\mathrm{c}} M : A$.*

(*Thinning*): *If $\Gamma,x{:}B \vdash_{\mathrm{c}} M : A$ and $x \notin fv(M)$, then $\Gamma \vdash_{\mathrm{c}} M : A$.*

We will leave the proof of this result as Exercise 2.18. Notice for the first part that we assume that Γ has no statements on term-variables that occur bound in M.

Notice that the formulation of the second part (rather than just stating $\Gamma' \supseteq \Gamma$, as is normally done) is forced by the extension of Barendregt's convention to judgements.

In order to formally prove the Soundness theorem, we first prove a term substitution lemma.

Lemma 2.7 (Term Substitution) $\exists C\,(\,\Gamma,x{:}C \vdash_{\mathrm{c}} M : A \wedge \Gamma \vdash_{\mathrm{c}} N : C\,) \Rightarrow \Gamma \vdash_{\mathrm{c}} M[N/x] : A$.

Proof: By induction on the structure of derivations.

(Ax): Then either:

$(M \equiv x)$: $\exists C\,(\,\Gamma, x{:}C \vdash_{\mathrm{c}} x : A \wedge \Gamma \vdash_{\mathrm{c}} N : C\,) \;\Rightarrow\; (Ax) \;\; A = C \wedge \Gamma \vdash_{\mathrm{c}} N : C \;\Rightarrow$
$\Gamma \vdash_{\mathrm{c}} N : A \;\Rightarrow\; \Gamma \vdash_{\mathrm{c}} x[N/x] : A$

$(M \equiv y \neq x)$: $\exists C\,(\,\Gamma, x{:}C \vdash_{\mathrm{c}} y : A \wedge \Gamma \vdash_{\mathrm{c}} N : C\,) \;\Rightarrow\; y{:}A \in \Gamma \;\Rightarrow\; \Gamma \vdash_{\mathrm{c}} y : A$

$(\to I)$: Then $M \equiv \lambda y.P$, and

$$
\begin{array}{lll}
\exists C\,(\,\Gamma, x{:}C \vdash_{\mathrm{c}} \lambda y.P : A \wedge \Gamma \vdash_{\mathrm{c}} N : C\,) & \Rightarrow & (\to I) \\
\exists C, A', B'\,(\,\Gamma, x{:}C, y{:}A' \vdash_{\mathrm{c}} P : B' \wedge A = A' \to B' \wedge \Gamma \vdash_{\mathrm{c}} N : C\,) & \Rightarrow & (2.6) \\
\exists C, A', B'\,(\,\Gamma, x{:}C, y{:}A' \vdash_{\mathrm{c}} P : B' \wedge A = A' \to B' \wedge \Gamma, y{:}A' \vdash_{\mathrm{c}} N : C\,) & \Rightarrow & (IH) \\
\exists A', B'\,(\,\Gamma, y{:}A' \vdash_{\mathrm{c}} P[N/x] : B' \wedge A = A' \to B') & \Rightarrow & (\to I) \\
\Gamma \vdash_{\mathrm{c}} \lambda y.P[N/x] : A & = & \Gamma \vdash_{\mathrm{c}} (\lambda y.P)[N/x] : A
\end{array}
$$

$(\to E)$: Then $M \equiv PQ$, and

$$
\begin{array}{lll}
\exists C\,(\,\Gamma, x{:}C \vdash_{\mathrm{c}} PQ : A \wedge \Gamma \vdash_{\mathrm{c}} N : C\,) & \Rightarrow & (\to E) \\
\exists B\,(\,\Gamma, x{:}C \vdash_{\mathrm{c}} P : B \to A \wedge \Gamma, x{:}C \vdash_{\mathrm{c}} Q : B \wedge \Gamma \vdash_{\mathrm{c}} N : C\,) & \Rightarrow & (IH) \\
\exists B\,(\,\Gamma \vdash_{\mathrm{c}} P[N/x] : B \to A \wedge \Gamma \vdash_{\mathrm{c}} Q[N/x] : B\,) & \Rightarrow & (\to E) \\
\Gamma \vdash_{\mathrm{c}} P[N/x]Q[N/x] : A & = & \Gamma \vdash_{\mathrm{c}} (PQ)[N/x] : A \Box
\end{array}
$$

The proof for Theorem 2.4 then becomes:

Proof: By induction on the derivation showing $M \to_\beta^* N$, using Fig 1; we only show some of the cases.

- (β): Then $M \equiv (\lambda x.P)\,Q \to_\beta P[Q/x]$. We have to show that $\Gamma \vdash_{\mathrm{c}} (\lambda x.P)\,Q : A$ implies $\Gamma \vdash_{\mathrm{c}} P[Q/x] : A$. Notice that, if $\Gamma \vdash_{\mathrm{c}} (\lambda x.P)\,Q : A$, then, by $(\to E)$ and $(\to I)$, there exists C such that $\Gamma, x{:}C \vdash_{\mathrm{c}} P : A$ and $\Gamma \vdash_{\mathrm{c}} Q : C$. The result then follows from Lemma 2.7.
- $(\mathit{Appl\text{-}L})$: Then $M \equiv PQ$, $N \equiv RQ$, and $PQ \to_\beta RQ$ because $P \to_\beta R$. If $\Gamma \vdash_{\mathrm{c}} PQ : A$, then, by $(\to E)$ there exists C such that $\Gamma \vdash_{\mathrm{c}} P : C \to A$ and $\Gamma \vdash_{\mathrm{c}} Q : C$. By induction we have $\Gamma \vdash_{\mathrm{c}} R : C \to A$; then by $(\to E)$ we obtain $\Gamma \vdash_{\mathrm{c}} RQ : A$.
- $(\mathit{Inherit}_r)$: Then $M \to_\beta^* N$ because $M \to_\beta N$. Assume $\Gamma \vdash_{\mathrm{c}} M : A$; since $M \to_\beta N$, by induction we have $\Gamma \vdash_{\mathrm{c}} N : A$.
- (Trans_r): Then $M \to_\beta^* N$ because $M \to_\beta^* P$ and $P \to_\beta^* N$. Assume $\Gamma \vdash_{\mathrm{c}} M : A$; since $M \to_\beta^* P$, by induction we have that $\Gamma \vdash_{\mathrm{c}} P : A$; since $P \to_\beta^* N$ again by induction we also have $\Gamma \vdash_{\mathrm{c}} N : A$. $\Box$

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

Theorem 2.8 *If $\Gamma \vdash_{\mathrm{c}} M : A$ and $M \to_\eta N$, then $\Gamma \vdash_{\mathrm{c}} N : A$.*

Proof: By induction on the definition of $\to_\eta$, of which only the part $\lambda x.Mx \to_\eta M$, $x \notin \mathit{fv}(M)$ is shown; the other parts follow by straightforward induction. Assume $x \notin \mathit{fv}(M)$, then

$$
\begin{array}{lll}
\Gamma \vdash_{\mathrm{c}} \lambda x.Mx : A & \Rightarrow & (\to I) \\
\exists B, C\,(A = B \to C \wedge \Gamma, x{:}B \vdash_{\mathrm{c}} Mx : C\,) & \Rightarrow & (\to E) \\
\exists B, C, D\,(A = B \to C \wedge \Gamma, x{:}B \vdash_{\mathrm{c}} M : D \to C \wedge \Gamma, x{:}B \vdash_{\mathrm{c}} x : D\,) & \Rightarrow & (2.6) \\
\exists B, C, D\,(A = B \to C \wedge \Gamma \vdash_{\mathrm{c}} M : D \to C \wedge B = D) & \Rightarrow & \\
\exists B, C\,(A = B \to C \wedge \Gamma \vdash_{\mathrm{c}} M : B \to C\,) & \Rightarrow & \Gamma \vdash_{\mathrm{c}} M : A. \qquad \Box
\end{array}
$$

Example 2.9 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 2.4), i.e. *Subject Expansion*: If $\Gamma \vdash_{\mathrm{c}} M : A$ and $N \to_\beta^* M$, then $\Gamma \vdash_{\mathrm{c}} N : A$. The counter example is in Exercise 2.17: take $M = \lambda bc.c$, and $N = \lambda bc.(\lambda y.c)\,(bc)$, then it is easy to check that $N \to_\beta^* M$, $\emptyset \vdash_{\mathrm{c}} \lambda bc.c : A \to B \to B$, but not $\emptyset \vdash_{\mathrm{c}} \lambda bc.(\lambda y.c)\,(bc) : A \to B \to B$.

2.3 The principal type property

Principal type schemes for Curry's system were first defined in [32]. In that paper Hindley actually proved the existence of principal types for an object in Combinatory Logic [21] (see Definition 6.19), 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*) The *substitution* $(\varphi \mapsto C) : \mathcal{T}_C \rightarrowtail \mathcal{T}_C$, where φ is a type variable and $C \in \mathcal{T}_C$, is inductively defined by:

$$\begin{array}{lcll} (\varphi \mapsto C)\,\varphi & = & C & \\ (\varphi \mapsto C)\,\varphi' & = & \varphi' & (\varphi' \neq \varphi) \\ (\varphi \mapsto C)\ A{\rightarrow}B & = & ((\varphi \mapsto C)\ A) \rightarrow ((\varphi \mapsto C)\ B) & \end{array}$$

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

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

iv) $S\,\langle \Gamma\,;\ A \rangle = \langle S\,\Gamma\,;\ S\,A \rangle$.

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

vi) Id_S is the identity substitution that replaces all type variables by themselves.

So, for Curry's 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 $S\langle \Pi\,;\ P \rangle = \langle \Gamma\,;\ A \rangle$.

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 acts 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 2.11 (Soundness of substitution) *For every substitution S: if (we have a derivation for) $\Gamma \vdash_c M : A$, then (we can construct a derivation for) $S\,\Gamma \vdash_c M : S\,A$.*

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), $S\,\Gamma \vdash_c x : S\,A$.

$({\rightarrow}I)$: Then there are P, A, C such that $M \equiv \lambda x.P$, $A = C \rightarrow D$, and $\Gamma, x{:}C \vdash_c P : D$. Since this statement is derived in a sub-derivation, by induction we can assume that $S\,(\Gamma, x{:}C) \vdash_c P : S\,D$. Since $S\,(\Gamma, x{:}C) = S\,\Gamma, x{:}S\,C$, we also have $S\,\Gamma, x{:}S\,C \vdash_c P : S\,D$. So there is a derivation that shows this, to which we can apply rule $({\rightarrow}I)$, to obtain $S\,\Gamma \vdash_c \lambda x.P : S\,C \rightarrow S\,D$. Since $S\,C \rightarrow S\,D = S\,(C \rightarrow D) = S\,A$, by definition of substitutions, we get $S\,\Gamma \vdash_c \lambda x.P : S\,A$.

$({\rightarrow}E)$: Then there are P, Q, and B such that $M \equiv PQ$, $\Gamma \vdash_c P : B{\rightarrow}A$, and $\Gamma \vdash_c Q : B$. Since these two statements are derived in a sub-derivation, by induction we can assume both $S\,\Gamma \vdash_c P : S\,(B{\rightarrow}A)$ and $S\,\Gamma \vdash_c Q : S\,B$. Since $S\,(B{\rightarrow}A) = S\,B \rightarrow S\,A$ by definition of substitution, we also have $S\,\Gamma \vdash_c P : S\,B \rightarrow S\,A$, and we can apply rule $({\rightarrow}E)$ to obtain $S\,\Gamma \vdash_c PQ : S\,A$. □

Principal types for λ-terms are defined using the notion of unification of types that was defined by Robinson in [49]. Robinson's unification, also used in logic programming, is a procedure on types (or logical formulae) 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 2.12 *i*) (ROBINSON'S UNIFICATION) Unification of Curry types is defined by:

$$
\begin{array}{lllcl}
\textit{unify} & \varphi & \varphi & = & (\varphi \mapsto \varphi) \\
\textit{unify} & \varphi & B & = & (\varphi \mapsto B) \quad (\varphi \textit{ does not occur in } B) \\
\textit{unify} & A & \varphi & = & \textit{unify } \varphi \; A \\
\textit{unify} & (A \to B) & (C \to D) & = & S_2 \circ S_1 \\
 & & & & \text{where } S_1 \; = \; \textit{unify } A \; C \\
 & & & & \qquad\quad\; S_2 \; = \; \textit{unify } (S_1 B) \; (S_1 D)
\end{array}
$$

ii) The operation *UnifyContexts* generalises *unify* to contexts:

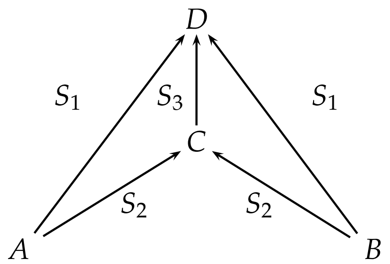
$$
\begin{array}{lllcl}
\textit{UnifyContexts} & (\Gamma_1, x{:}A) & (\Gamma_2, x{:}B) & = & S_2 \circ S_1, \\
 & & & & \text{where } S_1 \; = \; \textit{unify } A \; B \\
 & & & & \qquad\quad\; S_2 \; = \; \textit{UnifyContexts } (S_1 \Gamma_1) \; (S_1 \Gamma_2) \\
\textit{UnifyContexts} & (\Gamma_1, x{:}A) & \Gamma_2 & = & \textit{UnifyContexts}\, \Gamma_1 \, \Gamma_2, \text{if } x \text{does not occur in } \Gamma_2. \\
\textit{UnifyContexts} & \emptyset & \Gamma_2 & = & \textit{Id}_S.
\end{array}
$$

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* C 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 C by substitution.

Proposition 2.13 ([49]) *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*

$$
\begin{array}{rcl}
S_2 & = & \textit{unify } A \; B \qquad \textit{and} \\
S_1 A = S_3 \circ S_2 A & = & S_3 \circ S_2 B = S_1 B.
\end{array}
$$

which corresponds to the diagram on the right. We say that S_3 extends *S_2.*



Unification is associative and commutative: we can show that, for all types A, B, and C

$$
\begin{array}{rcl}
\textit{unify } ((\textit{unify } A \; B) \; A) \; C & = & \textit{unify } ((\textit{unify } A \; C) \; A) \; B \\
 & = & \textit{unify } ((\textit{unify } B \; C) \; B) \; A \\
 & = & \textit{unify } A \; ((\textit{unify } B \; C) \; B) \\
 & = & \textit{unify } B \; ((\textit{unify } A \; C) \; A) \\
 & = & \textit{unify } C \; ((\textit{unify } A \; B) \; A)
\end{array}
$$

which justifies a 'higher-order notation'; we write *unify* $A \; B \; C \; \ldots$ for any number of types. We will use the same higher-order notation for *UnifyContexts*.

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

Definition 2.14 We define for every term M the *(Curry) principal pair* by defining the notion $pp_{\mathrm{C}}\, M = \langle \Pi ;\, P \rangle$ inductively by:

i) For all x, φ: $pp_{\mathrm{C}}\, x = \langle x{:}\varphi ;\, \varphi \rangle$.

ii) If $pp_{\mathrm{C}}\, M = \langle \Pi ;\, P \rangle$, then:

a) If $x \in fv(M)$, then (by construction) there is a A such that $x{:}A \in \Pi$, and $pp_{\mathrm{C}}\, \lambda x.M =$

$$\begin{array}{lll} pp_{\mathrm{C}}\, x & = & \langle x{:}\varphi;\ \varphi\rangle \\ \quad \text{where} & & \varphi \textit{ is fresh} \\ pp_{\mathrm{C}}\, \lambda x.M & = & \begin{cases} \langle \Pi';\ A{\to}P\rangle & (\Pi = \Pi', x{:}A) \\ \langle \Pi;\ \varphi{\to}P\rangle & (x \notin \Pi) \end{cases} \\ \quad \text{where} & & \langle \Pi;\ P\rangle \ = \ pp_{\mathrm{C}}\, M \\ & & \varphi \quad \textit{is} \quad \textit{fresh} \end{array} \qquad \begin{array}{lll} pp_{\mathrm{C}}\, MN & = & S_2{\circ}S_1\, \langle \Pi_1 \cup \Pi_2;\ \varphi\rangle \\ \quad \text{where} & & \langle \Pi_1;\ P_1\rangle \ = \ pp_{\mathrm{C}}\, M \\ & & \langle \Pi_2;\ P_2\rangle \ = \ pp_{\mathrm{C}}\, N \\ & & S_1 \ = \ \textit{unify}\ P_1\ P_2{\to}\varphi \\ & & S_2 \ = \ \textit{UnifyContexts}\ (S_1\,\Pi_1)\ (S_1\,\Pi_2) \\ & & \varphi \quad \textit{is} \quad \textit{fresh} \end{array}$$

Figure 2. The principal pair algorithm for Λ in Curry's system

$\langle \Pi \backslash x;\ A{\to}P\rangle$.

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

iii) If $pp_{\mathrm{C}}\, M_1 = \langle \Pi_1;\ P_1\rangle$ and $pp_{\mathrm{C}}\, M_2 = \langle \Pi_2;\ P_2\rangle$ (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

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

then $pp_{\mathrm{C}}\,(M_1M_2) = S_2{\circ}S_1\,\langle \Pi_1 \cup \Pi_2;\ \varphi\rangle$.

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

This definition in fact gives an algorithm that finds the principal pair for λ-terms, if it exists. 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; in Haskell this problem can be dealt with using 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 2.15 The principal pair algorithm for Curry's system is given in Figure 2.

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.

The proof that the procedure 'pp_{C}' indeed returns 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 2.13 is needed.

Theorem 2.16 (Completeness of substitution.) *If $\Gamma \vdash_{\mathrm{c}} M : A$, then there are context Π, type P and a substitution S such that: $pp_{\mathrm{C}}\, M = \langle \Pi;\ P\rangle$, and both $S\,\Pi \subseteq \Gamma$ and $S\,P = A$.*

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

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

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

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

b) otherwise $pp_{\mathrm{C}}\,(\lambda x.N) = \langle \Pi';\ \varphi{\to}P'\rangle$, 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 $\Gamma \vdash_{\mathrm{C}} M_1 : B \to A$ and $\Gamma \vdash_{\mathrm{C}} M_2 : B$. By induction, there are S_1, S_2, $\langle \Pi_1 ; P_1 \rangle = pp_{\mathrm{C}}\, M_1$ and $\langle \Pi_2 ; P_2 \rangle = pp_{\mathrm{C}}\, M_2$ (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 $S_2 P_2 = B$. Notice that S_1, S_2 do not interfere. Let φ be a type variable that does not occur in any of the pairs $\langle \Pi_i ; P_i \rangle$, and

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

then, by the definition above, $pp_{\mathrm{C}}\, M_1 M_2 = S_c \circ S_u \langle \Pi_1 \cup \Pi_2 ; \varphi \rangle$.

We will now argue that $pp_{\mathrm{C}}\, M_1 M_2$ is successful: since this can only fail on unification (of P_1 and $P_2 \to \varphi$, or in the unification of the contexts), we need to argue that these unification are successful. Take $S_3 = S_2 \circ S_1 \circ (\varphi \mapsto A)$, then

$$\begin{array}{rcl} S_3\ P_1 & = & B \to A, \text{ and} \\ S_3\ P_2 \to \varphi & = & B \to A. \end{array}$$

so P_1 and $P_2 \to \varphi$ have a common instance $B \to A$, and by Proposition 2.13, S_u exists.

We now need to argue that a substitution S exists such that $S(S_c \circ S_u (\Pi_1 \cup \Pi_2)) \subseteq \Gamma$. Notice that we have

$$\begin{array}{rcl} S_3 \Pi_1 & \subseteq & \Gamma, \text{ and} \\ S_3 \Pi_2 & \subseteq & \Gamma \end{array}$$

since Π_1 and Π_2 share no type-variables. Since Γ is a context, each term variable has only one type, and therefore S_3 is a unifier for Π_1 and Π_2, so we know that S_4 exists that extends the substitution that unifies the contexts, even after being changed with S_u, so such that

$$\begin{array}{rcl} S_4 (S_u \Pi_1) & \subseteq & \Gamma, \text{ and} \\ S_4 (S_u \Pi_2) & \subseteq & \Gamma. \end{array}$$

So S_4 also unifies $S_u \Pi_1$ and $S_u \Pi_2$, so by Proposition 2.13 there exists a substitution S_5 such that $S_4 = S_5 \circ S_c \circ S_u$. Take $S = S_5$. $\square$

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. 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 [32], and Milner's seminal paper [43] 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 normalisable. 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.

Exercises

Exercise 2.17 Verify the following results:

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

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

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

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

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

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

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

Exercise 2.18 Verify the following results:

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

ii) If $\Gamma \vdash_{c} M : A$ *, then* $\{\, y{:}D \in \Gamma \mid y \in fv(M) \,\} \vdash_{c} M : A$.

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

so show Lemma 5.5.

* *Exercise 2.19 Show that, for all substitutions* S *and types* A *and* B*,* $S(A \rightarrow B) = SA \rightarrow SB$.

* *Exercise 2.20 Show Lemma 2.11: for every substitution* S*: if* $\Gamma \vdash_{c} M : A$ *, then* $S\,\Gamma \vdash_{c} M : S\,A$.

* *Exercise 2.21 Show that, if* $pp_{c}\, M = \langle \Pi\,;\, P \rangle$*, then* $\Pi \vdash_{c} M : P$ *. You'll need Lemma 2.6 and 2.11 here.*

Exercise 2.22 Give, with derivations, the principal types for the terms $\lambda xyz.xz(yz)$*,* $\lambda xy.x$*,* $\lambda xy.xy$*, and* $\lambda x.x$ *in Curry's system.*

Using these types, calculate the principal types of the terms $(\lambda xyz.xz(yz))(\lambda xy.x)(\lambda x.x)$ *and* $(\lambda xy.xy)(\lambda x.x)$ *in the Curry system; specify the main calls to unify.*

What can you conclude about the sets of types assignable to both terms?

Exercise 2.23 Is it possible to type $(\lambda xyz.xz(yz))(\lambda x.x)(\lambda x.x)((\lambda xyz.xz(yz))(\lambda x.x)(\lambda x.x))$ *in Curry's system? Motivate your answer.*

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 that are *polymorphic*, i.e. 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:

$$\begin{array}{rcl} I\,x & = & x \\ I\,I & & \end{array}$$

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 [43] (see also Section 5). 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.

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

This is used to find types for terms that contain names. Each occurrence of a name *name* in the program can be regarded as an abbreviation of the right-hand side in its definition, and therefore the type associated to the occurrence should be an instance of the principal type of the right-hand side term; we associate *name* to that term, and store the principal type of the right-hand side with *name* in an *environment*: the type of *name* is called *generic*, and the term defined by *name* is called a *polymorphic function*. In fact, we would like to model that each call to *name* 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* (the *main*) which may contain *calls*.

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

$$\begin{array}{lcll} M,N & ::= & x \mid name \mid \lambda x.N \mid MN & \\ name & ::= & \textit{'string of characters'} & \\ Defs & ::= & Defs\,;name = M \mid \epsilon & (M \textit{ is closed and name-free}) \\ Program & ::= & Defs : M & \end{array}$$

Like before, redundant brackets will be omitted.

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

$$\frac{}{(\lambda x.M)\,N \to M[N/x]} \qquad \frac{}{name \to M}\;(name = M \in Defs)$$

$$\frac{M \to N}{\lambda x.M \to \lambda x.N} \qquad \frac{M \to N}{MP \to NP} \qquad \frac{M \to N}{PM \to PN} \qquad \frac{M \to N}{M \to^{*} N} \qquad \frac{}{M \to^{*} M} \qquad \frac{M \to^{*} N \quad N \to^{*} P}{M \to^{*} P}$$

and reduction on Programs is defined by:

$$\frac{M \to N}{Defs : M \to Defs : N}$$

We have put the restriction that in $name = M$, M is a closed λ-term. This forms, in fact, a sanity criterion; similarly, free variables are not allowed in method bodies in Java, or on the right or term rewriting rules (see Section 6). We could deal with open terms as well, but this would complicate the definitions and results below unnecessarily.

Notice that, in the body of definitions, calls to other definitions are not allowed, so names are just abbreviations for closed, pure λ-terms; we could even restrict bodies of definitions such that no redexes occur there, but that is not necessary for the present purpose. Moreover, it is possible that a name is used that is not in the list of definitions; this term is then irreducible.

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.

It is possible to give a natural interpretation of Λ^{N} in Λ, which is a generalisation of the one we saw in the previous section.

Definition 3.2 $\langle\cdot\rangle_\lambda : \Lambda^{\mathrm{N}} \to \Lambda$ is defined by:

$$\begin{array}{lcl}
\langle x\rangle_\lambda & = & x, \\
\langle name\rangle_\lambda & = & \begin{cases} \langle M\rangle_\lambda & ((name = M) \textit{ appears in the list of definitions.}) \\ \textit{undefined} & (\textit{otherwise}) \end{cases} \\
\langle \lambda x.N\rangle_\lambda & = & \lambda x.\langle N\rangle_\lambda \\
\langle PQ\rangle_\lambda & = & \langle P\rangle_\lambda \langle Q\rangle_\lambda
\end{array}$$

It is now straightforward to check that reduction is preserved by this interpretation (see Exercise 3.7).

3.2 Type assignment for Λ^{N}

In this section we will develop a notion of type assignment on programs in Λ^{N}. In short, we will 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 *name*, we will force the type for *name* 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 the λ-calculus, and has been studied extensively in the past, e.g. in [32, 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. 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, i.e. correct.

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, and M a λ-term, perhaps containing names. We also use the notion $\mathcal{E} \vdash \mathit{Defs} : \Diamond$; 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 3.3 (Type Assignment for Λ^{N}) *i*) An *environment* $\mathcal{E}$ is a mapping from *names* to types; similar to contexts, $\mathcal{E}, name{:}A$ is the environment defined as $\mathcal{E} \cup \{name{:}A\}$ where either $name{:}A \in \mathcal{E}$ or *name* 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 (*Call*).

$$(Ax): \frac{}{\Gamma, x{:}A;\mathcal{E} \vdash x : A} \qquad (\to I): \frac{\Gamma, x{:}A;\mathcal{E} \vdash N : B}{\Gamma;\mathcal{E} \vdash \lambda x.N : A \to B} \qquad (\to E): \frac{\Gamma;\mathcal{E} \vdash P : A \to B \quad \Gamma;\mathcal{E} \vdash Q : A}{\Gamma;\mathcal{E} \vdash PQ : B}$$

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

$$(\mathit{Call}): \frac{}{\Gamma;\mathcal{E}, name{:}A \vdash name : S\,A} \qquad (\mathit{Program}): \frac{\mathcal{E} \vdash \mathit{Defs} : \Diamond \qquad \Gamma;\mathcal{E} \vdash M : A}{\Gamma;\mathcal{E} \vdash \mathit{Defs} : M : A}$$

Notice that, in rule (*Defs*), we insist that M is closed by demanding it be typed using an empty context. Moreover, the rule extends the environment by adding the pair of *name* and the type A found for M; this does not preclude that $name{:}A$ already occurs in $\mathcal{E}$.

For this notion of type assignment, we can easily show the subject reduction result (see Exercise 3.8). Notice 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.

$$
\begin{array}{lcl}
pp_{\Lambda^N}\ x\ \mathcal{E} & = & \langle x{:}\varphi;\ \varphi\rangle \\
 & & \quad \textit{where } \varphi \textit{ is fresh} \\
pp_{\Lambda^N}\ \textit{name}\ \mathcal{E} & = & \langle \emptyset;\ \textit{FreshInstance}\,(\mathcal{E}\textit{name})\rangle \\
pp_{\Lambda^N}\ (\lambda x.M)\ \mathcal{E} & = & \begin{cases} \langle \Pi';\ A{\to}P\rangle & (\Pi = \Pi', x{:}A) \\ \langle \Pi;\ \varphi{\to}P\rangle & (x \notin \Pi) \end{cases} \\
 & & \quad \begin{array}{lrcl} \textit{where} & \langle \Pi;\ P\rangle & = & pp_{\Lambda^N}\ M\ \mathcal{E} \\ & \varphi & \textit{is} & \textit{fresh} \end{array} \\
pp_{\Lambda^N}\ (MN)\ \mathcal{E} & = & S_2 \circ S_1\, \langle \Pi_1 \cup \Pi_2;\ \varphi\rangle \\
 & & \quad \begin{array}{lrcl} \textit{where} & \langle \Pi_1;\ P_1\rangle & = & pp_{\Lambda^N}\ M\ \mathcal{E} \\ & \langle \Pi_2;\ P_2\rangle & = & pp_{\Lambda^N}\ N\ \mathcal{E} \\ & S_1 & = & \textit{unify}\ P_1\ P_2{\to}\varphi \\ & S_2 & = & \textit{UnifyContexts}\ (S_1\,\Pi_1)\ (S_1\,\Pi_2) \\ & \varphi & \textit{is} & \textit{fresh} \end{array} \\
 & & \\
\textit{BuildEnv}\ (\textit{Defs}; \textit{name} = M) & = & (\textit{BuildEnv Defs}), \textit{name}{:}A \\
 & & \quad \textit{where } \langle \emptyset;\ A\rangle = pp_{\Lambda^N}\ M\ \emptyset \\
\textit{BuildEnv}\ \epsilon & = & \{\} \\
 & & \\
pp_{\Lambda^N}\ (\textit{Defs}; M) & = & pp_{\Lambda^N}\ M\ \mathcal{E} \\
 & & \quad \textit{where } \mathcal{E} \ = \ \textit{BuildEnv Defs}
\end{array}
$$

Figure 3. Principal contexts and types for Λ^{N}

Example 3.4 We can now derive:

$$
\dfrac{
\dfrac{
\dfrac{}{\emptyset \vdash \epsilon : \Diamond}\,(\epsilon)
\quad
\dfrac{\dfrac{}{x{:}\varphi;\emptyset \vdash x : \varphi}\,(Ax)}{\emptyset;\emptyset \vdash_{\mathrm{c}} \lambda x.x : \varphi{\to}\varphi}\,({\to}I)
}{I{:}\varphi{\to}\varphi \vdash I = \lambda x.x : \Diamond}\,(\textit{Defs})
\qquad
\dfrac{
\begin{array}{c}\dfrac{}{\emptyset;I{:}\varphi{\to}\varphi \vdash I : (A{\to}A) \to A{\to}A}\,(\textit{Call}) \\ \vdots \end{array}
\quad
\dfrac{}{\emptyset;I{:}\varphi{\to}\varphi \vdash I : A{\to}A}\,(\textit{Call})
}{\emptyset;I{:}\varphi{\to}\varphi \vdash II : A{\to}A}\,({\to}E)
}{\emptyset;I{:}\varphi{\to}\varphi \vdash I = \lambda x.x : II : A{\to}A}\,(\textit{Program})
$$

We will now give an algorithm that, using an environment, checks if the term in a program in Λ^{N} can be typed. As far as variables, abstraction, and applications are concerned, the algorithm follows the approach of Definition 2.15. 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 *name* 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, by calling the substitution *FreshInstance* which replaces all type variables by fresh ones, and uses this instance when trying to find a type for the term; since the algorithm calls unification which gets applied to types, the type might change of course, depending on the structure of the term surrounding the call *name*. Would we take the type stored in the environment directly, 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 *name* do not affect the type already found for another. Moreover, this way the types actually used for *name* will always be substitution instances of the principal type that is associated to *name* in the environment.

Definition 3.5 The algorithm that calculates the types for a program in Λ^{N} is defined in Figure 3.

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 in the environment.

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.

We leave the soundness of this algorithm as an exercise.

Exercises

Exercise 3.6 Consider the Λ^N program

$$\begin{array}{lcll} \mathsf{I} & = & \lambda x.x & ; \\ \mathsf{B} & = & \lambda xyz.x(yz) & ; \\ \mathsf{K} & = & \lambda xy.x & ; \\ \mathsf{S} & = & \lambda xyz.xz(yz) & : \\ \multicolumn{4}{l}{\mathsf{S}\,(\mathsf{B}\,\mathsf{I})\,(\mathsf{K}\,\mathsf{I})} \end{array}$$

i) *Give a suitable environment for this program, and give a type-derivation for the final term.*
ii) *Reduce this term to normal form.*
iii) *Show that the type found is a valid type for the normal form as well.*

* *Exercise 3.7 Prove that, if $t \to t'$ in Λ^N, then and $\langle t \rangle_\lambda$ is defined, then $\langle t \rangle_\lambda \to_\beta^* \langle t' \rangle_\lambda$.*

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

4 Dealing with recursion

In this section, we will focus on a type assignment system for a simple applicative language called Λ^{NR} that is in fact Λ^N with recursion.

4.1 The language Λ^{NR}

First, we define 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 programs in Λ^{NR} is defined by:

$$\begin{array}{lcl} \mathit{name} & ::= & \text{'string of characters'} \\ M,N & ::= & x \mid \mathit{name} \mid \lambda x.M \mid MN \\ \mathit{Defs} & ::= & \mathit{Defs}\,;(\mathit{name} = M) \mid \mathit{Defs}\,;(\mathsf{rec}\ \mathit{name} = M) \mid \epsilon \quad (M \textit{ is closed}) \\ \mathit{Program} & ::= & \mathit{Defs} : M \end{array}$$

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

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.

Example 4.2

$$\begin{array}{lcll} \mathsf{S} & = & \lambda xyz.xz(yz) & ; \\ \mathsf{K} & = & \lambda xy.x & ; \\ \mathsf{I} & = & \mathsf{SKK} & ; \\ \mathsf{rec}\ \mathsf{Y} & = & \lambda m.m(\mathsf{Y}m) & : \\ \multicolumn{4}{l}{\mathsf{Y}\,\mathsf{I}} \end{array}$$

4.2 Expressing recursion in the Lambda Calculus

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

$$\begin{array}{lcll} \mathsf{Fac}\ n & = & 1 & (n=0) \\ \mathsf{Fac}\ n & = & n \times (\mathsf{Fac}\ n{-}1) & (\mathit{otherwise}) \end{array}$$

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

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

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:

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

To see that this indeed does what we intend, consider:

$$\begin{array}{lcl} \mathsf{Fac}\ 4 & = & \mathbf{Y}(\lambda f.\lambda n.\mathsf{Cond}\ (n=0)\ 1\ (n \times (f\ (n-1))))\ 4 \\ & = & (\lambda n.\mathsf{Cond}\ (n=0)\ 1\ (n \times (\mathsf{Fac}\ (n-1))))\ 4 \\ & \to & \mathsf{Cond}\ (4=0)\ 1\ (4 \times (\mathsf{Fac}\ 3)) \\ & = & 4 \times (\mathsf{Fac}\ 3) \\ & \to & 4 \times (\mathsf{Cond}\ (3=0)\ 1\ (3 \times (\mathsf{Fac}\ 2))) \\ & \to & 4 \times (3 \times \mathsf{Fac}\ 2) \\ & \to & 4 \times (3 \times (2 \times (1 \times (\mathsf{Cond}\ (0=0)\ 1\ (0 \times (\mathsf{Fac}\ {-1})))))) \\ & \to & 4 \times (3 \times (2 \times (1 \times 1))) \ = \ 4! \end{array}$$

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 the example above. We know that $\mathsf{num}{\to}\mathsf{num}$ should be the type for Fac, and that

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

so $\mathsf{num}{\to}\mathsf{num}$ should also be the type for

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

Notice that, by checking its context, the occurrence of Fac in this term has type num→num as well. Therefore,

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

should have type $(\mathsf{num}{\rightarrow}\mathsf{num})\rightarrow\mathsf{num}{\rightarrow}\mathsf{num}$. 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

$$((\mathsf{num}{\rightarrow}\mathsf{num})\rightarrow\mathsf{num}{\rightarrow}\mathsf{num})\rightarrow\mathsf{num}{\rightarrow}\mathsf{num}.$$

Therefore, for Λ^{NR} it suffices, when typing a recursive definition, to demand that the recursive calls have exactly the same type as the whole body of the recursive definition: $\Gamma;\mathcal{E},f{:}A\vdash C[f]:A$.

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 $(A{\rightarrow}A){\rightarrow}A$, for all A. So when enforcing typeable recursion for the λ-calculus, we would need to extend the language Λ by adding a constant term

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

add the reduction rule

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

and add the type assignment rule

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

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

When we will discuss Milner's ML in the next section, we will add a new *language construct*, i.e. a second kind of abstraction, *fixg.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}) *i*) An *environment* $\mathcal{E}$ is a mapping from *names* and *recursive names* to types:

$$\mathcal{E}\ ::=\ \epsilon\mid\mathcal{E},\mathit{name}{:}A\mid\mathcal{E},\mathsf{rec}\ \mathit{name}{:}A$$

where as before we assume that when writing $\mathcal{E},\mathit{name}{:}A$, we have that $\mathit{name}{:}A\in\mathcal{E}$ or *name* does not occur in $\mathcal{E}$, and similar for rec *name*.

ii) Type assignment for programs in Λ^{NR} is defined through the following rules:

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

$$(\mathit{Call}):\ \frac{}{\Gamma;\mathcal{E},\mathit{name}{:}A\vdash\mathit{name}:S\,A}\qquad(\mathit{Def}):\ \frac{\mathcal{E}\vdash\mathit{Defs}:\Diamond\qquad\phi;\mathcal{E}\vdash M:A}{\mathcal{E},\mathit{name}{:}A\vdash\mathit{Defs}\,;\mathit{name}=M:\Diamond}$$

$$(\mathit{Rec\ Call}):\ \frac{}{\Gamma;\mathcal{E},\mathsf{rec}\ \mathit{name}{:}A\vdash\mathit{name}:A}\qquad(\mathit{Rec\ Def}):\ \frac{\mathcal{E}\vdash\mathit{Defs}:\Diamond\quad\phi;\mathcal{E},\mathsf{rec}\ \mathit{name}{:}A\vdash M:A}{\mathcal{E},\mathit{name}{:}A\vdash\mathit{Defs}\,;\mathsf{rec}\ \mathit{name}=M:\Diamond}$$

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

Notice that the main difference between this and the notion we defined for Λ^{N} lies in rule (*Def*); 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, in rule (*Rec Def*) we insist on $\phi;\mathcal{E},\mathsf{rec}\ \mathit{name}{:}A\vdash M:A$ as a premise, not just $\phi;\mathcal{E}\vdash M:A$; this is paired with the presence of

$$
\begin{array}{lcl}
pp_{\Lambda^{\mathsf{NR}}}\ x\ \mathcal{E} & = & \langle x{:}\varphi\,;\,\varphi\,;\,\mathcal{E}\rangle \\
 & & \textit{where } \varphi \textit{ is fresh} \\
pp_{\Lambda^{\mathsf{NR}}}\ \textit{name}\ \mathcal{E} & = & \begin{cases} \langle \phi\,;\,A\,;\,\mathcal{E}\rangle & (\mathsf{rec}\ \textit{name}{:}A \in \mathcal{E}) \\ \langle \phi\,;\,\textit{FreshInstance}\,A\,;\,\mathcal{E}\rangle & (\textit{name}{:}A \in \mathcal{E}) \end{cases} \\
pp_{\Lambda^{N}}\ \lambda x.M\ \mathcal{E} & = & \begin{cases} \langle \Pi'\,;\,A{\to}P\,;\,\mathcal{E}'\rangle & (\Pi = \Pi', x{:}A) \\ \langle \Pi\,;\,\varphi{\to}P\,;\,\mathcal{E}'\rangle & (x \notin \Pi) \end{cases} \\
 & & \textit{where } \langle \Pi\,;\,P\,;\,\mathcal{E}'\rangle \ = \ pp_{\Lambda^{N}}\ M\ \mathcal{E} \\
 & & \qquad\quad \varphi \textit{ is fresh} \\
pp_{\Lambda^{N}}\ MN\ \mathcal{E} & = & S_2{\circ}S_1\,\langle \Pi_1 \cup \Pi_2\,;\,\varphi, \mathcal{E}''\rangle \\
 & & \textit{where } \begin{array}[t]{lcl} \langle \Pi_1\,;\,P_1\,;\,\mathcal{E}'\rangle & = & pp_{\Lambda^{N}}\ M\ \mathcal{E} \\ \langle \Pi_2\,;\,P_2\,;\,\mathcal{E}''\rangle & = & pp_{\Lambda^{N}}\ N\ \mathcal{E}' \\ S_1 & = & \textit{unify}\ P_1\ P_2{\to}\varphi \\ S_2 & = & \textit{UnifyContexts}\ (S_1\,\Pi_1)\ (S_1\,\Pi_2) \\ \varphi \textit{ is fresh} & & \end{array} \\
 & & \\
\textit{BuildEnv}\ (\textit{Defs}\,;\,\textit{name} = M)\ \mathcal{E} & = & (\textit{BuildEnv Defs}\ \mathcal{E}), \textit{name}{:}A \\
 & & \textit{where } \langle \phi\,;\,A\,;\,\mathcal{E}\rangle = pp_{\Lambda^{N}}\ M\ \mathcal{E} \\
\textit{BuildEnv}\ (\textit{Defs}\,;\,\mathsf{rec}\ \textit{name} = M)\ \mathcal{E} & = & (\textit{BuildEnv Defs}\ \mathcal{E}), \textit{name}{:}S\,A \\
 & & \textit{where } \begin{array}[t]{lcl} \langle \phi\,;\,A\,;\,\mathcal{E}'\rangle & = & pp_{\Lambda^{\mathsf{NR}}}\ M\ (\mathcal{E}, \mathsf{rec}\ \textit{name}{:}\varphi) \\ S & = & \textit{unify}\ A\ B \\ \mathsf{rec}\ \textit{name}{:}B & \in & \mathcal{E}' \\ \varphi \textit{ is fresh} & & \end{array} \\
\textit{BuildEnv}\ \epsilon\ \mathcal{E} & = & \mathcal{E} \\
 & & \\
pp_{\Lambda^{N}}\ (\textit{Defs};\,M) & = & pp_{\Lambda^{N}}\ M\ \mathcal{E} \\
 & & \textit{where } \mathcal{E} \ = \ \textit{BuildEnv Defs}\ \phi
\end{array}
$$

Figure 4. Principal environments and types for Λ^{NR}

rule (*Rec Call*), which enforces that all recursive calls are typed with exactly the environment type.

When looking to type a program using an algorithm, we need a type for every name we encounter in the environment and need to have dealt with definitions before their use. With recursive definitions, the latter creates an obvious problem, which we solve 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 4.6 The algorithm that calculates the types for a program in Λ^{NR} is defined in Figure 4.

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.

Exercises

Exercise 4.7 Take P to be the program in Example 4.2. Find an environment $\mathcal{E}$ such that $\phi;\mathcal{E} \vdash P : \varphi$, and build the derivation that justifies this judgement. Show that the final term reduces to itself.

Exercise 4.8 Consider the Λ^{NR} program

$$\begin{array}{lcl} \mathsf{E} & = & \lambda xy.xy\,; \\ \mathsf{S} & = & \lambda xyz.xz(yz)\,; \\ \mathsf{rec}\,\mathsf{Y} & = & \lambda m.m(\mathsf{Y}m) : \\ \mathsf{Y}\,(\mathsf{S}\,(\mathsf{E}\,\mathsf{E})) & & \end{array}$$

Give a suitable environment for this program.

Is the program typeable? If so, give a derivation; if not, argue why.

5 Milner's ML

In [43], a formal type discipline was presented for polymorphic procedures in a simple programming language called $\mathcal{L}_{\textsc{ml}}$, designed to express that certain procedures work well on objects of a wide variety. This kind of procedure is called *(shallow) polymorphic*, and it is essential to obtain enough flexibility in programming.

$\mathcal{L}_{\textsc{ml}}$ is based on the λ-calculus, 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 [24] 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 [43] – 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 [45], 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 [53] and Haskell [33]. 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 [25], and not as in [24, 43], because the former presentation is more detailed and clearer.

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

$$E \;::=\; x \mid c \mid \lambda x.E \mid E_1E_2 \mid \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \mid \mathsf{fix}\,g.E$$

We consider x bound over E_2 in $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$, and g bound over E in $\mathsf{fix}\,g.E$.

ii) The notion of term substitution over ML-terms is the natural extension of the similar notion for λ-terms of Definition 1.6:

$$\begin{array}{rcll} x\,[E/x] & = & E & \\ y\,[E/x] & = & y & (y \neq x) \\ (\lambda y.E')\,[E/x] & = & \lambda y.(E'\,[E/x]) & \end{array} \qquad \begin{array}{rcl} (E_1E_2)\,[E/x] & = & E_1\,[E/x]\,E_2\,[E/x] \\ (\mathsf{let}\ y = E_1\ \mathsf{in}\ E_2)\,[E/x] & = & \mathsf{let}\ y = E_1\,[E/x]\ \mathsf{in}\ E_2\,[E/x] \\ (\mathsf{fix}\,g.E')\,[E/x] & = & \mathsf{fix}\,g.E'\,[E/x] \end{array}$$

iii) The notion of reduction on $\mathcal{L}_{\textsc{ml}}$, $\rightarrow_{\textsc{ml}}$, is defined as $\rightarrow_\beta$, extended by:

$$\begin{array}{rcl} \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 & \rightarrow_{\textsc{ml}} & E_2\,[E_1/x] \\ \mathsf{fix}\,g.E & \rightarrow_{\textsc{ml}} & E\,[(\mathsf{fix}\,g.E)/g] \end{array}$$

and the additional contextual rules:

$$E \rightarrow_{\text{ML}} E' \;\Rightarrow\; \begin{cases} \mathsf{let}\ x = E\ \mathsf{in}\ E_2 & \rightarrow_{\text{ML}}\ \mathsf{let}\ x = E'\ \mathsf{in}\ E_2 \\ \mathsf{let}\ x = E_1\ \mathsf{in}\ E & \rightarrow_{\text{ML}}\ \mathsf{let}\ x = E_1\ \mathsf{in}\ E' \\ \mathsf{fix} g.E & \rightarrow_{\text{ML}}\ \mathsf{fix} g.E' \end{cases}$$

Here c is a term constant, like a number, character, or operator. As before, we will economise on brackets.

With this extended notion of reduction, the terms ($\mathsf{let}\ x = E_2\ \mathsf{in}\ E_1$) 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 ($\mathsf{let}\ x = E_2\ \mathsf{in}\ E_1$) 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 ($\mathsf{let}\ x = E_2\ \mathsf{in}\ E_1$). As argued in [24], 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 would be 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 [43] also contains a conditional-structure (*if-then-else*). It is not present in the definition of $\mathcal{L}_{\text{ML}}$ in [24], 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 [43], but not in [24]. 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 (extended with type constants 'C that can range over the normal types like int, bool, etc.), ranged over by A, B; these types can be *quantified*, creating *generic* types or *type schemes*, ranged over by σ, τ. 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 τ is defined as the replacement of *free* type variables by renaming the generic type variables of τ if necessary.

Definition 5.2 (ML TYPES AND TYPE SUBSTITUTION [43]) *i*) The set of ML *types* is defined in two layers.

$$\begin{array}{lcll} A, B & ::= & \varphi \mid \mathrm{c} \mid (A \rightarrow B) & (\textit{basic types}) \\ \sigma, \tau & ::= & A \mid (\forall \varphi . \tau) & (\textit{polymorphic types}) \end{array}$$

We will call types of the shape $\forall \varphi . \tau$ also *quantified* types. We will omit brackets as before, and abbreviate $(\forall \varphi_1 . (\forall \varphi_2 . \cdots (\forall \varphi_n . A) \cdots))$ by $\forall \overrightarrow{\varphi} . A$. We say that φ is bound in $\forall \varphi . \tau$, and define free and bound type variables accordingly; as is the case for λ-terms, we keep names of *bound* and *free* type variables separate.

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

$$\begin{array}{lcllcl} (\varphi \mapsto C)\ \varphi & = & C & (\varphi \mapsto C)\ A \rightarrow B & = & ((\varphi \mapsto C)\ A) \rightarrow ((\varphi \mapsto C)\ B) \\ (\varphi \mapsto C)\ \mathrm{c} & = & \mathrm{c} & (\varphi \mapsto C)\ \forall \varphi' . \psi & = & \forall \varphi' . ((\varphi \mapsto C)\ \psi) \\ (\varphi \mapsto C)\ \varphi' & = & \varphi' \quad (\varphi' \neq \varphi) & & & \end{array}$$

iii) Unification of is extended to type constants by:

$$\begin{array}{lll} \mathit{unify}\ \ \varphi\ \ \mathrm{c} &=& (\varphi \mapsto \mathrm{c}), \\ \mathit{unify}\ \ \mathrm{c}\ \ \varphi &=& \mathit{unify}\ \varphi\ \mathrm{c} \\ \mathit{unify}\ \ \mathrm{c}\ \ \mathrm{c} &=& \mathit{Id}_S \end{array}$$

Notice that all other cases involving a type constant hereby fail.

Since φ' is bound in $\forall\varphi'.\psi$, we can safely assume that, in $(\varphi \mapsto C)\ \forall\varphi'.\psi$, $\varphi \neq \varphi'$ and $\varphi' \notin fv(C)$.

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.

Remark that for $\forall\varphi_1\cdots\forall\varphi_n.A$ the set of type variables occurring in A is not necessarily equal to $\{\varphi_1,\ldots,\varphi_n\}$.

We now define the closure of a type with respect to a context; we need this in Definition 5.10.

Definition 5.3 $\overline{\Gamma}\ A = \forall\vec{\varphi}.A$ where $\vec{\varphi}$ are the type variables that occur free in A but not in Γ.

In the following definition we will give the derivation rules for Milner's system as presented in [25]; in the original paper [43] no derivation rules are given; instead, it contains a rather complicated definition of 'well typed prefixed expressions'.

Definition 5.4 (ML TYPE ASSIGNMENT [25]) We assume the existence of a function ν that maps each constant to 'its' type, which can be Int, Char, or even a polymorphic type, but is always a *closed type*, *i.e.* has no free type variables.

ML-*type assignment* and ML-*derivations* are defined by the following deduction system.

$$(Ax):\ \dfrac{}{\Gamma,x{:}\tau \vdash x:\tau} \qquad\qquad (\mathcal{C}):\ \dfrac{}{\Gamma \vdash c:\nu\,c}$$

$$(\to I):\ \dfrac{\Gamma,x{:}A \vdash E:B}{\Gamma \vdash \lambda x.E : A\to B} \qquad\qquad (\to E):\ \dfrac{\Gamma \vdash E_1 : A\to B \quad \Gamma \vdash E_2 : A}{\Gamma \vdash E_1E_2 : B}$$

$$(let):\ \dfrac{\Gamma \vdash E_1:\tau \quad \Gamma,x{:}\tau \vdash E_2 : B}{\Gamma \vdash \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : B} \qquad\qquad (fix):\ \dfrac{\Gamma,g{:}A \vdash E:A}{\Gamma \vdash \mathsf{fix}\,g.E : A}$$

$$(\forall I):\ \dfrac{\Gamma \vdash E:\tau}{\Gamma \vdash E:\forall\varphi.\tau}\ (\varphi\ \textit{not (free) in}\ \Gamma) \qquad (\forall E):\ \dfrac{\Gamma \vdash E:\forall\varphi.\tau}{\Gamma \vdash E:\tau[A/\varphi]}$$

Notice the use of basic types in rules $(\to I)$, $(\to E)$, and (let), and of polymorphic types in rules (Ax) and (let).

The quantification of type variables is introduced in order to model the substitution operation on types that we have seen in previous sections; rather than allowing a blanket replacement of all type variables in-one-go, using $\forall\varphi.\tau$ labels the type variable φ as being available for substitution, so stands for 'all φ in τ can be replaced by a basic type'. The side condition on the rule $(\forall I)$ is there to make sure that only those type variables are labelled that do not occur (also) in the context; the labelling only takes place on the type, and there is no back pointer of any kind to the context.

We can model the replacement of the type variable φ in A by the type B for the (closed) term M (which we modelled by $(\varphi \mapsto B)\,A$ and the soundness of substitution), through

$$\dfrac{\dfrac{\emptyset \vdash_{\mathrm{ML}} E : A}{\emptyset \vdash_{\mathrm{ML}} E : \forall\varphi.A}\,(\forall I)}{\emptyset \vdash_{\mathrm{ML}} E : A[B/\varphi]}\,(\forall E)$$

The *let*-construct corresponds in a way to the use of *definitions* in Λ^{NR}; notice that we can represent $\mathsf{n} = N : M$ by $\mathsf{let}\ \mathsf{n} = N\ \mathsf{in}\ M$. 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, E_1 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 $\forall \vec{\varphi}\,.\,A$, 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 $\Gamma \vdash_{\mathrm{ML}} E : \forall \vec{\varphi}\,.\,A$, where the free type variables in A are those occurring in Γ.

For the above defined notion of type assignment, we have a number of results. As for $\vdash_{\mathrm{C}}$, we can show:

Lemma 5.5 (Free variables): If : $\Gamma \vdash_{\mathrm{ML}} E : \tau$ and $x \in fv(E)$, then there exists σ such that $x{:}\sigma \in \Gamma$.

(Weakening): If $\Gamma \vdash_{\mathrm{ML}} E : \tau$, and Γ' is such that, for all $x{:}\sigma \in \Gamma'$ either $x{:}\sigma \in \Gamma$ or x does not occur free or bound in E, then $\Gamma' \vdash_{\mathrm{ML}} E : \tau$.

(Thinning): If $\Gamma, x{:}\sigma \vdash_{\mathrm{ML}} E : \tau$ and $x \notin fv(E)$, then $\Gamma \vdash_{\mathrm{ML}} E : \tau$.

The proof of this result is much as that for Exercise 2.18.

Before expressing the relation between terms and the types we can assign to them, we define some auxiliary relations between types.

Definition 5.6 We define the relation '$>_\Gamma$' as the smallest reflexive and transitive relation such that:

$$\begin{array}{rcll} \rho & >_\Gamma & \forall \varphi.\rho & (\varphi \textit{ not free in } \Gamma \textit{ and not bound in } \rho) \\ \forall \varphi.\rho & >_\Gamma & \rho[B/\varphi] & \end{array}$$

provided no φ' is free in A. If $\sigma >_\Gamma \tau$, we call τ a *generic instance* of σ.

Notice that '$>_\Gamma$' depends on Γ in the sense that each context induces a relation.

The relation '$>_\Gamma$' is used in Theorem 5.11, and $\sigma >_\Gamma \tau$ expresses that we can change a derivation for $\Gamma \vdash_{\mathrm{ML}} E : \sigma$ to one for $\Gamma \vdash_{\mathrm{ML}} E : \tau$ by applying a sequence of $(\forall I)$ and $(\forall E)$ steps.

The following follows easily:

Proposition 5.7 If $\Gamma \vdash_{\mathrm{ML}} E : \sigma$ and $\sigma >_\Gamma \tau$, then $\Gamma \vdash_{\mathrm{ML}} E : \tau$.

When a system has non-syntax directed type assignment rules, a generation lemma expresses the structure of derivable judgements for each syntactic construct.

Lemma 5.8 (GENERATION LEMMA) *i) If $\Gamma \vdash_{\mathrm{ML}} x : \sigma$, then there exists $x{:}\tau \in \Gamma$ such that $\tau >_\Gamma \sigma$.*

ii) If $\Gamma \vdash_{\mathrm{ML}} \lambda x.E : \sigma$, then there exist A,B such that $\Gamma, x{:}A \vdash_{\mathrm{ML}} E : B$, and $\sigma = \forall \vec{\varphi_i}\,.\,A \to B$, and $A \to B >_\Gamma \sigma$.

iii) If $\Gamma \vdash_{\mathrm{ML}} E_1 E_2 : \sigma$, then there exist A,B such that $\Gamma \vdash_{\mathrm{ML}} E_1 : A \to B$, $\Gamma \vdash_{\mathrm{ML}} E_2 : A$, and $B >_\Gamma \sigma$.

iv) If $\Gamma \vdash_{\mathrm{ML}}$ fix$g\,.\,E : \sigma$, then there exists A such that $\Gamma, g{:}A \vdash_{\mathrm{ML}} E : A$, and $\sigma = \forall \vec{\varphi_i}\,.\,A$, and $A >_\Gamma \sigma$.

v) If $\Gamma \vdash_{\mathrm{ML}}$ let $x = E_1$ in $E_2 : \sigma$, then there exists A,τ such that $\Gamma, x{:}\tau \vdash_{\mathrm{ML}} E_2 : A$, and $\Gamma \vdash_{\mathrm{ML}} E_1 : \tau$, and $A >_\Gamma \sigma$.

Proof: All are shown by induction on the structure of derivations. We show just two cases.

i) If $\Gamma \vdash_{\mathrm{ML}} x : \sigma$, then this derivation ends either with an application of rule:

(Ax): Then $x{:}\sigma \in \Gamma$; notice that $\sigma >_\Gamma \sigma$.

$(\forall I)$: Then $\sigma = \forall \varphi.\rho$, and $\Gamma \vdash_{\mathrm{ML}} x : \rho$, with φ not free in Γ. By induction, there exists $x{:}\tau \in \Gamma$ such that $\tau >_\Gamma \rho$; since $\rho >_\Gamma \forall \varphi.\rho$, also $\sigma >_\Gamma \forall \varphi.\rho$.

$(\forall E)$: Then $\sigma = \rho[B/\varphi]$ and $\Gamma \vdash_{\mathrm{ML}} x : \forall \varphi.\rho$. By induction, there exists $x{:}\tau \in \Gamma$ such that $\tau >_\Gamma \forall \varphi.\rho$; since $\forall \varphi.\rho >_\Gamma \rho[B/\varphi]$, also $\tau >_\Gamma \rho[B/\varphi]$.

v) If $\Gamma \vdash_{\mathrm{ML}}$ let $x = E_1$ in $E_2 : \sigma$, then this derivation ends either with an application of rule:

(let): Then there exists A,τ such that $\sigma = A$ and $\Gamma \vdash_{\mathrm{ML}} E_1 : \tau$ and $\Gamma, x{:}\tau \vdash_{\mathrm{ML}} E_2 : A$. Notice that $A >_\Gamma A$.

($\forall I$): Then $\sigma = \forall\varphi.\rho$, and $\Gamma \vdash_{\text{ML}} \textsf{let } x = E_1 \textsf{ in } E_2 : \rho$, with φ not free in Γ. By induction, there exists A, τ such that $\Gamma \vdash_{\text{ML}} E_1 : \tau$, $\Gamma, x{:}\tau \vdash_{\text{ML}} E_2 : A$, and $A >_\Gamma \rho$. Since $\rho >_\Gamma \forall\varphi.\rho$, also $A >_\Gamma \forall\varphi.\rho$.

($\forall E$): Then $\sigma = \rho[B/\varphi]$ and $\Gamma \vdash_{\text{ML}} \textsf{let } x = E_1 \textsf{ in } E_2 : \forall\varphi.\rho$. By induction,, there exists A, τ such that $\Gamma \vdash_{\text{ML}} E_1 : \tau$, $\Gamma, x{:}\tau \vdash_{\text{ML}} E_2 : A$, and $A >_\Gamma \forall\varphi.\rho$. Since $\forall\varphi.\rho >_\Gamma \rho[B/\varphi]$, also $A >_\Gamma \rho[B/\varphi]$. □

So, for each syntactic construct, a derivation for that construct contains an application of the type assignment rule directly dealing with it, perhaps followed by a sequence of applications of the rules ($\forall I$) and ($\forall E$).

It is easy to show that the subject reduction property holds also for ML (see Exercise 5.16).

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. Moreover, type assignment in System F is undecidable, as shown by Wells [55], whereas as we will see it is decidable in ML.

In understanding the (*let*)-rule, notice that the generic type τ is used. Assume that $\tau = \forall\vec{\varphi}.A$, and that in building the derivation for the statement $E_2 : B$, τ 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 $\vec{B}$ 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 $E_1 : \tau$ is obtained from $E_1 : A$ by (repeatedly) 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:

$$
\dfrac{
\dfrac{\dfrac{\vdots}{\Gamma \vdash E_1 : A}}{\Gamma \vdash E_1 : \forall\varphi.A}\,(\forall I)
\qquad
\dfrac{
\dfrac{\dfrac{}{\Gamma, x{:}\forall\varphi.A \vdash x : \forall\varphi.A}\,(Ax)}{\Gamma, x{:}\forall\varphi.A \vdash x : A[B/\varphi]}\,(\forall E)
\quad
\dfrac{\dfrac{}{\Gamma, x{:}\forall\varphi.A \vdash x : \forall\varphi.A}\,(Ax)}{\Gamma, x{:}\forall\varphi.A \vdash x : A[C/\varphi]}\,(\forall E)
\quad \vdots
}{\Gamma, x{:}\forall\varphi.A \vdash E_2 : D}
}{\Gamma \vdash \textsf{let } x = E_1 \textsf{ in } E_2 : D}\,(let)
$$

Since we can show the Substitution Lemma also for ML, from $\Gamma \vdash_{\text{ML}} E_1 : A$ we can show that $\Gamma \vdash_{\text{ML}} E_1 : A[B/\varphi]$ and $\Gamma \vdash_{\text{ML}} E_1 : A[C/\varphi]$ (notice that φ does not occur in Γ), and we can type the contraction of the redex as follows (notice that then quantification is no longer used):

$$
\dfrac{
\dfrac{\vdots}{\Gamma \vdash E_1 : A[B/\varphi]}
\qquad
\dfrac{\vdots}{\Gamma \vdash E_1 : A[C/\varphi]}
\quad \vdots
}{\Gamma \vdash E_2[E_1/x] : D}
$$

Example 5.9 The program '$I = \lambda x.x : II$' translates as '$\textsf{let } i = \lambda x.x \textsf{ in } ii$' which we can type by:

$$
\dfrac{
\dfrac{\dfrac{\dfrac{}{x{:}\varphi \vdash x : \varphi}\,(Ax)}{\emptyset \vdash \lambda x.x : \varphi\rightarrow\varphi}\,(\rightarrow I)}{\emptyset \vdash \lambda x.x : \forall\varphi.\varphi\rightarrow\varphi}\,(\forall I)
\qquad
\dfrac{
\dfrac{\dfrac{}{i{:}\forall\varphi.\varphi\rightarrow\varphi \vdash i : \forall\varphi.\varphi\rightarrow\varphi}}{i{:}\forall\varphi.\varphi\rightarrow\varphi \vdash i : (A\rightarrow A)\rightarrow A\rightarrow A}
\quad
\dfrac{\dfrac{}{i{:}\forall\varphi.\varphi\rightarrow\varphi \vdash i : \forall\varphi.\varphi\rightarrow\varphi}\,(Ax)}{i{:}\forall\varphi.\varphi\rightarrow\varphi \vdash i : A\rightarrow A}\,(\forall E)
}{i{:}\forall\varphi.\varphi\rightarrow\varphi \vdash ii : A\rightarrow A}\,(\rightarrow E)
}{\emptyset \vdash \textsf{let } i = \lambda x.x \textsf{ in } ii : A\rightarrow A}\,(let)
$$

As for rule (fix), remember that, to express recursion, we look for a solution to an equation

$$
\begin{array}{llll}
\mathcal{W}\,\Gamma\,\mathsf{c} & = & \langle id, B\rangle & \\
\quad\text{where} & \nu\,\mathsf{c} & = & \forall\vec{\varphi}.A \\
 & B & = & A[\overrightarrow{\varphi'/\varphi}] \\
 & \text{all } \varphi' & \textit{are} & \textit{fresh} \\
\mathcal{W}\,\Gamma\,x & = & \langle id, B\rangle & \\
\quad\text{where} & x{:}\forall\vec{\varphi}.A & \in & \Gamma \\
 & B & = & A[\overrightarrow{\varphi'/\varphi}] \\
 & \text{all } \varphi' & \textit{are} & \textit{fresh} \\
\mathcal{W}\,\Gamma\,(\lambda x.E) & = & \langle S,\, S(\varphi\to A)\rangle & \\
\quad\text{where} & \langle S,A\rangle & = & \mathcal{W}\,(\Gamma,x{:}\varphi)\,E \\
 & \varphi & \textit{is} & \textit{fresh}
\end{array}
$$

$$
\begin{array}{llll}
\mathcal{W}\,\Gamma\,(\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2) & = & \langle S_2\circ S_1, B\rangle & \\
\quad\text{where} & \langle S_1,A\rangle & = & \mathcal{W}\,\Gamma\,E_1 \\
 & \langle S_2,B\rangle & = & \mathcal{W}\,(S_1\Gamma,x{:}\sigma)\,E_2 \\
 & \sigma & = & \overline{S_1\Gamma}A \\
\mathcal{W}\,\Gamma\,(\mathsf{fix}\,g.E) & = & \langle S_2\circ S_1, S_2 A\rangle & \\
\quad\text{where} & \langle S_1,A\rangle & = & \mathcal{W}\,(\Gamma,g{:}\varphi)\,E \\
 & S_2 & = & \textit{unify}\,(S_1\varphi)\,A \\
 & \varphi & \textit{is} & \textit{fresh} \\
\mathcal{W}\,\Gamma\,(E_1E_2) & = & \langle S_3\circ S_2\circ S_1, S_3\varphi\rangle & \\
\quad\text{where} & \varphi & \textit{is} & \textit{fresh} \\
 & \langle S_1,A\rangle & = & \mathcal{W}\,\Gamma\,E_1 \\
 & \langle S_2,B\rangle & = & \mathcal{W}\,(S_1\Gamma)\,E_2 \\
 & S_3 & = & \textit{unify}\,(S_2A)\,(B\to\varphi)
\end{array}
$$

Figure 5. Milner's Algorithm $\mathcal{W}$

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 [24], is to add the constant $\mathbf{Y}$ to the calculus as discussed in Example 4.4.

Instead, here we follow the approach of [25] 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 $\mathsf{fix}\,g\ x.E$ as general fixed-point constructor, which would correspond to our $\mathsf{fix}\,g.\lambda x.E$; the rule then is formulated as follows:

$$(\mathit{fix}):\ \frac{\Gamma,g{:}A\to B,x{:}A\vdash E:B}{\Gamma\vdash \mathsf{fix}\,g\ x.E : A\to B}$$

This is, for example, the approach of [46].

Another approach is the use of letrec, a combination of *let* and *fix*, of the form

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

with derivation rule

$$(\mathit{letrec}):\ \frac{\Gamma,g{:}B\to C,x{:}B\vdash E_1:C \qquad \Gamma,g{:}\tau\vdash E_2:A}{\Gamma\vdash \mathsf{letrec}\ g=\lambda x.E_1\ \mathsf{in}\ E_2:A}\ (\tau=\overline{\Gamma}(B\to C))$$

This construct letrec $g = \lambda x.E_1$ in E_2 can be viewed as syntactic sugar for

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

but we will not be using these notations here.2

5.2 Milner's $\mathcal{W}$

We will now define Milner's 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 5.10 (MILNER'S ALGORITHM $\mathcal{W}$ [43]) Milner's type assignment algorithm for ML is defined in Figure 5. Notice the use of $\overline{S_1\Gamma}A$ (see Definition 5.3) 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 [56]. This is essentially due to the fact that, when a derivation for $\Gamma, x{:}\tau \vdash_{\text{ML}} E : A$ might exists, the abstraction $\lambda x.E$ need not be typeable.
- Type assignment is decidable.

In fact, $\mathcal{W}$ satisfies:

Theorem 5.11
- 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 $\Gamma' \vdash_{\text{ML}} E : A$, then $\mathcal{W}\,\Gamma\,E = \langle S, B\rangle$, and there is a substitution S' such that $\Gamma' = S'\,(S\,\Gamma)$ and $S'\,(S\,B) >_{\Gamma'} A$.*
- Soundness of $\mathcal{W}$. *For every term E: if $\mathcal{W}\,\Gamma\,E = \langle S, A\rangle$, then $S\,\Gamma \vdash_{\text{ML}} E : A$.*

Example 5.12 To express *addition* in ML, we can proceed as follows. We can define addition by:

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

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

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

Assuming we have:

$$\begin{array}{lcl}
\nu\,\mathsf{Succ} & = & \mathsf{Num}\to\mathsf{Num} \\
\nu\,\mathsf{Pred} & = & \mathsf{Num}\to\mathsf{Num} \\
\nu\,\mathsf{IsZero} & = & \mathsf{Num}\to\mathsf{Bool} \\
\nu\,\mathsf{Cond} & = & \forall\varphi.\mathsf{Bool}\to\varphi\to\varphi\to\varphi
\end{array}$$

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

Let

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

$$\mathcal{D}_2 = \dfrac{\dfrac{}{\Gamma \vdash_{\text{ML}} \mathsf{Succ} : \mathsf{N}\to\mathsf{N}}\,(\mathcal{C}) \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\text{ML}} a : \mathsf{N}\to\mathsf{N}\to\mathsf{N}}\,(Ax) \quad \dfrac{\dfrac{}{\Gamma \vdash_{\text{ML}} \mathsf{Pred} : \mathsf{N}\to\mathsf{N}}\,(\mathcal{C}) \quad \dfrac{}{\Gamma \vdash_{\text{ML}} x : \mathsf{N}}\,(Ax)}{\Gamma \vdash_{\text{ML}} \mathsf{Pred}\ x : \mathsf{N}}\,(\to E)}{\Gamma \vdash_{\text{ML}} a\ (\mathsf{Pred}\ x) : \mathsf{N}\to\mathsf{N}}\,(\to E) \quad \dfrac{}{\Gamma \vdash_{\text{ML}} y : \mathsf{N}}\,(Ax)}{\Gamma \vdash_{\text{ML}} a\ (\mathsf{Pred}\ x)\ y : \mathsf{N}}\,(\to E)}{\Gamma \vdash_{\text{ML}} \mathsf{Succ}\ (a\ (\mathsf{Pred}\ x)\ y) : \mathsf{N}}\,(\to E)$$

then we can construct:

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

5.3 Polymorphic recursion

In [45] A. Mycroft defined a generalisation of Milner's system (independently, a similar system was defined in [35]). 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

$$\begin{array}{lcl} \mathsf{map}\,(f,l) & = & \textit{if}\ \mathsf{null}\,(l)\ \textit{then}\ l\ \textit{else}\ \mathsf{cons}\,(\ f\ (\mathsf{hd}\ l),\mathsf{map}\,(f,\mathsf{tl}\ l)\ ) \\ \mathsf{squarelist}\,(l) & = & \mathsf{map}\ (\lambda x.x^2,l) \end{array}$$

Using the notation of Λ^{NR}, this becomes:

$$\begin{array}{lcll} \mathsf{map} & = & \lambda fl.\mathsf{Cond}\ (\mathsf{null}\ l)\ l\ (\mathsf{cons}\ (f\ (\mathsf{hd}\ l))\ (\mathsf{map}\ f\ (\mathsf{tl}\ l))\ ) & ; \\ \mathsf{squarelist} & = & \lambda l.\mathsf{map}\ (\lambda x.\mathsf{mul}\,x\,x)\ l & : \\ \multicolumn{3}{l}{\mathsf{squarelist}\ (\mathsf{cons}\ 2\ \mathsf{nil})} \end{array}$$

where Cond, hd, tl, null (the test if a list is empty, i.e, the same as nil), cons, and mul are assumed to be familiar list constructors and functions, and we supply a final term to build a program. 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 '$\langle\cdot,\cdot\rangle$', and the translation of the above expression into $\mathcal{L}_{\mathrm{ML}}$ will be:

$$\begin{array}{lcl} \textit{let}\ \langle\mathsf{map},\mathsf{squarelist}\rangle & = & \mathsf{fix}\,\langle m,s\rangle.\langle\lambda gl.\mathsf{Cond}\ (\mathsf{null}\ l)\ l\ (\mathsf{cons}\ (g\ (\mathsf{hd}\ l))(mg(\mathsf{tl}\ l))), \\ & & \qquad\qquad\qquad \lambda l.(m(\lambda x.\mathsf{mul}\,x\,x)\ l)\rangle \\ & & \mathsf{in}\ (\mathsf{squarelist}\ (\mathsf{cons}\ 2\ \mathsf{nil})) \end{array}$$

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

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

while the definition of map alone would yield the type:

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

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 recognise mutual recursive rules, and treat them as one definition. (Easy to implement, but difficult to formalise, a problem we run into in Section 6). Then, the translation of the above program could be:

$$\begin{array}{l} \textit{let}\ \mathsf{map}\ =\ (\mathsf{fix}\,m.\lambda gl.\mathsf{Cond}\ (\mathsf{null}\ l)\ l\ (\mathsf{cons}\ (g\ (\mathsf{hd}\ l))(mg(\mathsf{tl}\ l)))) \\ \qquad\qquad \mathsf{in}\ \ (\textit{let}\ \mathsf{squarelist}\ =\ (\lambda l.(\mathsf{map}\ (\lambda x.\mathsf{mul}\,x\,x)\ l)) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \ (\mathsf{squarelist}\ (\mathsf{cons}\ 2\ \mathsf{nil}))) \end{array}$$

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).

Definition 5.13 ([45]) *Mycroft type assignment* is defined by replacing rule (fix) by:

$$(\mathsf{fix}):\ \frac{\Gamma,g{:}\tau \vdash_{\mathrm{Myc}} E:\tau}{\Gamma \vdash_{\mathrm{Myc}} \mathsf{fix}\,g.E:\tau}$$

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 the recursive variable can be

$$\begin{array}{lcl} pp_{\Lambda^{\mathrm{NR}}}\; x\; \mathcal{E} & = & \langle x{:}\varphi;\, \varphi\rangle \\ & \text{where} & \varphi \textit{ is fresh} \\ pp_{\Lambda^{\mathrm{NR}}}\; name\; \mathcal{E} & = & \langle \emptyset;\, \mathit{FreshInstance}\,(\mathcal{E}\,name)\rangle \\ pp_{\Lambda^{\mathrm{N}}}\; \lambda x.M\; \mathcal{E} & = & \langle \Pi';\, A{\to}P\rangle \quad (\Pi = \Pi', x{:}A) \\ & & \langle \Pi;\, \varphi{\to}P\rangle \quad (x \notin \Pi) \\ & \mathit{where} & \langle \Pi;\, P\rangle \;=\; pp_{\Lambda^{\mathrm{N}}}\; M\; \mathcal{E} \\ & & \varphi \quad \mathit{is\ fresh} \end{array}$$

$$\begin{array}{lcl} pp_{\Lambda^{\mathrm{N}}}\; MN\; \mathcal{E} & = & S_2{\circ}S_1\, \langle \Pi_1 \cup \Pi_2;\, \varphi\rangle \\ \text{where} & & \langle \Pi_1;\, P_1\rangle \;=\; pp_{\Lambda^{\mathrm{N}}}\; M\; \mathcal{E} \\ & & \langle \Pi_2;\, P_2\rangle \;=\; pp_{\Lambda^{\mathrm{N}}}\; N\; \mathcal{E} \\ & & S_1 \;=\; \mathit{unify}\; P_1\; P_2{\to}\varphi \\ & & S_2 \;=\; \mathit{UnifyContexts}\; (S_1\,\Pi_1)\; (S_1\,\Pi_2) \\ & & \varphi \quad \mathit{is\ fresh} \end{array}$$

$$\begin{array}{lcl} \mathit{CheckEnv}\; (\mathit{Defs}\,;\, name = M)\; \mathcal{E} & = & (\mathit{CheckEnv}\; \mathit{Defs}\; \mathcal{E}) \wedge (\mathcal{E}\; name) = P \\ & & \text{where} \quad \langle \emptyset, P\rangle \;=\; pp_{\Lambda^{\mathrm{N}}}\; M\; \mathcal{E} \\ \mathit{CheckEnv}\; \epsilon\; \mathcal{E} & = & \mathsf{true} \\ & & \\ pp_{\Lambda^{\mathrm{N}}}\; (\mathit{Defs};\, M)\; \mathcal{E} & = & pp_{\Lambda^{\mathrm{N}}}\; M\; \mathcal{E}, \quad \mathit{if\ CheckEnv\ Defs}\; \mathcal{E} \end{array}$$

Figure 6. A type-check algorithm for Λ^{NR} using Mycoft's approach

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 [36].

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

Definition 5.14 (MYCROFT-STYLE TYPE ASSIGNMENT FOR Λ^{NR}) Type assignment for programs in Λ^{NR} using polymorphic recursion is defined through the rules (Ax), $({\to}I)$, $({\to}E)$, and $(\mathit{Program})$ of Definition 4.5, and replacing the rules (ϵ), (Call), $(\mathit{Rec\ Call})$, (Def), and $(\mathit{Rec\ Def})$ by following rules:

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

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

Mycroft's approach is implemented by the algorithm in Figure 6.

Notice that these rules now *check the environment*, rather than construct it. In this approach, in the algorithm the environment never gets updated, so has to be provided (by the user) before the program starts running. This implies that the above algorithm is more a *type-check* algorithm rather than a *type-inference* algorithm as are those that we have seen above.

5.4 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, in Mycroft's system

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

is a derivable statement (as shown in Figure 7, where $\Gamma = g{:}\forall \varphi_1\varphi_2\,.\,\varphi_1{\to}\varphi_2$; notice that the type variables φ_3 and φ_4 do not occur (free) in the context, so can be bound, and that we use renaming of bound type variables when quantifying). 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

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$\Gamma,a{:}\varphi_3\to\varphi_4,b{:}B \vdash a : \varphi_3\to\varphi_4$}
\RightLabel{$(\to I)$}
\UnaryInfC{$\Gamma,b{:}B \vdash \lambda b.a : (\varphi_3\to\varphi_4)\to B\to\varphi_3\to\varphi_4$}
\RightLabel{$(\to I)$}
\UnaryInfC{$\Gamma \vdash \lambda ab.a : (\varphi_3\to\varphi_4)\to B\to\varphi_3\to\varphi_4$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$\Gamma \vdash g : \forall\varphi_1\varphi_2\,.\,\varphi_1\to\varphi_2$}
\RightLabel{$(\forall E)$}
\UnaryInfC{$\Gamma \vdash g : \forall\varphi_2\,.\,(C\to C)\to\varphi_2$}
\RightLabel{$(\forall E)$}
\UnaryInfC{$\Gamma \vdash g : (C\to C)\to\varphi_3\to\varphi_4$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$\Gamma,c{:}C \vdash c : C$}
\RightLabel{$(\to I)$}
\UnaryInfC{$\Gamma \vdash \lambda c.c : C\to C$}
\RightLabel{$(\to E)$}
\BinaryInfC{$\Gamma \vdash g\,\lambda c.c : \varphi_3\to\varphi_4$}
\RightLabel{$(\to E)$}
\BinaryInfC{$\Gamma \vdash (\lambda ab.a)\,(g\,\lambda c.c) : B\to\varphi_3\to\varphi_4$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$\Gamma \vdash g : \forall\varphi_1\varphi_2\,.\,\varphi_1\to\varphi_2$}
\RightLabel{$(\forall E)$}
\UnaryInfC{$\Gamma \vdash g : \forall\varphi_2\,.\,(D\to E\to D)\to\varphi_2$}
\RightLabel{$(\forall E)$}
\UnaryInfC{$\Gamma \vdash g : (D\to E\to D)\to B$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$\Gamma,e{:}E \vdash d : D$}
\RightLabel{$(\to I)$}
\UnaryInfC{$\Gamma,d{:}D \vdash \lambda e.d : E\to D$}
\RightLabel{$(\to I)$}
\UnaryInfC{$\Gamma \vdash \lambda de.d : D\to E\to D$}
\RightLabel{$(\to E)$}
\BinaryInfC{$\Gamma \vdash g\,\lambda de.d : B$}
\RightLabel{$(\to E)$}
\BinaryInfC{$\Gamma \vdash (\lambda ab.a)\,(g\,\lambda c.c)\,(g\,\lambda de.d) : \varphi_3\to\varphi_4$}
\RightLabel{$(\forall I)$}
\UnaryInfC{$\Gamma \vdash (\lambda ab.a)\,(g\,\lambda c.c)\,(g\,\lambda de.d) : \forall\varphi_2\,.\,\varphi_3\to\varphi_2$}
\RightLabel{$(\forall I)$}
\UnaryInfC{$\Gamma \vdash (\lambda ab.a)\,(g\,\lambda c.c)\,(g\,\lambda de.d) : \forall\varphi_1\varphi_2\,.\,\varphi_1\to\varphi_2$}
\RightLabel{(fix)}
\UnaryInfC{$\emptyset \vdash \mathsf{fix}\,g\,.\,(\lambda ab.a)\,(g\,\lambda c.c)\,(g\,\lambda de.d) : \forall\varphi_1\varphi_2\,.\,\varphi_1\to\varphi_2$}
\end{prooftree}
$$

Figure 7. A derivation for $\emptyset \vdash_{\mathrm{MYC}} \mathsf{fix}\,g\,.\,(\lambda ab.a)\,(g\,\lambda c.c)(g\,\lambda de.d) : \forall\varphi_1\varphi_2\,.\,\varphi_1\to\varphi_2$

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$\Gamma \vdash r : \forall\varphi_1\varphi_2\varphi_3\,.\,\varphi_1\to\varphi_2\to\varphi_3$}
\RightLabel{$(\forall E)$}
\UnaryInfC{$\Gamma \vdash r : \varphi_7\to\varphi_4\to\varphi_6$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$\Gamma \vdash r : \forall\varphi_1\varphi_2\varphi_3\,.\,\varphi_1\to\varphi_2\to\varphi_3$}
\RightLabel{$(\forall E)$}
\UnaryInfC{$\Gamma \vdash r : \varphi_5\to(A\to B\to A)\to\varphi_7$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$\Gamma \vdash y : \varphi_5$}
\RightLabel{$(\to E)$}
\BinaryInfC{$\Gamma \vdash r\,y : (A\to B\to A)\to\varphi_7$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$\Gamma,a{:}A,b{:}B \vdash a : A$}
\RightLabel{$(\to I)$}
\UnaryInfC{$\Gamma,a{:}A \vdash \lambda b.a : B\to A$}
\RightLabel{$(\to I)$}
\UnaryInfC{$\Gamma \vdash \lambda ab.a : A\to B\to A$}
\RightLabel{$(\to E)$}
\BinaryInfC{$\Gamma \vdash r\,y\,(\lambda ab.a) : \varphi_7$}
\RightLabel{$(\to E)$}
\BinaryInfC{$\Gamma \vdash r\,(r\,y\,(\lambda ab.a)) : \varphi_4\to\varphi_6$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$\Gamma \vdash x : \varphi_4$}
\RightLabel{$(\to E)$}
\BinaryInfC{$\Gamma = r{:}\forall\varphi_1\varphi_2\varphi_3\,.\,\varphi_1\to\varphi_2\to\varphi_3, x{:}\varphi_4, y{:}\varphi_5 \vdash r\,(r\,y\,(\lambda ab.a))\,x : \varphi_6$}
\RightLabel{$(\to I)$}
\UnaryInfC{$r{:}\forall\varphi_1\varphi_2\varphi_3\,.\,\varphi_1\to\varphi_2\to\varphi_3, x{:}\varphi_4 \vdash \lambda y.r\,(r\,y\,(\lambda ab.a))\,x : \varphi_5\to\varphi_6$}
\RightLabel{$(\to I)$}
\UnaryInfC{$r{:}\forall\varphi_1\varphi_2\varphi_3\,.\,\varphi_1\to\varphi_2\to\varphi_3 \vdash \lambda xy.r\,(r\,y\,(\lambda ab.a))\,x : \varphi_4\to\varphi_5\to\varphi_6$}
\RightLabel{$(\forall I)$}
\UnaryInfC{$r{:}\forall\varphi_1\varphi_2\varphi_3\,.\,\varphi_1\to\varphi_2\to\varphi_3 \vdash \lambda xy.r\,(r\,y\,(\lambda ab.a))\,x : \forall\varphi_1\varphi_2\varphi_3\,.\,\varphi_1\to\varphi_2\to\varphi_3$}
\RightLabel{(fix)}
\UnaryInfC{$\emptyset \vdash \mathsf{fix}\,r\,.\,\lambda xy.r\,(r\,y\,(\lambda ab.a))\,x : \forall\varphi_1\varphi_2\varphi_3\,.\,\varphi_1\to\varphi_2\to\varphi_3$}
\end{prooftree}
$$

Figure 8. A type derivation for $\vdash_{\mathrm{MYC}} \mathsf{fix}\,r\,.\,\lambda xy.r\,(r\,y\,(\lambda ab.a))\,x : \forall\varphi_4\varphi_5\varphi_6\,.\,\varphi_4\to\varphi_5\to\varphi_6$

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. For example, the statement

$$\emptyset \vdash_{\mathrm{MYC}} \mathsf{fix}\,r\,.\,\lambda xy.r\,(r\,y\,(\lambda ab.a))\,x : \forall\varphi_1\varphi_2\varphi_3\,.\,\varphi_1\to\varphi_2\to\varphi_3$$

is derivable in Mycroft's system, as shown in Figure 8 (where we have collapsed sequences of $(\forall I)$ and $(\forall E)$ steps). Notice that $\forall\varphi_4\varphi_5\varphi_6\,.\,\varphi_4\to\varphi_5\to\varphi_6 = \forall\varphi_1\varphi_2\varphi_3\,.\,\varphi_1\to\varphi_2\to\varphi_3$, so we can apply rule (fix); moreover A and B are irrelevant for this construction. That term is also typeable in Milner's system, as shown in Exercise 5.21.

We have seen the differences between the two approaches also in the previous section. Milner's ML uses fix to represent recursion definitions, and the occurrences of f in E in $\mathsf{fix}\,f\,.\,E$ are recursive calls to the function defined by $\mathsf{fix}\,f\,.\,E$. The type assignment rule for fix states that if $\Gamma, f{:}A \vdash E : A$, then $\Gamma \vdash \mathsf{fix}\,f\,.\,E : A$, so the recursive definition is only considered typeable if $\mathsf{fix}\,f\,.\,E : A$, E, and all recursive calls for f in E all have the same type. ML then allows the type for $\mathsf{fix}\,f\,.\,E : A$ to be quantified, and allow recursive definitions to be *let*-bound (which, as we know, models calling a definition), and be polymorphic.

When using this approach in Λ^{NR}, the recursive calls all have the same type as the definition; calls outside the definitions can be treated as polymorphic. Mycroft does not insist on typeing recursive calls with the same type, so using his approach for Λ^{NR} all calls can be polymorphic. This is reflected in the type assignment rules for calls.

Exercises

* *Exercise 5.15 Show that* ML*-substitution is a sound operation in the ML type assignment system. You can use* $(S\,\sigma)[(S\,A)/\varphi] = S\,(\sigma[A/\varphi])$*, and can assume that free and bound (by* $\forall$*) type-variables are distinct.*

* *Exercise 5.16 Show that type assignment in* ML *satisfies subject reduction: if* $\Gamma \vdash_{\mathrm{ML}} M:\sigma$*, and* $M \rightarrow_{\mathrm{ML}} N$*, then* $\Gamma \vdash_{\mathrm{ML}} N:\sigma$*.*

Exercise 5.17 Assuming the types Bool *and* Int *and the following extensions to* ML*:*

- *a prefix addition '+' with type* Int$\rightarrow$Int$\rightarrow$Int*;*
- *a conditional language construct '*Cond*' with type* Bool$\rightarrow A\rightarrow A\rightarrow A$*;*
- *a test for zero '=0' with type* Int$\rightarrow$Bool*;*
- *a function '-1' with type* Int$\rightarrow$Int*;*

express the multiplication function *as an* ML-expression*, and give a derivation that types this term; you can assume that all added functions are treated as constants.*

Exercise 5.18 Assume now the type constructor $[\cdot]$ *and the constructor* Cons *with type* $\forall\varphi.\varphi\rightarrow[\varphi]\rightarrow[\varphi]$*. Abbreviating your answer to Exercise 5.17 by* $\mathcal{D} :: \ \vdash \mathsf{Times} : \mathsf{Int}\rightarrow\mathsf{Int}\rightarrow\mathsf{Int}$*, and using the information for the previous part, write an* ML-expression *that represents the (infinite) list of* all square numbers*. Show that this term is typeable.*

Exercise 5.19 Express, using the information from Exercise 5.17, the factorial function as an ML expression*, and show that this term is* typeable*.*

Exercise 5.20 Using your answer to Exercise 5.19, write an ML expression *that represents the list of all factorial numbers, and show, abbreviating if needed your answer to the previous part, that this term is* typeable*.*

Exercise 5.21 Take $R = \mathit{fix}\,r\,.\,\lambda xy.r\,(r\,y\,(\lambda ab.a))\,x$*, the term of Figure 8; give the derivation for*

$$\vdash_{\mathrm{ML}} R : \forall\varphi_4\varphi_5\,.\,(\varphi_4\rightarrow\varphi_5\rightarrow\varphi_4)\rightarrow(\varphi_4\rightarrow\varphi_5\rightarrow\varphi_4)\rightarrow\varphi_4\rightarrow\varphi_5\rightarrow\varphi_4.$$

6 Pattern matching: term rewriting

The notion of reduction we will study in this section is that of *term rewriting* [38, 39], a notion of computation which main feature is that of *pattern matching*, making it syntactically closer to most functional programming languages than the pure Lambda Calculus.

6.1 Term Rewriting Systems

Term rewriting systems can be seen as an extension of the λ-calculus 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 6.1 (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 '·' and obsolete brackets.

iii) A *replacement*, written $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ or as a capital character like 'R' when we need not be specific, is an operation on terms where term variables are consistently replaced by terms, and corresponds to the implicit substitution of the λ-calculus. We write t^{R} for the result of applying the replacement 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 6.2 (Reduction) *i*) A *rewrite rule* is a pair (l,r) of terms. Often, a rewrite rule will get a name, e.g. **r**, and we write

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

Two conditions are imposed:

a) $l = F\,t_1 \cdots t_n$, for some $F \in \mathcal{F}$ with arity n 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 x and there is a t_j $(1 \leq i \neq j \leq n)$ such that $x \in fv(t_j)$.

iii) A rewrite rule $l \to r$ determines a set of *rewrites* $l^{\mathrm{R}} \to r^{\mathrm{R}}$ for all replacements 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 $\mathsf{C}\lceil\cdot\rfloor$; this gives rise to *rewrite steps* $\mathsf{C}\lceil t\rfloor \to \mathsf{C}\lceil t'\rfloor$. 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 $t_0 \to^* t_n$.

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 there to be 'filled' with sub-terms during matching, which 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.

As we have seen above, Combinatory Logic is a special TRS.

Definition 6.3 A *Term Rewriting System* (TRS) is defined by a triple $\langle \mathcal{F},\mathcal{X},\mathbf{R}\rangle$ 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 6.4 Let $\langle \mathcal{F},\mathcal{X},\mathbf{R}\rangle$ be a TRS. In a rewrite rule $\mathbf{r} : F\,t_1 \cdots t_n \to r \in \mathbf{R}$, $F \in \mathcal{F}$ is called *the defined symbol* of **r**, and **r** is said to *define F*. F is *a defined symbol* if there is a rewrite rule that defines it, and $Q \in \mathcal{F}$ is called a *constructor* if Q is not a defined symbol.

The notion of defined symbols is important when we define how we type a rule.

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 6.5 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.

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

With this notion of rewriting, we obtain more than just the normal functional paradigm: there, in a rewrite rule, function symbols are not allowed to appear in 'constructor position' (i.e in a pattern) and vice-versa. For example, in the rule $F\,t_1 \cdots t_n \ \to\ r$, the symbol F appears in function position and is thereby a function symbol (we have called 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.

Example 6.1 (Surjective Pairing) In particular, the following is a correct TRS.

$$
\begin{array}{lcl}
\mathsf{In\text{-}left}\ (\mathsf{Pair}\ x\ y) & \to & x \\
\mathsf{In\text{-}right}\ (\mathsf{Pair}\ x\ y) & \to & y \\
\mathsf{Pair}\ (\mathsf{In\text{-}left}\ x)\ (\mathsf{In\text{-}right}\ x) & \to & x
\end{array}
$$

A difficulty with this TRS is that it forms Klop's famous 'Surjective Pairing' example [37]; this function cannot be expressed in the λ-calculus because when added to the λ-calculus, the Church-Rosser property no longer holds.

This implies that, although both the λ-calculus 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 the λ-calculus, so a full-purpose translation (interpretation) of TRS in the λ-calculus is not feasible.

6.2 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 6.6 (Environment) Let $\langle \mathcal{F}, \mathcal{X}, \mathbf{R} \rangle$ be a TRS. An *environment* for this TRS is a mapping $\mathcal{E} : \mathcal{F} \to \mathcal{T}_{\mathrm{C}}$.

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 6.7 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 6.8 (Type Assignment on Terms) Let $\langle \mathcal{F}, \mathcal{X}, \mathbf{R} \rangle$ be a TRS, and $\mathcal{E}$ and environment for it. *Type assignment* (with respect to $\mathcal{E}$) is defined by the following natural deduction system. Note the use of a substitution in rule (*Call*).

$$(Ax): \frac{}{\Gamma, x{:}A; \mathcal{E} \vdash x : A} \qquad (Call): \frac{}{\Gamma; \mathcal{E}, F{:}A \vdash F : SA} \qquad (\rightarrow E): \frac{\Gamma; \mathcal{E} \vdash t_1 : A \rightarrow B \quad \Gamma; \mathcal{E} \vdash t_2 : A}{\Gamma; \mathcal{E} \vdash t_1 t_2 : B}$$

As before, the use of an environment in rule $(Call)$ introduces a notion of polymorphism for our function symbols. 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 6.3.
- Subject reduction. This will be proven in Section 6.4.

Remark 6.9 It is, in general, not possible to prove a strong normalisation result for typeable terms: take a term t that is typeable, and the rewrite 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

$$F\,\overrightarrow{\mathsf{C}[\vec{x}]} \;\rightarrow\; \mathsf{C}'[F\,\overrightarrow{\mathsf{C}_1[\vec{x}]}\,\ldots\,F\,\overrightarrow{\mathsf{C}_m[\vec{x}]}],$$

where, for every $1 \leq j \leq m$, $\overrightarrow{\mathsf{C}_j[\vec{x}]}$ is a strict subterm of $\overrightarrow{\mathsf{C}[\vec{x}]}$, so F is only called recursively on terms that are substructures of its initial arguments (see [7] for details); this scheme generalises primitive recursion.

Notice that, for example, the rules of a combinator system like CL (see Section 6.6) are not recursive, so this result gives us immediately a strong normalisation result for combinator systems.

6.3 The principal pair for a term

In this subsection, the principal pair for a term t with respect to the environment $\mathcal{E}$ 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 6.10 Let $\langle \mathcal{F}, \mathcal{X}, \mathbf{R} \rangle$ be a TRS, and $\mathcal{E}$ and environment for it. We define the notion $pp\ t\ \mathcal{E} = \langle \Pi ; P \rangle$ inductively by:

$$\begin{array}{lcl} pp\ x\ \mathcal{E} & = & \langle x{:}\varphi ; \varphi \rangle \\ & & \text{where } \varphi = \mathit{fresh} \\ pp\ F\ \mathcal{E} & = & \langle \phi ; \mathit{FreshInstance}(\mathcal{E}\,F) \rangle \end{array} \qquad \begin{array}{lcl} pp\ t_1 t_2\ \mathcal{E} & = & S\langle \Pi_1 \cup \Pi_2 ; \varphi \rangle \\ & & \text{where } \begin{array}{lcl} \varphi & = & \mathit{fresh} \\ \langle \Pi_1 ; P_1 \rangle & = & pp\ t_1\ \mathcal{E} \\ \langle \Pi_2 ; P_2 \rangle & = & pp\ t_2\ \mathcal{E} \\ S & = & \mathit{unify}\ P_1\ P_2 \rightarrow \varphi \end{array} \end{array}$$

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.

Lemma 6.11 (Soundness of substitution) *Let $\langle \mathcal{F}, \mathcal{X}, \boldsymbol{R} \rangle$ be a* TRS, *and $\mathcal{E}$ and environment for it. If $\Gamma; \mathcal{E} \vdash t : A$, then $S\,\Gamma; \mathcal{E} \vdash t : S\,A$, for every type-substitution S.*

(see Exercise 6.28).

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

Theorem 6.12 (Completeness of substitution) *Let $\langle \mathcal{F}, \mathcal{X}, \boldsymbol{R} \rangle$ be a* TRS, *and $\mathcal{E}$ and environment for it. If $\Gamma; \mathcal{E} \vdash t : A$, then there are Π, P, and a substitution S such that: $pp\ t\ \mathcal{E} = \langle \Pi ; P \rangle$, and $S\,\Pi \subseteq \Gamma$, $S\,P = A$.*

Proof: By induction on the structure of t.

$(t \equiv x)$: Then $x{:}A \in \Gamma$. Then there is a φ such that $pp\ x\ \mathcal{E} = \langle x{:}\varphi;\ \varphi\rangle$. Take $S = (\varphi \mapsto A)$.

$(t \equiv F)$: Then $\Pi = \emptyset$, and $P = \mathit{FreshInstance}(\mathcal{E}\,F)$. By rule (Call) there exists a substitution S such that $A = S(\mathcal{E}\,F)$. But then there exists a substitution S' such that $A = S'(\mathit{FreshInstance}(\mathcal{E}\,F))$.

$(t \equiv t_1\,t_2)$: Then there exists $B \in \mathcal{T}_C$ such that $\Gamma;\mathcal{E} \vdash t_1 : B{\to}A$, and $\Gamma;\mathcal{E} \vdash t_2 : B$. By induction, for $i = 1,2$, there are Π_i, P_i, and a substitution S_i such that

$$pp\ t_i\ \mathcal{E} = \langle \Pi_i;\ P_i\rangle,\ S_1\,\Pi_1 \subseteq \Gamma,\ S_2\,\Pi_2 \subseteq \Gamma,\ S_1\,P_1 = B \to A, \text{and } S_2\,P_2 = B,$$

Let φ be a fresh type-variable; since now $S_1\,P_1 = B \to A = S_2\circ(\varphi \mapsto A)\,(P_2{\to}\varphi)$, by Property 2.13, there exists substitutions S_u, S' such that $S_u = \mathit{unify}\ P_1\ P_2{\to}\varphi$, and $S_1\circ S_2\circ(\varphi \mapsto A) = S'\circ S_u = S_2\circ(\varphi \mapsto A)\circ S_1$. Take $S = S_1\circ S_2\circ(\varphi \mapsto A)$, and $C = S\,\varphi$.

6.4 Subject reduction

By Definition 6.2, if a term t is rewritten to the term t' using the rewrite rule $l \to r$, 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 $l \to r$, that satisfy:

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

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, it is easy to formulate a condition that rewrite rules should satisfy in order to be acceptable.

Definition 6.13 Let $\langle \mathcal{F},\mathcal{X},\mathbf{R}\rangle$ be a TRS, and $\mathcal{E}$ and environment for it.

i) We say that $l \to r \in \mathbf{R}$ with defined symbol F *is typeable with respect to* $\mathcal{E}$, if there are Π, P and $\mathcal{E}$ such that $pp\ l\ \mathcal{E} = \langle \Pi;\ P\rangle$, $\Pi;\mathcal{E} \vdash r : P$, and such that the leftmost occurrence of F in finding $\langle \Pi;\ P\rangle$ (so in the derivation for $\Pi;\mathcal{E} \vdash l : P$) is typed with $\mathcal{E}(F)$.

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

To illustrate the need of separating how we type the defined symbol of a rule, consider how we deal with a term like S K. If it is just a subterm, then we see both S and K as calls, and allow their types to be instances of the type stored for them in the environment, so can take fresh instances for both. If S K is the leftmost part of the left-hand side of a rewrite rule, we have

$$\mathsf{S}\ \mathsf{K}\ t_1 \cdots t_n \ \to\ \mathit{Rhs}$$

for some $t_1,\ldots,t_n$ with $n \geq 0$, then S is the defined symbol, so is *not considered to be a call*, but rather the symbol that is being defined in this rule, much like `fact` is defined by:

```
fact 0 = 1
fact (Succ n) = (Succ n) x (fact n)
```

Then for that defining occurrence (or all others, recursive calls, in the rule if we use Milner's approach), we use the environment type for S; this is a way of checking that the type we have in the environment is the correct one, one that fits the definition of S, i.e. fits the rule. Since K is not the defined symbol, it is a call, so we take an instance.

Notice that the notion $pp\ t\ \mathcal{E}$ is defined independently from the definition of typeable rewrite rules; the structure of the rules is only represented through the types of the defined symbols in $\mathcal{E}$. Also, since only *'the leftmost occurrence of F in the derivation for $\Pi;\mathcal{E} \vdash l : P$ is typed with $\mathcal{E}(F)$'*, this notion of type assignment uses Mycroft's solution for recursion; using Milner's, the definition would have defined *'all occurrences of F in the derivations for $\Pi;\mathcal{E} \vdash l : P$*

and $\Pi;\mathcal{E} \vdash r : P$ 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 6.14 Let $\langle \mathcal{F},\mathcal{X},\boldsymbol{R}\rangle$ be a TRS*, and $\mathcal{E}$ and environment for it. If F is the defined symbol of the typeable rewrite rule $F\,t_1 \cdots t_n \to r \in \boldsymbol{R}$, then there are contexts Π,Γ, and types A_i ($i \in \underline{n}$) and A such that*

$$\begin{array}{lcl} \mathcal{E}\,F & = & A_1 \to \cdots \to A_n \to A, \\ pp\; l\; \mathcal{E} & = & \langle \Pi;\, A\rangle \\ \Pi;\mathcal{E} \vdash t_i : A_i & & \\ \Gamma;\mathcal{E} \vdash l : A, \text{ and} & & \\ \Gamma;\mathcal{E} \vdash r : A. & & \end{array}$$

Proof: Easy, using Theorem 6.12 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 typeable, take the 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 6.15 As an example of a rewrite rule that does not satisfy the above restriction, so will *not* be considered to be typeable, take

$$M\,(S\,x\,y) \;\to\; S\,I\,y.$$

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 $pp\; M\,(S\,x\,y)\;\mathcal{E}$, we assign types to nodes in the tree in the following way. Let

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

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

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

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

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

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

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

$$\dfrac{\dfrac{\overline{\phi;\mathcal{E}\vdash M:(C\to C)\to C\to\varphi_5} \quad \dfrac{\dfrac{\overline{\phi;\mathcal{E}\vdash S:D} \quad \overline{\phi;\mathcal{E}\vdash K:C\to C\to C}}{\phi;\mathcal{E}\vdash S\,K:(C\to C)\to C\to C} \quad \overline{\phi;\mathcal{E}\vdash I:C\to C}}{\phi;\mathcal{E}\vdash S\,K\,I:C\to C}}{\phi;\mathcal{E}\vdash M\,(S\,K\,I):(\varphi_4\to\varphi_5)\to\varphi_5}}{}$$

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* $(S\,I\,I)$ $\mathcal{E}$ fails on unification.

So this rule does not satisfy subject reduction, 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, as formulated in Definition 6.13. 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 6.16 (REPLACEMENT LEMMA) *Let $\langle \mathcal{F},\mathcal{X},\boldsymbol{R}\rangle$ be a* TRS*, and $\mathcal{E}$ and environment for it.*

i) If pp t $\mathcal{E} = \langle \Pi;\, P\rangle$, and for the replacement R there are Γ, A such that $\Gamma;\mathcal{E}\vdash t^{\mathrm{R}}:A$, then there is a substitution S, such that $S\,P = A$, and, for every statement $x{:}C\in\Pi$: $\Gamma;\mathcal{E}\vdash x^{\mathrm{R}}:S\,C$.

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

Proof: By induction on the structure of t.

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

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

$(t\equiv t_1\,t_2)$: Let φ be a type-variable not occurring in any other type. If *pp* $t_1\,t_2$ $\mathcal{E}=\langle\Pi;\,P\rangle$, then for $i=1,2$, there are $\langle\Pi_i;\,P_i\rangle$ (disjoint), such that *pp* t_i $\mathcal{E}=\langle\Pi_i;\,P_i\rangle$. By induction, for $i=1,2$, there is a substitution S_i such that $S_i\,P_i=A_i$, and, for every $x{:}A'\in\Pi_i$, $\Gamma;\mathcal{E}\vdash x^{\mathrm{R}}:S_i\,A'$. 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 $x{:}A'\in\Pi_1\cup\Pi_2$, $\Gamma;\mathcal{E}\vdash x^{\mathrm{R}}:S'\,A'$, and $S'\,\varphi=A$. By property 2.13 there are substitutions S and S_g such that

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

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

ii) $(t\equiv x,F)$: Trivial.

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

Theorem 6.17 (SUBJECT REDUCTION THEOREM) *Let $\langle \mathcal{F},\mathcal{X},\boldsymbol{R}\rangle$ be a* TRS*, and $\mathcal{E}$ an environment for it. If all rules in $\boldsymbol{R}$ are typeable with respect to $\mathcal{E}$, then: if $\Gamma;\mathcal{E}\vdash t:A$ and $t\to t'$, then $\Gamma;\mathcal{E}\vdash t':A$.*

Proof: Let $\mathbf{r}:l\to r$ be the typeable rewrite rule applied in the rewrite step $t\to t'$. We will prove that for every replacement R and type A, if $\Gamma;\mathcal{E}\vdash l^{\mathrm{R}}:A$, then $\Gamma;\mathcal{E}\vdash r^{\mathrm{R}}:A$, which proves the

$$\begin{array}{llcll}
\mathsf{TypeTerm}\ x & \mathcal{E} & \rightarrow & \langle x{:}\varphi\,;\ \varphi\rangle & \\
 & & & \text{where } \varphi \textit{ is fresh} & \\
\mathsf{TypeTerm}\ t_1 t_2 & \mathcal{E} & \rightarrow & S\,\langle \Pi_1 \cup \Pi_2\,;\ \varphi\rangle & \\
 & & & \text{where } \langle \Pi_1\,;\ P_1\rangle \ = \ \mathsf{TypeTerm}\ t_1\ \mathcal{E} & \\
 & & & \qquad\ \ \langle \Pi_2\,;\ P_2\rangle \ = \ \mathsf{TypeTerm}\ t_2\ \mathcal{E} & \\
 & & & \qquad\ \ S \qquad\quad\ = \ \mathit{unify}\ P_1\ P_2 \rightarrow \varphi & \\
 & & & \qquad\ \ \varphi \qquad\quad\ \ \mathit{is}\ \ \mathit{fresh} & \\
\mathsf{TypeTerm}\ F & \mathcal{E} & \rightarrow & \mathsf{Freeze}\,(\mathcal{E}\,F) & (\textit{this is defining occurrence of } F) \\
 & & & \mathit{FreshInstance}\,(\mathcal{E}\,F) & (\textit{otherwise})
\end{array}$$

Figure 9. The algorithm TypeTerm

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 $\Pi;\mathcal{E} \vdash r : P$. Suppose R is a replacement such that $\Gamma;\mathcal{E} \vdash l^{\mathrm{R}} : A$. By Lemma 6.16(*i*) there is a Γ' such that for every $x{:}C \in \Gamma'$, $\Gamma;\mathcal{E} \vdash x^{\mathrm{R}} : C$, and $\Gamma';\mathcal{E} \vdash l : A$. Since $\langle \Pi\,;\ P\rangle$ is a principal pair for l with respect to $\mathcal{E}$, by Definition 6.10 there is a substitution S such that $S\,\langle \Pi\,;\ P\rangle = \langle \Gamma'\,;\ A\rangle$. Since $\Pi;\mathcal{E} \vdash r : P$, by Theorem 6.11 also $\Gamma';\mathcal{E} \vdash r : A$. Then by Lemma 6.16(*ii*) $\Gamma;\mathcal{E} \vdash r^{\mathrm{R}} : A$. □

* 6.5 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 6.8 and 6.13 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 6.18 *Let $\langle \mathcal{F}, \mathcal{X}, \boldsymbol{R}\rangle$ be a* TRS*, and $\mathcal{E}$ and environment for it.*

i) If t is typeable with respect to $\mathcal{E}$, then TypeTerm $t\,\mathcal{E}$ *returns pp t $\mathcal{E}$.*

ii) If TypeTerm $t\,\mathcal{E}$ *returns the pair $\langle \Gamma\,;\ A\rangle$, then pp t $\mathcal{E} = \langle \Gamma\,;\ A\rangle$.*

iii) There is a type assignment with respect to $\mathcal{E}$ for the TRS $\boldsymbol{R}$*, if and only if* TypeRules $\boldsymbol{R}\ \mathcal{E}$.

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 6.1 and 6.2. 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 untypeable 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, in Figure 9.

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 recognised, so that it refuses to substitute these variables by other types.

TypeRule takes care of checking the safeness constraint as given in Definition 6.13, by checking if the unification of left and right-hand sides of a rewrite rule has changed the left-hand side context. It calls on *UnifyContexts* because we need to make sure that the variables have the same types both on the left as on the right.

$$
\begin{array}{llll}
\textsf{TypeRule} & (l \to r) \;\; \mathcal{E} & \to & (S_2\,(S_1\,\Gamma_l)) = \Gamma_l \\
& & & \text{where } \langle \Pi_l\,;\, P_1 \rangle \;=\; \textsf{TypeTerm}\; l\; \mathcal{E} \\
& & & \qquad\;\; \langle \Pi_r\,;\, P_2 \rangle \;=\; \textsf{TypeTerm}\; r\; \mathcal{E} \\
& & & \qquad\;\; S_2 \;=\; \mathit{UnifyContexts}\; (S_1\,\Pi_l)\; (S_1\,\Pi_r) \\
& & & \qquad\;\; S_1 \;=\; \mathit{unify}\; P_1\; P_2, \\
\textsf{TypeRules} & [\,] \quad\;\; \mathcal{E} & \to & \textbf{true} \\
\textsf{TypeRules} & [\mathbf{r} \mid \mathbf{R}] \;\; \mathcal{E} & \to & (\textsf{TypeRule}\; \mathbf{r}\; \mathcal{E}) \;\wedge\; (\textsf{TypeRules}\; \mathbf{R}\; \mathcal{E})
\end{array}
$$

and the procedure that type checks the program:

$$\textsf{TypeTRS} \;\; \langle \mathbf{R} : t \rangle \;\; \mathcal{E} \;\; \to \;\; \textsf{TypeTerm}\; t\; \mathcal{E}, \text{if } \textsf{TypeRules}\; \mathbf{R}\; \mathcal{E}$$

6.6 An example: Combinatory Logic

We now will focus on Curry's Combinatory Logic [21], 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, 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 6.19 (Combinatory Logic [21]) 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.

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

Notice that this defines a *higher-order language* with a *first-order* reduction system: the combinators K, S, and I officially 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.

* 6.7 The relation between CL and the Lambda Calculus

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

Definition 6.20 *i*) For CL, the interpretation of terms in Λ is given by:

$$\begin{array}{lcl}\langle x\rangle_\lambda & = & x\\ \langle \mathsf{S}\rangle_\lambda & = & (\lambda xyz.xz(yz)),\\ \langle \mathsf{K}\rangle_\lambda & = & (\lambda xy.x),\\ \langle \mathsf{I}\rangle_\lambda & = & (\lambda x.x),\\ \langle t_1t_2\rangle_\lambda & = & \langle t_1\rangle_\lambda\langle t_2\rangle_\lambda\end{array}$$

ii) (Bracket abstraction) Fun x t,[2] with $t\in\mathcal{T}_{\text{CL}}$, is defined by induction on the structure of terms:

$$\begin{array}{lcll}\mathsf{Fun}\ x\ x & = & \mathsf{I}, & \\ \mathsf{Fun}\ x\ t & = & \mathsf{K}\ t & (x\ \textit{not in}\ t),\\ \mathsf{Fun}\ x\ (t_1t_2) & = & \mathsf{S}(\mathsf{Fun}\ x\ t_1)(\mathsf{Fun}\ x\ t_2) & (\textit{otherwise})\end{array}$$

iii) The mapping $\llbracket\ \rrbracket_{\text{CL}} : \Lambda\to\mathcal{T}_{\text{CL}}$ is defined by:

$$\begin{array}{lcl}\llbracket x\rrbracket_{\text{CL}} & = & x,\\ \llbracket \lambda x.M\rrbracket_{\text{CL}} & = & \mathsf{Fun}\ x\ \llbracket M\rrbracket_{\text{CL}},\\ \llbracket MN\rrbracket_{\text{CL}} & = & \llbracket M\rrbracket_{\text{CL}}\llbracket N\rrbracket_{\text{CL}}.\end{array}$$

Notice that Fun, that takes a variable and a term in $\mathcal{T}_{\text{CL}}$ and returns a term in $\mathcal{T}_{\text{CL}}$, is only evaluated in the definition of $\llbracket\cdot\rrbracket_{\text{CL}}$ 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 6.21 $(\mathsf{Fun}\ x\ t)\,v\to t[v/x]$.

Proof: By induction on the definition of Fun.

$$\begin{array}{lcllll}(\mathsf{Fun}\ x\ x)\,t & = & \mathsf{I}\,t & \to & t & \\ (\mathsf{Fun}\ x\ t_1)\,t_2 & = & \mathsf{K}\,t_1\,t_2 & \to & t_1 & (x\ \textit{not in}\ t_1)\\ (\mathsf{Fun}\ x\ (t_1\,t_2))\,t_3 & = & \mathsf{S}\,(\mathsf{Fun}\ x\ t_1)\,(\mathsf{Fun}\ x\ t_2)\,t_3 & \to & & \\ & & ((\mathsf{Fun}\ x\ t_1)\,t_3)\,((\mathsf{Fun}\ x\ t_2)\,t_3) & \to & (IH) & \\ & & t_1[t_3/x]\,t_2[t_3/x] & = & (t_1\,t_2)[t_3/x]. & \end{array}$$

□

For the interpretations defined above the following property holds:

Lemma 6.22 ([10]) *i)* $\langle\mathsf{Fun}\ x\ t\rangle_\lambda\to_\beta^*\lambda x.\langle t\rangle_\lambda$

ii) $\langle\llbracket M\rrbracket_{\text{CL}}\rangle_\lambda\twoheadrightarrow_\beta M$.

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

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

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

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

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

a) $M=x$. Since $\langle\llbracket x\rrbracket_{\text{CL}}\rangle_\lambda=\langle x\rangle_\lambda=x$, this is immediate.

[2] Fun is normally called λ^* in the literature.

b) $M = \lambda x.N$. Since $\langle [\![\lambda x.N]\!]_{\text{CL}} \rangle_\lambda = \langle \mathsf{Fun}\ x [\![N]\!]_{\text{CL}} \rangle_\lambda \to_\beta^* \lambda x.\langle [\![N]\!]_{\text{CL}} \rangle_\lambda$ by the previous part, and $\lambda x.\langle [\![N]\!]_{\text{CL}} \rangle_\lambda \to_\beta^* \lambda x.N$ by induction.

c) $M = PQ$. Since $\langle [\![PQ]\!]_{\text{CL}} \rangle_\lambda = \langle [\![P]\!]_{\text{CL}} \rangle_\lambda \langle [\![Q]\!]_{\text{CL}} \rangle_\lambda \to_\beta^* PQ$ 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 $\langle t \rangle_\lambda = (\lambda x_1 \cdots x_n.\langle t' \rangle_\lambda)\langle t_1 \rangle_\lambda \cdots \langle t_n \rangle_\lambda \to_\beta^* \langle t' \rangle_\lambda[\overrightarrow{\langle t_i/x_i \rangle_\lambda}] = \langle t'[\overrightarrow{t_i/x_i}] \rangle_\lambda = \langle u \rangle_\lambda$. □

Example 6.23
$$\begin{array}{rcl} [\![\lambda xy.x]\!]_{\text{CL}} & = & \mathsf{Fun}\ x\ [\![\lambda y.x]\!]_{\text{CL}} \\ & = & \mathsf{Fun}\ x\ (\mathsf{Fun}\ y\ x) \\ & = & \mathsf{Fun}\ x\ (\mathsf{K}\, x) \\ & = & \mathsf{S}\,(\mathsf{Fun}\ x\ \mathsf{K})\,(\mathsf{Fun}\ x\ x) \\ & = & \mathsf{S}\,(\mathsf{K}\,\mathsf{K})\,\mathsf{I} \end{array}$$

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

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 6.24 ([10]) *i*) $\mathsf{S}\,\mathsf{K}$ is a normal form, but $\langle \mathsf{S}\,\mathsf{K} \rangle_\lambda \to_\beta^* \lambda xy.y$;

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 $\langle t \rangle_\lambda \to_\beta^* \lambda c.(\lambda x.xx)\,(\lambda x.xx)$, which does not have a β-normal form, and not even a head-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 $\langle t \rangle_\lambda \to_\beta^* \lambda x.x$.

6.8 Extending CL

Bracket abstraction algorithms like $[\![\ ]\!]_{\text{CL}}$ are used to translate λ-terms to combinator systems, and form, together with the technique of *lambda lifting* the basis of the Miranda [53] compiler. It is possible to define such a translation also for combinator systems that contain other combinators. With some accompanying optimisation rules they provide an interesting example. If in the bracket abstraction we would use the following combinator set on the left:

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

then we could, as in [23], extend the notion of reduction by defining the optimisations on the right. The correctness of these new rules is easy to check.

Notice that, by adding this optimisation, we are stepping outside the realm of λ-calculus by adding pattern matching: 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*.

Also, we now allow reduction of terms starting with S that have *only two* arguments present.

6.9 Type Assignment for CL

We now give the definition of type assignment on combinatory terms, that is a simplified version of the notion of type assignment for TRS we saw above.

Definition 6.25 (TYPE ASSIGNMENT FOR CL) *Type assignment* on terms in CL is defined by the following natural deduction system.

$$(\mathsf{S}):\ \frac{}{\Gamma \vdash \mathsf{S} : (A{\rightarrow}B{\rightarrow}C){\rightarrow}(A{\rightarrow}B){\rightarrow}A{\rightarrow}C} \qquad (Ax):\ \frac{}{\Gamma, x{:}A \vdash x : A}$$

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

$$(\mathsf{I}):\ \frac{}{\Gamma \vdash \mathsf{I} : A{\rightarrow}A}$$

Example 6.26 It is easy to check that the term $\mathsf{S\,K\,I\,I}$ can be assigned the type $\varphi{\rightarrow}\varphi$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_{\mathrm{CL}} \mathsf{S} : B{\rightarrow}C{\rightarrow}D{\rightarrow}\varphi{\rightarrow}\varphi} \quad \dfrac{}{\vdash_{\mathrm{CL}} \mathsf{K} : B}}{\vdash_{\mathrm{CL}} \mathsf{S\,K} : C{\rightarrow}D{\rightarrow}\varphi{\rightarrow}\varphi} \quad \dfrac{}{\vdash_{\mathrm{CL}} \mathsf{I} : C}}{\vdash_{\mathrm{CL}} \mathsf{S\,K\,I} : D{\rightarrow}\varphi{\rightarrow}\varphi} \quad \dfrac{}{\vdash_{\mathrm{CL}} \mathsf{I} : D}}{\vdash_{\mathrm{CL}} \mathsf{S\,K\,I\,I} : \varphi{\rightarrow}\varphi}$$

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

The relation between type assignment in the λ-calculus and that in CL is very strong, as Exercise 6.34 shows. As a corollary of this exercise, we obtain the decidability of type assignment in our system. As a matter of fact, decidability of type assignment for CL was the first of this kind of property proven, by J.R. Hindley [32].

Exercises

Exercise 6.27 Take the following term rewriting system and environment.

$$\begin{array}{lcl@{\qquad}lcl} \mathsf{B}\,x\,y\,z &\rightarrow& x\,(y\,z) & \mathcal{E}\,\mathsf{K} &=& 1{\rightarrow}2{\rightarrow}1, \\ \mathsf{C}\,x\,y\,z &\rightarrow& x\,z\,y & \mathcal{E}\,\mathsf{B} &=& (1{\rightarrow}2){\rightarrow}(3{\rightarrow}1){\rightarrow}3{\rightarrow}2, \\ \mathsf{K}\,x\,y &\rightarrow& x & \mathcal{E}\,\mathsf{C} &=& (1{\rightarrow}2{\rightarrow}3){\rightarrow}2{\rightarrow}1{\rightarrow}3, \\ \mathsf{S}\,x\,y\,z &\rightarrow& x\,z\,(y\,z) & \mathcal{E}\,\mathsf{S} &=& (1{\rightarrow}2{\rightarrow}3){\rightarrow}(1{\rightarrow}2){\rightarrow}1{\rightarrow}3 \end{array}$$

i) Add the following rules to the system above.

$$\begin{array}{lcl} \mathsf{S}\,(\mathsf{K}\,x)\,(\mathsf{K}\,y) &\rightarrow& \mathsf{K}\,(x\,y) \\ \mathsf{S}\,(\mathsf{K}\,x)\,y &\rightarrow& \mathsf{B}\,x\,y \\ \mathsf{S}\,x\,(\mathsf{K}\,y) &\rightarrow& \mathsf{C}\,x\,y \end{array}$$

Show that the system is still typeable using the same environment.

ii) Add now also the rule

$$\mathsf{S}\,(\mathsf{K}\,x)\,\mathsf{I} \rightarrow x$$

Show that the system is no longer typeable using the same environment. What would you have to change to make the system typeable?

iii) What would be a reason, if any, not to add the last rule?

* *Exercise 6.28 If $\Gamma;\mathcal{E} \vdash t : A$, then $S\,\Gamma;\mathcal{E} \vdash t : S\,A$, for every S.*

* *Exercise 6.29* (SOUNDNESS OF *pp*) *Verify that $pp\ t\ \mathcal{E} = \langle \Pi;\, P\rangle$ implies $\Pi;\mathcal{E} \vdash t : P$. You need the previous exercise here, and can assume that weakening is admissible.*

Exercise 6.30 *i) Express the factorial function as a* term rewriting system *and show the necessary* derivations.

ii) Extend the rewrite system with a rule, *and a* single term *(so write a program) that calculates the list of all factorial numbers. Extend the environment appropriately and show that the new rule and the term are typeable.*

Exercise 6.31 *Take the Term Rewriting System*

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

Give an environment that makes these rules typeable (you can use [ ]*); show the corresponding derivations for the last rule.*

Exercise 6.32 *Define*

$$\begin{array}{llllcl}
\mathsf{B} & x & y & z & \to & x\,(y\,z) \\
\mathsf{C} & x & y & z & \to & x\,z\,y \\
\mathsf{W} & x & y & & \to & x\,y\,y
\end{array}$$

Give an environment that makes these rules typeable, and check the result through derivations.

Exercise 6.33 *Check that the optimisation rules given in Section 6.8 are admissible, and check if they introduce any conflict with respect to types.*

* *Exercise 6.34* *Show, by induction on the definition of* Fun, $\llbracket\ \rrbracket_{\mathrm{CL}}$ *and* $\langle\ \rangle_\lambda$, *that*

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

ii) $\Gamma \vdash_{\mathrm{C}} M : A$ *implies* $\Gamma \vdash_{\mathcal{E}_{\mathrm{CL}}} \llbracket M \rrbracket_{\mathrm{CL}} : A$.

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

* *Exercise 6.35* *i) Show that* $\llbracket \lambda xy.xy \rrbracket_{\mathrm{CL}} = \mathsf{S}(\mathsf{S}(\mathsf{KS})(\mathsf{S}(\mathsf{KK})\mathsf{I}))(\mathsf{KI})$.

ii) Show that $\langle \llbracket \lambda xy.xy \rrbracket_{\mathrm{CL}} \rangle_\lambda \to_\beta^* \lambda xy.xy$ *by reducing the first term (you do not need to show each individual step, but could do as many as you like 'in parallel').*

iii) Show that $\emptyset \vdash_{\mathcal{E}_{\mathrm{CL}}} \mathsf{S}(\mathsf{S}(\mathsf{KS})(\mathsf{S}(\mathsf{KK})\mathsf{I}))(\mathsf{KI}) : (A \to B) \to A \to B$.

Exercise 6.36 *Is* $\mathsf{S\,K\,((S\,I\,I)\,(S\,I\,I))}$ *typeable? Motivate your answer.*

7 Basic extensions to the type language

In this section we will briefly discuss a few basic extensions (to, in our case, ML) that can be made to obtain a more expressive programming language, i.e. to add those type features that are considered basic: *data structures*, and *recursive types.*[3]

7.1 Data structures

Two basic notions that we would like to add are *tuples* and *choice*, via the introduction of the type constructors *product* and *sum* (or *disjoint union*) to our type language.

Definition 7.1 The grammar of types is extended as follows:

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

[3] This section is in part based on [47]

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.

We will extend ML with syntactic structure for these type constructs, that act as markers for the introduction or elimination for them.

Definition 7.2 (PAIRING) We extend the calculus with the following constructors

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

with their type assignment rules:

$$(\mathit{Pair}): \frac{\Gamma \vdash E_1 : A \quad \Gamma \vdash E_2 : B}{\Gamma \vdash \langle E_1, E_2 \rangle : A \times B} \qquad (\mathit{left}): \frac{\Gamma \vdash E : A \times B}{\Gamma \vdash \mathit{left}\,(E) : A} \qquad (\mathit{right}): \frac{\Gamma \vdash E : A \times B}{\Gamma \vdash \mathit{right}\,(E) : B}$$

The reduction rules that come with these constructs are:

$$\begin{array}{rcl} \mathit{left}\,\langle E_1, E_2 \rangle & \rightarrow & E_1 \\ \mathit{right}\,\langle E_1, E_2 \rangle & \rightarrow & E_2 \end{array} \qquad E \rightarrow E' \ \Rightarrow \left\{ \begin{array}{rcl} \langle E, E_2 \rangle & \rightarrow & \langle E', E_2 \rangle \\ \langle E_1, E \rangle & \rightarrow & \langle E_1, E' \rangle \\ \mathit{left}\,(E) & \rightarrow & \mathit{left}\,(E') \\ \mathit{right}\,(E) & \rightarrow & \mathit{right}\,(E') \end{array} \right.$$

Notice that these rules are expressed through pattern matching, and that *left* and *right* are *term constructors*, not constants.

We could be tempted to add the rule

$$\langle \mathit{left}\,(E), \mathit{right}\,(E) \rangle \ \rightarrow \ E$$

as well, but a difficulty with this in combination with the two projection rules is that it forms Klop's famous 'Surjective Pairing' counter example that we mentioned in Example 6.1 and destroys confluence, an arguably very desirable property for programming languages.

Definition 7.3 ((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:

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

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

Notice that the additional syntactic structure as added to the programming language acts as a syntactic 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 constructors are:

$$\begin{array}{rcl} \mathsf{case}\,(\mathsf{inj{\cdot}l}\,(E_1), E_2, E_3) & \rightarrow & E_2\ E_1 \\ \mathsf{case}\,(\mathsf{inj{\cdot}r}\,(E_1), E_2, E_3) & \rightarrow & E_3\ E_1 \end{array} \qquad E \rightarrow E' \ \Rightarrow \left\{ \begin{array}{rcl} \mathsf{case}\,(E, E_2, E_3) & \rightarrow & \mathsf{case}\,(E', E_2, E_3) \\ \mathsf{case}\,(E_1, E, E_3) & \rightarrow & \mathsf{case}\,(E_1, E', E_3) \\ \mathsf{case}\,(E_1, E_2, E) & \rightarrow & \mathsf{case}\,(E_1, E_2, E') \\ \mathsf{inj{\cdot}l}\,(E) & \rightarrow & \mathsf{inj{\cdot}l}\,(E') \\ \mathsf{inj{\cdot}r}\,(E) & \rightarrow & \mathsf{inj{\cdot}r}\,(E') \end{array} \right.$$

Notice that application is used on the right-hand side of these rules and that also these rules are expressed through pattern matching.

7.2 Recursive types

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.

To be able to express recursive types properly, some computer programming languages have a unit type as a type that holds no information and allows only one value; it can be seen as the type of 0-tuples, i.e. the product of no types. It is also used to specify the argument type of a function that does not require arguments; then we write $E : A$ rather than $E : \mathsf{unit} \rightarrow A$. In the functional programming languages Haskell [33], and Clean [13], the unit type is called $()$ and its only value is also $()$, reflecting the 0-tuple interpretation. In SML (Standard ML [31, 44]), the type is called unit but the value is written as $()$. Using this approach here, we extend the syntax with '()', the type language with 'unit' and add the rule

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

but under the condition that unit can only be assigned to $()$, and vice versa.

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

$$A \ = \ \mathsf{unit} + (B \times A);$$

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, i.e. to extend the language of types in such a way that there exists a type A such that $A = \mathsf{unit} + (B \times A)$; remark that we cannot solve this without such an extension.

Definition 7.4 (Recursive types) The grammar of types is extended with:

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

We define a relation $=_\mu$ on types as the smallest equivalence relation containing

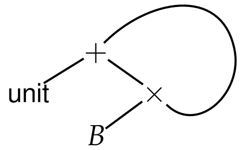
$$\mu X.A \ =_\mu \ A[\mu X.A/X]$$

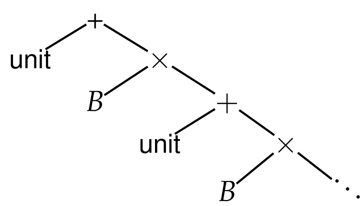
Then the '*list* B' type (or $[B]$) that is a solution to the above equation is

$$\mu X.\mathsf{unit} + (B \times X)$$

because
$$\begin{array}{rcll} [B] & \triangleq & \mu X.\mathsf{unit} + (B \times X) & \\ & =_\mu & (\mathsf{unit} + (B \times X))\,[\mu X.\mathsf{unit} + (B \times X)/X] & \\ & = & \mathsf{unit} + (B \times (\mu X.\mathsf{unit} + (B \times X))) & \triangleq \ \mathsf{unit} + (B \times [B]) \end{array}$$

which corresponds to the graphs:





We can see recursive 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 unfolding coincide. Conditions on recursive types rule out meaningless types, such as $\mu X.X$, which (infinite) unfolding is not well defined.

There are two ways to deal with recursive types in programming, either by having syntactic markers for the $=_\mu$ steps or not.

7.3 The equi-recursive approach

We first look at the variant that does not use syntactic markers.

Definition 7.5 (EQUI-RECURSIVE TYPE ASSIGNMENT) In the *equi-recursive* approach, two equal types can be used interchangeably: this is formalised by introducing a new typing rule:

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

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

Example 7.6 A term now has a $[B]$ type if either it is of the shape $\mathsf{inj{\cdot}l}\,()$ or $\mathsf{inj{\cdot}r}\,\langle a,b\rangle$:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash () : \mathsf{unit}}\,(\mathsf{unit})}{\Gamma \vdash \mathsf{inj{\cdot}l}\,() : \mathsf{unit} + (B \times [B])}\,(\mathsf{inj{\cdot}l})}{\Gamma \vdash \mathsf{inj{\cdot}l}\,() : [B]}\,(\mu) \qquad \dfrac{\dfrac{\dfrac{\Gamma \vdash E_1 : B \qquad \Gamma \vdash E_2 : [B]}{\Gamma \vdash \langle E_1,E_2\rangle : B \times [B]}\,(\mathit{Pair})}{\Gamma \vdash \mathsf{inj{\cdot}r}\,\langle E_1,E_2\rangle : \mathsf{unit} + (B \times [B])}\,(\mathsf{inj{\cdot}r})}{\Gamma \vdash \mathsf{inj{\cdot}r}\,\langle E_1,E_2\rangle : [B]}\,(\mu)$$

Assuming numbers and pre-fix addition, we can express the function that calculates the length of a list by:

$$LL \ = \ \mathsf{fix}\, ll\,.\lambda \mathit{list}.\mathsf{case}\,(\mathit{list},\lambda x.0,\lambda x.+\ 1\ (ll(\mathit{right}\ (x))))$$

Notice that now

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

Using I for the type for numbers, for the term above we can construct the derivation (hiding obsolete statements in contexts) in Figure 10.

This approach to recursive types is known as the *equi-recursive* approach [2, 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 [48]. Its main disadvantage is that, in the presence of equi-recursive types, apparently meaningless programs have types.

Example 7.7 We can type self-application; the term $\lambda x.xx$ can be assigned the type $\mu X.X{\to}\varphi$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}\mu X.X{\to}\varphi \vdash x : \mu X.X{\to}\varphi}\,(Ax)}{x{:}\mu X.X{\to}\varphi \vdash x : (\mu X.X{\to}\varphi){\to}\varphi}\,(\mu) \qquad \dfrac{}{x{:}\mu X.X{\to}\varphi \vdash x : \mu X.X{\to}\varphi}\,(Ax)}{x{:}\mu X.X{\to}\varphi \vdash xx : \varphi}\,({\to}E)}{\vdash \lambda x.xx : (\mu X.X{\to}\varphi){\to}\varphi}\,({\to}I)}{\vdash \lambda x.xx : \mu X.X{\to}\varphi}\,(\mu)$$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\dfrac{}{\Gamma''\vdash l:[\varphi]}}{\Gamma''\vdash l:\mathsf{unit}+(\varphi\times[\varphi])}\,(\mu)
\quad
\dfrac{\dfrac{}{\Gamma'',x{:}\mathsf{unit}\vdash 0:I}}{\Gamma''\vdash \lambda x.0:\mathsf{unit}\to I}
\quad
\begin{array}{c}
\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma'\vdash +:I\to I\to I}\quad\dfrac{}{\Gamma'\vdash 1:I}}{\Gamma'\vdash +\,1:I\to I}\quad\dfrac{\dfrac{}{\Gamma'\vdash ll:[\varphi]\to I}\quad\dfrac{\dfrac{}{\Gamma'\vdash x:\varphi\times[\varphi]}}{\Gamma'\vdash \mathit{right}\ (x):[\varphi]}}{\Gamma'\vdash ll(\mathit{right}\ (x)):I}}{\Gamma'\vdash +\,1\,(ll(\mathit{right}\ (x))):I}}{\Gamma''\vdash \lambda x.+\,1\,(ll(\mathit{right}\ (x))):(\varphi\times[\varphi])\to I}\\
\vdots
\end{array}
}{\Gamma''\vdash \mathsf{case}\,(l,\lambda x.0,\lambda x.+\,1\,(ll(\mathit{right}\ (x)))):I}
}{\Gamma,ll{:}[\varphi]\to I\vdash \lambda\,\mathit{list}.\mathsf{case}\,(l,\lambda x.0,\lambda x.+\,1\,(ll(\mathit{right}\ (x)))):[\varphi]\to I}
}{\Gamma\vdash \mathsf{fix}\,ll.\lambda\,\mathit{list}.\mathsf{case}\,(l,\lambda x.0,\lambda x.+\,1\,(ll(\mathit{right}\ (x)))):[\varphi]\to I}
}{}
\quad
\dfrac{\dfrac{\dfrac{\Gamma\vdash a:\varphi\quad\Gamma\vdash b:[\varphi]}{\Gamma\vdash\langle a,b\rangle:\varphi\times[\varphi]}}{\Gamma\vdash \mathsf{inj{\cdot}r}\,\langle a,b\rangle:\mathsf{unit}+(\varphi\times[\varphi])}}{\Gamma\vdash \mathsf{inj{\cdot}r}\,\langle a,b\rangle:[\varphi]}\,(\mu)
}{\Gamma\vdash (\mathsf{fix}\,ll.\lambda\,\mathit{list}.\mathsf{case}\,(l,\lambda x.0,\lambda x.+\,1\,(ll(\mathit{right}\ (x)))))\,(\mathsf{inj{\cdot}r}\,\langle a,b\rangle):I}
$$

Figure 10. A derivation for $\vdash (\mathsf{fix}\,ll.\lambda\,\mathit{list}.\mathsf{case}\,(l,\lambda x.0,\lambda x.+\,1\,(ll(\mathit{right}\ (x)))))\,\langle a,b\rangle:I$ (where $\Gamma' = \Gamma'',x{:}\varphi\times[\varphi]$, and $\Gamma'' = \Gamma,ll{:}[\varphi]\to I,l{:}[\varphi]$).

7.4 The iso-recursive approach

In the *iso-recursive* approach to recursive types, the above is not possible. It prohibits the use of the non-syntax sensitive rule (μ), and adds syntactic markers for the two conversions $\mu X.A = A[\mu X.A/X]$ and $A[\mu X.A/X] = \mu X.A$.

Definition 7.8 The syntax is extended by

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

and we add the reduction rules

$$\mathsf{unfold}\,(\mathsf{fold}\,(E)) \ \to \ E \qquad E\to E' \ \Rightarrow \ \begin{cases} \mathsf{unfold}\,(E) \ \to \ \mathsf{unfold}\,(E') \\ \mathsf{fold}\,(E) \ \to \ \mathsf{fold}\,(E') \end{cases}$$

and the type assignment rules:

$$(\mathsf{fold}):\ \frac{\Gamma\vdash E:A[\mu X.A/X]}{\Gamma\vdash \mathsf{fold}\,(E):\mu X.A} \qquad (\mathsf{unfold}):\ \frac{\Gamma\vdash E:\mu X.A}{\Gamma\vdash \mathsf{unfold}\,(E):A[\mu X.A/X]}$$

Notice that it is possible to apply (unfold) directly after (fold), but that would be a waste of effort; however, as a result of reduction such a derivation can be constructed. We therefore also have the reduction rule $\mathsf{unfold}\,(\mathsf{fold}\,(E))\to E$.

Example 7.9 A term now has a $[B]$ type if either it is of the shape $\mathsf{fold}\,(\mathsf{inj{\cdot}l}\,())$ or $\mathsf{fold}\,(\mathsf{inj{\cdot}r}\,\langle a,b\rangle)$:

$$
\dfrac{\dfrac{\dfrac{}{\Gamma\vdash ():\mathsf{unit}}\,(\mathsf{unit})}{\Gamma\vdash \mathsf{inj{\cdot}l}\,():\mathsf{unit}+(B\times[B])}\,(\mathsf{inj{\cdot}l})}{\Gamma\vdash \mathsf{fold}\,(\mathsf{inj{\cdot}l}\,()):[B]}\,(\mathsf{fold})
\qquad
\dfrac{\dfrac{\dfrac{\Gamma\vdash E_1:B\quad \Gamma\vdash E_2:[B]}{\Gamma\vdash\langle E_1,E_2\rangle:B\times[B]}\,(\mathsf{Pair})}{\Gamma\vdash \mathsf{inj{\cdot}r}\,\langle E_1,E_2\rangle:\mathsf{unit}+(B\times[B])}\,(\mathsf{inj{\cdot}r})}{\Gamma\vdash \mathsf{fold}\,(\mathsf{inj{\cdot}r}\,\langle E_1,E_2\rangle):[B]}\,(\mathsf{fold})
$$

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

$$\dfrac{\dfrac{\dfrac{}{x{:}\mu X.X{\to}\varphi \vdash x : \mu X.X{\to}\varphi}}{x{:}\mu X.X{\to}\varphi \vdash \mathsf{unfold}(x) : (\mu X.X{\to}\varphi){\to}\varphi} \qquad \dfrac{}{x{:}\mu X.X{\to}\varphi \vdash x : \mu X.X{\to}\varphi}}{\dfrac{x{:}\mu X.X{\to}\varphi \vdash \mathsf{unfold}(x)x : \varphi}{\dfrac{\vdash \lambda x.\mathsf{unfold}(x)x : (\mu X.X{\to}\varphi){\to}\varphi}{\vdash \mathsf{fold}(\lambda x.\mathsf{unfold}(x)x) : \mu X.X{\to}\varphi}}}$$

So, in a sense, in the iso-recursive approach we can replace a recursive type by its folding or unfolding only 'on demand', i.e. when specified in the term.

* 7.5 Recursive data types

Remark that the two added rules above depend on the equation $\mu X.A = A[\mu X.A/X]$ which is itself only implicitly part of the inferred statements, so a better representation would be:

$$(\mathsf{fold}_{\mu X.A}) : \frac{\Gamma \vdash E : A[\mu X.A/X]}{\Gamma \vdash \mathsf{fold}_{\mu X.A}(E) : \mu X.A} \qquad (\mathsf{unfold}_{\mu X.A}) : \frac{\Gamma \vdash E : \mu X.A}{\Gamma \vdash \mathsf{unfold}_{\mu X.A}(E) : A[\mu X.A/X]}$$

since the equation $\mu X.A = A[\mu X.A/X]$ is of course implicit in $\mu X.A$. Then each recursive type has its own fold and unfold statements.

If we now add identifiers to recursive types, and express the $[A]$ type constructor as a solution to the type equation

$$A \;=\; \mathsf{unit} + (B \times A);$$

we have the type assignment rules

$$(\mathsf{fold}_{[B]}) : \frac{\Gamma \vdash E : \mathsf{unit}{+}(B \times [B])}{\Gamma \vdash \mathsf{fold}_{[B]}(E) : [B]} \qquad (\mathsf{unfold}_{[B]}) : \frac{\Gamma \vdash E : [B]}{\Gamma \vdash \mathsf{unfold}_{[B]}(E) : \mathsf{unit} + (B \times [B])}$$

The $(\mathsf{fold}_{[B]})$ rule now expresses: if we have derived that a term E has type $\mathsf{unit} + (B \times [B])$ (typically by deriving either unit and using (inj·l) or deriving $B \times [B]$ and using (inj·r)), then we 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 we can unfold that information, and say that E has type $\mathsf{unit} + (B \times [B])$ (this is typically for used for a variable x, where $x{:}[B]$ is assumed); we then have access to the types unit and $B \times [B]$, and can do a case analysis.

For the list type constructor declared as above, the empty list (of type $[B]$) is written

$$\mathsf{fold}_{[B]}(\mathsf{inj{\cdot}l}(\;))$$

A list l of type $[\varphi]$ is deconstructed by

$$\mathsf{case}\,(\mathsf{unfold}_{[B]}\; l, \lambda x.\ldots, \lambda X.\mathsf{let\ hd} = \mathit{left}\ (x)\ \mathsf{in\ let\ tl} = \mathit{right}\ (x)\ \mathsf{in} \ldots)$$

More generally, recursive (data) types can be defined via:

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

where C is the user-defined *type constructor*, defined over a number of type variables $\vec{\varphi}$, and $A_{\mathrm{C}}[\vec{\varphi}]$ is a type which main structure is A and 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 $\mathrm{C}\,\vec{\varphi}$ and $A_{\mathrm{C}}[\vec{\varphi}]$ are distinct types, but it is possible to convert one into the other via *folding* and *unfolding*.

Definition 7.10 The syntax is extended by

$$E \ ::= \ \cdots \mid \mathsf{fold}_{\mathrm{C}}(E) \mid \mathsf{unfold}_{\mathrm{C}}(E)$$

and we add the reduction rules

$$\mathsf{unfold}_{\mathrm{C}}(\mathsf{fold}_{\mathrm{C}}(E)) \ \to \ E \qquad E \to E' \ \Rightarrow \ \begin{cases} \mathsf{unfold}_{\mathrm{C}}(E) \ \to \ \mathsf{unfold}_{\mathrm{C}}(E') \\ \mathsf{fold}_{\mathrm{C}}(E) \ \to \ \mathsf{fold}_{\mathrm{C}}(E') \end{cases}$$

and the type assignment rules

$$(\mathsf{fold}_{\mathrm{C}}): \ \frac{\Gamma \vdash E : A_{\mathrm{C}}[\vec{B}]}{\Gamma \vdash \mathsf{fold}_{\mathrm{C}}(E) : \mathrm{C}\,\vec{B}} \qquad (\mathsf{unfold}_{\mathrm{C}}): \ \frac{\Gamma \vdash E : \mathrm{C}\,\vec{B}}{\Gamma \vdash \mathsf{unfold}_{\mathrm{C}}(E) : A_{\mathrm{C}}[\vec{B}]}$$

for every type definition $\mathrm{C}\,\vec{\varphi} = A_{\mathrm{C}}[\vec{\varphi}]$.

Notice that we have a kind of polymorphism this way: the type is declared using variables, whereas the inference rule is specified with for every instance.

Converting $\mathrm{C}\,\vec{B}$ to its unfolding $A_{\mathrm{C}}[\vec{B}]$ – or folding $A_{\mathrm{C}}[\vec{B}]$ to $\mathrm{C}\,\vec{B}$ – requires an explicit use of $\mathsf{fold}_{\mathrm{C}}$ or $\mathsf{unfold}_{\mathrm{C}}$, that is, an explicit syntax in the calculus, making a recursive type-conversion only possible on *call*, i.e. if a $\mathsf{fold}_{\mathrm{C}}$ or $\mathsf{unfold}_{\mathrm{C}}$ 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. 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 7.11 In this setting, the (silent) μ-conversion in the definition of LL in Example 7.9 are now made explicit, and LL becomes

$$LL \ = \ \mathsf{fix}\, ll\,.\lambda\, list.\mathsf{case}\,(\mathsf{unfold}_{[B]}\,(list),\lambda x.0,\lambda x.{+1}\ (ll(right\ (x))))$$

Notice that now

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

and

$$\begin{array}{ll}
(\mathsf{fix}\, ll\,.\lambda\, list.\mathsf{case}\,(\mathsf{unfold}_{[B]}(list),\lambda x.0,\lambda x.{+1}\ (ll(right\ (x)))))\,(\mathsf{fold}_{[B]}(\mathsf{inj{\cdot}l}(\ ))) & \to \\
(\lambda\, list.\mathsf{case}\,(\mathsf{unfold}_{[B]}(list),\lambda x.0,\lambda x.{+1}\ (LL\,(right\ (x)))))\,(\mathsf{fold}_{[B]}(\mathsf{inj{\cdot}l}(\ ))) & \to \\
\mathsf{case}\,(\mathsf{unfold}_{[B]}(\mathsf{fold}_{[B]}(\mathsf{inj{\cdot}l}(\ ))),\lambda x.0,\lambda x.{+1}\ (LL\,(right\ (x)))) & \to \\
\mathsf{case}\,(\mathsf{inj{\cdot}l}(),\lambda x.0,\lambda x.{+1}\ (LL\,(right\ (x)))) \ \to \ (\lambda x.0)\,() \ \to \ 0 &
\end{array}$$

* 7.6 Algebraic datatypes

In ML and Haskell, structural products and sums are defined via iso-recursive types, yielding so-called *algebraic data types* [14]. The idea is to avoid requiring both a (type) name and a (field or tag) number, as in $\mathsf{fold}\,(\mathsf{inj{\cdot}l}())$. Instead, it would be desirable to mention a single name, as in [ ] for the empty list. This is permitted by *algebraic data type* declarations.

Definition 7.12 An algebraic data type constructor C is introduced via a *record* type definition:

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

or the or *variant* type definition:

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

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

As before, these definitions are interpreted polymorphically.

For readability, we normally write ℓ for $\ell\,()$ (so when E is empty in $\ell\ E$), so the label needs no arguments. The implicit type of the label ℓ_i is $A_i[\vec{\varphi}]\to \mathrm{C}\,\vec{\varphi}$; we can in fact also allow the label to be parameterless, as in the definition

$$\mathsf{Bool} \;=\; \mathsf{True} : \mathsf{unit} + \mathsf{False} : \mathsf{unit}$$

which we normally write as

$$\mathsf{Bool} \;=\; \mathsf{True} + \mathsf{False}$$

Definition 7.13 The *record* type definition

$$\mathrm{C}\,\vec{\varphi} \;=\; \Pi_{i=1}^{k}\,\ell_i : A_i[\vec{\varphi}]$$

introduces the constructors ℓ_i for $1 \leq i \leq k$ and $\mathsf{build}_{\mathrm{C}}$, with the following rules:

$$(\ell_i) : \frac{\Gamma \vdash E : \mathrm{C}\,\vec{B}}{\Gamma \vdash \ell_i\, E : A_i[\vec{B}]}\ (1 \leq i \leq k) \qquad (\mathsf{build}_{\mathrm{C}}) : \frac{\Gamma \vdash E_1 : A_1[\vec{B}] \quad \cdots \quad \Gamma \vdash E_k : A_k[\vec{B}]}{\Gamma \vdash \mathsf{build}_{\mathrm{C}}\ E_1 \ldots E_k : \mathrm{C}\,\vec{B}}$$

so the labels act as projection functions into the product type, defined as term constants.

Notice that, since the labels are constants, we could have defined

$$(\ell_i) : \frac{}{\Gamma \vdash \ell_i : \mathrm{C}\,\vec{B} \to A_i[\vec{B}]}\ (1 \leq i \leq k) \qquad (\mathsf{build}_{\mathrm{C}}) : \frac{}{\Gamma \vdash \mathsf{build}_{\mathrm{C}} : A_1[\vec{B}] \to \cdots \to A_k[\vec{B}] \to \mathrm{C}\,\vec{B}}$$

Example 7.14 In this setting, pairing can be expressed via the product type

$$\langle\rangle\ \varphi_1\ \varphi_2 \;=\; \mathit{left} : \varphi_1 \times \mathit{right} : \varphi_2$$

and the rules

$$\frac{\Gamma \vdash E : \langle\rangle\ A_1\ A_2}{\Gamma \vdash \mathit{left}\, E : A_1} \qquad \frac{\Gamma \vdash E : \langle\rangle\ A_1\ A_2}{\Gamma \vdash \mathit{right}\, E : A_2} \qquad \frac{\Gamma \vdash E_1 : A_1 \quad \Gamma \vdash E_2 : A_2}{\Gamma \vdash \mathsf{build}_{\langle\rangle}\ E_1\ E_2 : \langle\rangle\ A_1\ A_2}$$

Of course an in-fix notation would give better readability: $\Gamma \vdash \langle E_1, E_2\rangle : \langle A_1, A_2\rangle$. Notice that now *left* and *right* are term constants, not term constructors.

Definition 7.15 The *variant* type definition

$$\mathrm{C}\,\vec{\varphi} \;=\; \Sigma_{i=1}^{k}\,\ell_i : A_i[\vec{\varphi}]$$

introduces the constructors ℓ_i (with $1 \leq i \leq k$) and $\mathsf{case}_{\mathrm{C}}$, typeable via the rules:

$$(\ell_i) : \frac{\Gamma \vdash E : A_i[\vec{B}]}{\Gamma \vdash \ell_i\ E : \mathrm{C}\,\vec{B}}\ (1 \leq i \leq k) \quad (\mathsf{case}_{\mathrm{C}}) : \frac{\Gamma \vdash E : \mathrm{C}\,\vec{B} \quad \Gamma \vdash E_1 : A_1[\vec{B}] \to D\ \cdots\ \Gamma \vdash E_k : A_k[\vec{B}] \to D}{\Gamma \vdash \mathsf{case}_{\mathrm{C}}\ (E,\ E_1,\ \ldots,\ E_k) : D}$$

(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 $\mathsf{case}_{\mathrm{C}}\,(E, E_1, \ldots, E_k)$ when $k > 0$, and $\mathrm{C}\,\vec{\varphi} = \Sigma_{i=1}^{k}\,\ell_i : A_i[\vec{\varphi}]$, thus avoiding to label case.

We can now give the type declaration for lists as

$$[\varphi] \;=\; [\,] : \mathsf{unit}\ +\ \mathsf{Cons} : \varphi \times [\varphi]$$

This gives rise to the rules

$$([\,]) : \frac{}{\Gamma \vdash [\,] : [B]} \qquad (\mathsf{Cons}) : \frac{\Gamma \vdash E : B \times [B]}{\Gamma \vdash \mathsf{Cons}\ E : [B]}$$

$$(\mathsf{case}_{[B]}) : \frac{\Gamma \vdash E_1 : [B] \quad \Gamma \vdash E_2 : \mathsf{unit} \to D \quad \Gamma \vdash E_3 : (B \times [B]) \to D}{\Gamma \vdash \mathsf{case}_{[B]}\,(E_1,\ E_2,\ E_3) : D}$$

Notice that here Cons and [ ] act as fold, and the rule (case) as unfold; also, we could have used $\Gamma \vdash E_2 : \varphi'$ in the last rule.

In this setting, our example becomes:

$$(\mathsf{fix}\, ll\,.\,\lambda list.\mathsf{case}_{[\varphi]}\ (list,\ \lambda x.0,\ \lambda x.+1\ (ll(right\ (x)))\ ))\,(\mathsf{Cons}\langle a,b\rangle)$$

or

$$(\mathsf{fix}\, ll\,.\,\lambda list.\mathsf{case}\ (list,\ \mathsf{Nil}:\lambda x.0,\ \mathsf{Cons}:\lambda x.+1\ (ll(right\ (x)))\ ))\,(\mathsf{Cons}\langle a,b\rangle)$$

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.

Exercises

Exercise 7.16 Using Example 7.7, find a type for $(\lambda x.xx)\,(\lambda x.xx)$.

Exercise 7.17 Similar to the previous exercise, find a type for $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$.

Exercise 7.18 Give the derivation for

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

Exercise 7.19 Using the equi-recursive approach and the type $\mu X.X{\rightarrow}(\varphi{\rightarrow}\varphi){\rightarrow}\varphi$*, Turing's fixed-point combinator* $(\lambda xy.y(xxy))(\lambda xy.y(xxy))$ *is typeable.*

Give a variant of this term that would be typeable in the iso-recursive approach and show that it is a fixed-point combinator.

8 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 [16, 18, 19, 11, 17, 20, 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 now turns out to be 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, 34], the trivial one being to do without intersection types at all, and use Curry types.

We will see that intersection types are strongly linked to (approximation) semantics, and form a powerful tool to study semantics of a variety of calculi.

8.1 Intersection types

Intersection types are defined by extending Curry types with the type constructor '$\cap$'; we limit the occurrence of intersection types in arrow types to the left-hand side, so have to use a two-level grammar.

Definition 8.1 (STRICT TYPES) *i*) The set $\mathcal{T}_s$ of intersection types is defined by the grammar:

$$\begin{array}{rcll} A & ::= & \varphi \mid (\sigma{\rightarrow}A) & \qquad (\textit{strict types}) \\ \sigma,\tau & ::= & (A_1\cap\cdots\cap A_n) \quad (n\geq 0) & \qquad (\textit{intersection types}) \end{array}$$

ii) On $\mathcal{T}_s$, the relation $\leq$ (induced by intersection) is defined as the smallest relation satisfying:

$$\begin{array}{rcll} \forall 1 \leq i \leq n \; [A_1 \cap \cdots \cap A_n & \leq & A_i] & (n \geq 1) \\ \forall 1 \leq i \leq n \; [\sigma \leq A_i] & \Rightarrow & \sigma \leq A_1 \cap \cdots \cap A_n & (n \geq 0) \\ \sigma \leq \tau \leq \rho & \Rightarrow & \sigma \leq \rho & \end{array}$$

iii) We define the equivalence relation $\sim$ on types by:

$$\begin{array}{rcl} \sigma \leq \tau \leq \sigma & \Rightarrow & \sigma \sim \tau \\ \sigma \sim \tau \wedge A \sim B & \Rightarrow & \sigma \rightarrow A \sim \tau \rightarrow B \end{array}$$

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 '$\rightarrow$', so $C \cap D \rightarrow C \rightarrow D$ stands for $((C \cap D) \rightarrow (C \rightarrow D))$.

We will write $\cap_n A_i$ for $A_1 \cap \cdots \cap A_n$, and use $\top$[4] 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.

The type $\top$ is typically used to type a (sub-)term that will disappear during reduction; we see it therefore as a 'don't care' type, and use it for terms which type is irrelevant (or non-existing) for the typing of the term under consideration.

Notice that, by definition, in $\cap_n A_i$, all A_i are strict; sometimes we will deviate from this by writing also $\sigma \cap \tau$; if $\sigma = \cap_n A_i$ and $\tau = \cap_m B_j$, then

$$\sigma \cap \tau \;=\; A_1 \cap \cdots \cap A_n \cap B_1 \cap \cdots \cap B_m$$

Definition 8.2 (Contexts) *i*) A *statement* is an expression of the form $M : \sigma$, where M is the *subject* and σ is the *predicate* of $M : \sigma$.

ii) A *context* Γ is a set of statements with (distinct) variables as subjects, normally written as a list of statements.

iii) The relations $\leq$ and $\sim$ are extended to contexts by:

$$\begin{array}{rcl} \Gamma \leq \Gamma' & \Longleftrightarrow & \forall x{:}\tau \in \Gamma' \; \exists x{:}\sigma \in \Gamma \; (\sigma \leq \tau) \\ \Gamma \sim \Gamma' & \Longleftrightarrow & \Gamma \leq \Gamma' \leq \Gamma. \end{array}$$

iv) Given two contexts Γ_1 and Γ_2, we define the contexts $\Gamma_1 \cap \Gamma_2$ as follows:

$$\begin{array}{rl} \Gamma_1 \cap \Gamma_2 \;\triangleq & \{ x{:}\sigma \cap \tau \mid x{:}\sigma \in \Gamma_1 \wedge x{:}\tau \in \Gamma_2 \} \;\cup \\ & \{ x{:}\sigma \mid x{:}\sigma \in \Gamma_1 \wedge x \notin \Gamma_2 \} \cup \{ x{:}\tau \mid x{:}\tau \in \Gamma_2 \wedge x \notin \Gamma_1 \} \end{array}$$

and write $\cap_{\underline{n}} \Gamma_i$ for $\Gamma_1 \cap \cdots \cap \Gamma_n$, and $\Gamma \cap x{:}\sigma$ for $\Gamma \cap \{x{:}\sigma\}$, and also use the notation of Definition 2.1.

8.2 Intersection type assignment

The intersection type assignment system is defined as follows.

Definition 8.3 *Strict type assignment* and *strict derivations* are defined by the following inference system (where all types displayed are strict, except σ in the derivation rules $(\rightarrow I)$ and $(\rightarrow E)$):

$$(Ax): \frac{}{\Gamma, x{:}\cap_n A_i \vdash x : A_i} \; (n \geq 1) \qquad (\cap I): \frac{\Gamma \vdash M : A_1 \quad \cdots \quad \Gamma \vdash M : A_n}{\Gamma \vdash M : \cap_n A_i} \; (n \geq 0)$$

$$(\rightarrow I): \frac{\Gamma, x{:}\sigma \vdash M : B}{\Gamma \vdash \lambda x.M : \sigma \rightarrow B} \qquad (\rightarrow E): \frac{\Gamma \vdash M : \sigma \rightarrow B \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : B}$$

We write $\Gamma \vdash_\cap M : \sigma$, if this is derivable using a strict derivation.

[4] In the literature, the symbol ω is often used.

Notice that in $\Gamma \vdash_{\cap} M : \sigma$ the context can contain types that are not strict. Moreover, in this system we can type the approximants; rule $(\cap I)$ allows for $\perp$ to occur inside M for the case $n = 0$ (see also Lemma 8.29).

For this notion of type assignment, the following properties hold:

Lemma 8.4 (Generation Lemma) *i)* $\Gamma \vdash_{\cap} MN : A \iff \exists \sigma \in \mathcal{T}_s\, (\Gamma \vdash_{\cap} M : \sigma{\to}A \wedge \Gamma \vdash_{\cap} N : \sigma)$.
ii) $\Gamma \vdash_{\cap} \lambda x.M : A \iff \exists \sigma, B\, (A \ = \sigma{\to}B \wedge \Gamma, x{:}\sigma \vdash_{\cap} M : B)$.

Also the properties of weakening and strengthening hold:

Lemma 8.5 *i) If* $\Gamma \vdash_{\cap} M : \sigma$*, and* $\Gamma' \supseteq \Gamma$*, then* $\Gamma' \vdash_{\cap} M : \sigma$*.*
ii) If $\Gamma \vdash_{\cap} M : \sigma$*, then* $\{\, x{:}\tau \mid x{:}\tau \in \Gamma \wedge x \in fv(M)\,\} \vdash_{\cap} M : \sigma$*.*

Notice that, by our extension of Barendregt's convention to judgements, in $\Gamma' \vdash M : \sigma$ no term variable bound in M can occur in Γ'.

We can now derive some unexpected results:

Example 8.6 In this system, we can derive both $\emptyset \vdash_{\cap} (\lambda xyz.xz(yz))(\lambda ab.a) : \top{\to}A{\to}A$ and $\emptyset \vdash_{\cap} \lambda yz.z : \top{\to}A{\to}A$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : A{\to}\top{\to}A} \quad \dfrac{}{\Gamma \vdash z : A}}{\Gamma \vdash xz : \top{\to}A} \qquad \dfrac{}{\Gamma \vdash yz : \top}}{x{:}A{\to}\top{\to}A, y{:}\top, z{:}A \vdash xz(yz) : A}}{x{:}A{\to}\top{\to}A, y{:}\top \vdash \lambda z.xz(yz) : A{\to}A}}{x{:}A{\to}\top{\to}A \vdash \lambda yz.xz(yz) : \top{\to}A{\to}A}}{\emptyset \vdash \lambda xyz.xz(yz) : (A{\to}\top{\to}A){\to}\top{\to}A{\to}A} \quad \dfrac{\dfrac{\dfrac{}{a{:}A, b{:}\top \vdash a : A}}{a{:}A \vdash \lambda b.a : \top{\to}A}}{\emptyset \vdash \lambda ab.a : A{\to}\top{\to}A}}{\emptyset \vdash (\lambda xyz.xz(yz))\,(\lambda ab.a) : \top{\to}A{\to}A}$$

(where $\Gamma = x{:}A{\to}\top{\to}A, y{:}\top, z{:}A$) and

$$\dfrac{\dfrac{\dfrac{}{z{:}A, y{:}\top \vdash z : A}}{y{:}\top \vdash \lambda z.z : A{\to}A}}{\emptyset \vdash \lambda yz.z : \top{\to}A{\to}A}$$

Notice that, by using $\Gamma = x{:}A{\to}\top{\to}A, y{:}B, z{:}A$ in the first derivation above, we could as well have derived $\emptyset \vdash_{\cap} (\lambda xyz.xz(yz))\,(\lambda ab.a) : B{\to}A{\to}A$, for any Curry types A and B; as we have seen in Example 2.3, this is not possible in Curry's system.

8.3 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 for Theorem 2.4, and will follow below; first we give an intuitive 'cut and paste' 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 $\Gamma \vdash_{\cap} (\lambda x.M)\,N : A$. Since A is not an intersection, 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 $\Gamma \vdash_{\cap} \lambda x.M : \cap_n B_i{\to}A$ (the type is also not an intersection), 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 $\Gamma \vdash_{\cap} N : B_i$. In other words, we have the derivation:

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{}{\Gamma,x{:}\cap_n B_i \vdash x:B_1}\,(Ax) \quad \ldots \quad \dfrac{}{\Gamma,x{:}\cap_n B_i \vdash x:B_n}\,(Ax) \\ \boxed{\qquad\qquad\mathcal{D}_1\qquad\qquad} \\ \Gamma,x{:}\cap_n B_i \vdash M:A\end{array}}{\Gamma \vdash \lambda x.M:(\cap_n B_i)\to A}\,(\to I) \qquad \dfrac{\begin{array}{c}\boxed{\mathcal{D}_2^1} \\ \Gamma\vdash N:B_1\end{array} \quad\cdots\quad \begin{array}{c}\boxed{\mathcal{D}_2^n} \\ \Gamma\vdash N:B_n\end{array}}{\Gamma\vdash N:\cap_n B_i}\,(\cap I)}{\Gamma \vdash (\lambda x.M)\,N:A}\,(\to E)$$

Contracting a derivation for the redex $M[N/x]$ then gives a derivation for $\Gamma \vdash_\cap M[N/x] : A$ that can be obtained by replacing for $1 \leq i \leq n$ the sub-derivations

$$\dfrac{}{\Gamma,x{:}\cap_n B_i \vdash_\cap x : B_j}\,(Ax)$$

by the derivation for

$$\begin{array}{c}\boxed{\mathcal{D}_2^j} \\ \Gamma \vdash N : B_j\end{array}$$

yielding

$$\begin{array}{c}\begin{array}{c}\boxed{\mathcal{D}_2^1} \\ \Gamma\vdash N:B_1\end{array} \quad\cdots\quad \begin{array}{c}\boxed{\mathcal{D}_2^n} \\ \Gamma\vdash N:B_n\end{array} \\ \boxed{\qquad\qquad\mathcal{D}_1[N/x]\qquad\qquad} \\ \Gamma \vdash M[N/x] : A\end{array}$$

The real 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 $\Gamma \vdash_\cap (\lambda x.M)\,N : A$.

We distinguish two cases. If the term-variable x occurs in M, then the term N is an actual subterm of $M[N/x]$; assume N occurs typed in the derivation for $\Gamma \vdash_\cap M[N/x] : A$[5] and is typed with the different types $B_1,\ldots,B_n$, so, for $1 \leq i \leq n$, $\Gamma \vdash_\cap N : B_i$.

$$\begin{array}{c}\begin{array}{c}\boxed{\mathcal{D}_2^1} \\ \Gamma\vdash N:B_1\end{array} \quad\cdots\quad \begin{array}{c}\boxed{\mathcal{D}_2^n} \\ \Gamma\vdash N:B_n\end{array} \\ \boxed{\qquad\qquad\mathcal{D}_1\qquad\qquad} \\ \Gamma \vdash M[N/x] : A\end{array}$$

Then in Curry's system M cannot 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 $\Gamma,x{:}\cap_n B_i \vdash_\cap M : A$, and from this we get, by rule $(\to I)$, $\Gamma \vdash_\cap \lambda x.M : \cap_n B_i \to A$. Since, for every $1 \leq i \leq n$, $\Gamma \vdash_\cap N : B_i$, by rule $(\cap I)$ we also have $\Gamma \vdash_\cap N : \cap_n B_i$. Then, using $(\to E)$, the redex can be typed.

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{}{\Gamma,x{:}\cap_n B_i \vdash x:B_1}\,(Ax) \quad \ldots \quad \dfrac{}{\Gamma,x{:}\cap_n B_i \vdash x:B_n}\,(Ax) \\ \boxed{\qquad\qquad\mathcal{D}_1\qquad\qquad} \\ \Gamma,x{:}\cap_n B_i \vdash M:A\end{array}}{\Gamma \vdash \lambda x.M:(\cap_n B_i)\to A}\,(\to I) \qquad \dfrac{\begin{array}{c}\boxed{\mathcal{D}_2^1} \\ \Gamma\vdash N:B_1\end{array} \quad\cdots\quad \begin{array}{c}\boxed{\mathcal{D}_2^n} \\ \Gamma\vdash N:B_n\end{array}}{\Gamma\vdash N:\cap_n B_i}\,(\cap I)}{\Gamma \vdash (\lambda x.M)\,N:A}\,(\to E)$$

For the second case, if x does not occur in M, then the term N is not a subterm of $M[N/x]$, so N is *not* typed in the derivation for $\Gamma \vdash_\cap M[N/x] : A$, then in fact we have

$$\begin{array}{c}\boxed{\mathcal{D}_1} \\ \Gamma \vdash M : A\end{array}$$

[5] This need not be the case, as it can be that it occurs inside a subterm of M that is typed with $\top$.

By weakening, the term M can then be typed by $\Gamma,x{:}\top \vdash_\cap M : A$, and from this we get, by rule $(\rightarrow I)$, $\Gamma \vdash_\cap \lambda x.M : \top\rightarrow A$. Since also $\Gamma \vdash_\cap N : \top$ by rule $(\cap I)$, using $(\rightarrow E)$, the redex can be typed.

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{\mathcal{D}_1}}{\Gamma \vdash M : A}}{\Gamma,x{:}\top \vdash M : A}\,(Weak)}{\Gamma \vdash \lambda x.M : \top\rightarrow A}\,(\rightarrow I) \qquad \dfrac{}{\Gamma \vdash N : \top}\,(\cap I)}{\Gamma \vdash (\lambda x.M)\,N : A}\,(\rightarrow E)$$

Notice that it might well be that N is typeable in its own right, with a strict type. This is of no consequence: the construction does not need N to be typeable, since it will be discarded during reduction.

Before we come to a formal proof of this result, first we need some auxiliary results that are needed in the proof. The next lemma states that type assignment is closed for '$\leq$'.

* *Lemma 8.7* If $\Gamma \vdash_\cap M : \sigma$ and $\sigma \leq \tau$, and $\Gamma' \leq \Gamma$, then $\Gamma' \vdash_\cap M : \tau$.

Also, a term-substitution lemma is needed. Notice that, unlike for Curry's system, the implication holds in both directions.

Lemma 8.8 $\exists \sigma\, (\Gamma,x{:}\sigma \vdash_\cap M : \tau \wedge \Gamma \vdash_\cap N : \sigma) \iff \Gamma \vdash_\cap M[N/x] : \tau$.

Proof: By induction on M. Only the case $\tau = A$ is considered.

$(M \equiv x)$: $(\Rightarrow)$: $\exists \sigma\, (\Gamma,x{:}\sigma \vdash_\cap x : A \wedge \Gamma \vdash_\cap N : \sigma) \quad \Rightarrow \quad (Ax)$
$\exists A_1,\ldots,A_n, 1 \leq j \leq n\, (\sigma = \cap_n A_i \wedge A = A_j \wedge \Gamma \vdash_\cap N : \cap_n A_i) \quad \Rightarrow \quad (8.7)$
$\Gamma \vdash_\cap x[N/x] : A_j$.
$(\Leftarrow)$: $\Gamma \vdash_\cap x[N/x] : A \Rightarrow \Gamma,x{:}A \vdash_\cap x : A \wedge \Gamma \vdash_\cap N : A$.
$(M \equiv y \neq x)$: $(\Rightarrow)$: $\exists \sigma\, (\Gamma,x{:}\sigma \vdash_\cap y : A \wedge \Gamma \vdash_\cap N : \sigma) \Rightarrow (8.5)\ \Gamma \vdash_\cap y[N/x] : A$.
$(\Leftarrow)$: $\Gamma \vdash_\cap y[N/x] : A \Rightarrow \Gamma \vdash_\cap y : A \wedge \Gamma \vdash_\cap N : \top$.
$(M \equiv \lambda y.M')$: $(\iff)$: $\exists \sigma\, (\Gamma,x{:}\sigma \vdash_\cap \lambda y.M' : A \wedge \Gamma \vdash_\cap N : \sigma) \iff (\rightarrow I)$
$\exists \sigma,\tau,B\, (\Gamma,x{:}\sigma,y{:}\tau \vdash_\cap M' : B \wedge A = \tau\rightarrow B \wedge \Gamma \vdash_\cap N : \sigma) \iff (IH)$
$\exists \tau,B\, (\Gamma,y{:}\tau \vdash_\cap M'[N/x] : B \wedge A = \tau\rightarrow B) \iff (\rightarrow I)$
$\Gamma \vdash_\cap \lambda y.M'[N/x] : A$.

$(M \equiv M_1M_2)$: $(\iff)$: $\Gamma \vdash_\cap M_1M_2\,[N/x] : A \iff (\rightarrow E)$
$\exists \rho\, (\Gamma \vdash_\cap M_1[N/x] : \rho\rightarrow A \wedge \Gamma \vdash_\cap M_2[N/x] : \rho) \iff (IH)$
$\exists \sigma_1,\sigma_2,\rho\, (\Gamma,x{:}\sigma_i \vdash_\cap M_1 : \rho\rightarrow A \wedge \Gamma \vdash_\cap N : \sigma_1 \wedge \Gamma,x{:}\sigma_2 \vdash_\cap M_2 : \rho \wedge \Gamma \vdash_\cap N : \sigma_2)$
$\iff (\sigma = \sigma_1 \cap \sigma_2) \wedge (\cap I) \wedge (8.7) \wedge (\rightarrow E)$
$\exists \sigma\, (\Gamma,x{:}\sigma \vdash_\cap M_1M_2 : A \wedge \Gamma \vdash_\cap N : \sigma)$. $\square$

Notice that, although we only present the case for strict types, we do need the property for all types in the last part.

Theorem 8.9 *If $M =_\beta N$, then $\Gamma \vdash_\cap M : \tau$ if and only if $\Gamma \vdash_\cap N : \tau$, so the following rule is admissible in '$\vdash_\cap$':*

$$(=_\beta) : \ \frac{\Gamma \vdash_\cap M : \tau}{\Gamma \vdash_\cap N : \tau}\,(M =_\beta N)$$

Proof: By induction on the definition of '$=_\beta$'. The only part that needs attention is that of a redex, and $\tau = A$, so $\Gamma \vdash_\cap (\lambda x.M)\,N : A \iff \Gamma \vdash_\cap M[N/x] : A$; 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 σ such that $\Gamma,x{:}\sigma \vdash_\cap M : A$ and $\Gamma \vdash_\cap N : \sigma$. The result then follows from

Lemma 8.8. $\square$

* *Example 8.10* Remember from Example 1.11 that we have:

$$\begin{array}{rl}
\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx)) & \to_\beta \ \lambda f.f((\lambda x.f(xx))(\lambda x.f(xx))) \\
& \to_\beta \ \lambda f.f(f((\lambda x.f(xx))(\lambda x.f(xx)))) \\
& \to_\beta \ \lambda f.f(f(f((\lambda x.f(xx))(\lambda x.f(xx))))) \\
& \ \ \vdots \\
& \to_\beta \ \lambda f.f(f(f(f(f(f(f(\ldots)))))))
\end{array}$$

We can type these reducts as follows:

First, for $\lambda f.f(ZZ)$ (where $Z = \lambda x.f(xx)$):

$$\dfrac{\dfrac{\dfrac{}{f{:}\top\to A \vdash f : \top\to A} \qquad \dfrac{}{f{:}\top\to A \vdash ZZ : \top}\,(\cap I)}{f{:}\top\to A \vdash f(ZZ) : A}}{\emptyset \vdash \lambda f.f(ZZ) : (\top\to A)\to A}$$

The two sub-terms Z in this derivation both occur in a term typed with $\top$, so are not typed themselves. Using the construction of the proof of subject expansion (Lemma 8.8), we need to create a derivation for Z from the types it has inside the starting derivation; there are none, so we have to use $\top$ to type it.

We first reverse the substitution $xx[Z/x]$, and abstract to x to create:

$$\dfrac{\dfrac{\dfrac{}{f{:}\top\to A, x{:}\top \vdash f : \top\to A} \qquad \dfrac{}{f{:}\top\to A, x{:}\top \vdash xx : \top}\,(\cap I)}{f{:}\top\to A, x{:}\top \vdash f(xx) : A}}{f{:}\top\to A \vdash \lambda x.f(xx) : \top\to A}$$

and following the construction discussed above we can construct for the redex $\lambda f.ZZ$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f{:}\top\to A, x{:}\top \vdash f : \top\to A} \qquad \dfrac{}{f{:}\top\to A, x{:}\top \vdash xx : \top}\,(\cap I)}{f{:}\top\to A, x{:}\top \vdash f(xx) : A}}{f{:}\top\to A \vdash \lambda x.f(xx) : \top\to A}\,(\to I) \qquad \dfrac{}{f{:}\top\to A \vdash \lambda x.f(xx) : \top}\,(\cap I)}{f{:}\top\to A \vdash (\lambda x.f(xx))\,Z : A}\,(\to E)}{\emptyset \vdash \lambda f.(\lambda x.f(xx))\,Z : (\top\to A)\to A}$$

Second, for $\lambda f.f(f(ZZ))$: take $\sigma = (\top\to A)\cap(A\to B)$, then we can construct:

$$\dfrac{\dfrac{\dfrac{}{f{:}\sigma \vdash f : A\to B} \qquad \dfrac{\dfrac{}{f{:}\sigma \vdash f : \top\to A} \qquad \dfrac{}{f{:}\sigma \vdash ZZ : \top}\,(\cap I)}{f{:}\sigma \vdash f(ZZ) : A}}{f{:}\sigma \vdash f(f(ZZ)) : B}}{\emptyset \vdash \lambda f.f(f(ZZ)) : (\top\to A)\cap(A\to B)\to B}$$

and following the construction discussed above we can construct a derivation for $\lambda f.f(ZZ)$.

$$\dfrac{\dfrac{\dfrac{}{f{:}\sigma \vdash f : A\to B} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{f{:}\sigma, x{:}\top \vdash f : \top\to A} \qquad \dfrac{}{f{:}\sigma, x{:}\top \vdash xx : \top}\,(\cap I)}{f{:}\sigma, x{:}\top \vdash f(xx) : A}}{f{:}\sigma \vdash \lambda x.f(xx) : \top\to A}\,(\to I) \qquad \dfrac{}{f{:}\sigma \vdash \lambda x.f(xx) : \top}\,(\cap I)}{f{:}\sigma \vdash (\lambda x.f(xx))\,Z : A}\,(\to E)}{f{:}\sigma \vdash f(ZZ) : B}}{\emptyset \vdash \lambda f.f(ZZ) : (\top\to A)\cap(A\to B)\to B}$$

Notice that now Z has a strict type $\top\to A$, so the sub-derivation for $f{:}\sigma \vdash_\cap \lambda x.f(xx) : \top\to A$

gets used to build the derivation for the redex; the second occurrence is typed with $\top$. We would need to build the intersection of $\top\to A$ and $\top$, but that corresponds to just using $\top\to A$.

So we can construct a derivation for $\lambda f.ZZ$:

$$
\dfrac{
\dfrac{
\dfrac{\overline{f{:}\sigma,x{:}\top\to A \vdash f : A\to B} \qquad \begin{array}{c}\dfrac{\overline{f{:}\sigma,x{:}\top\to A \vdash x : \top\to A} \quad \overline{f{:}\sigma,x{:}\top\to A \vdash x : \top}}{f{:}\sigma,x{:}\top\to A \vdash xx : A}\,(\cap I)\\ \vdots\end{array}}{\dfrac{f{:}\sigma,x{:}\top\to A \vdash f(xx) : B}{f{:}\sigma \vdash \lambda x.f(xx) : (\top\to A)\to B}\,(\to I)}
\qquad
\dfrac{\overline{f{:}\sigma,x{:}\top \vdash f : \top\to A} \quad \overline{f{:}\sigma,x{:}\top \vdash xx : \top}\,(\cap I)}{\dfrac{f{:}\sigma,x{:}\top \vdash f(xx) : A}{f{:}\sigma \vdash \lambda x.f(xx) : \top\to A}}
}{f{:}\sigma \vdash ZZ : B}\,(\to E)
}{\emptyset \vdash \lambda f.ZZ : (\top\to A)\cap(A\to B)\to B}
$$

etc.

Interpreting a term M by its set of assignable types $\mathcal{T}(M) = \{\sigma \in \mathcal{T}_s \mid \exists \Gamma\,(\Gamma \vdash_\cap M : \sigma)\}$ gives a *semantics* for M, and a *filter model* for the Lambda Calculus (for details, see [11, 3, 5]).

Example 8.11 Types are not invariant by η-reduction. For example, notice that $\lambda xy.xy \to_\eta \lambda x.x$; we can derive $\vdash_\cap \lambda xy.xy : (\varphi_1\to\varphi_2)\to\varphi_1\cap\varphi_3\to\varphi_2$, but not $\vdash_\cap \lambda x.x : (\varphi_1\to\varphi_2)\to\varphi_1\cap\varphi_3\to\varphi_2$.

$$
\dfrac{\dfrac{\dfrac{\overline{x{:}\varphi_1\to\varphi_2,y{:}\varphi_1\cap\varphi_3 \vdash_\cap x : \varphi_1\to\varphi_2}\,(Ax) \qquad \overline{x{:}\varphi_1\to\varphi_2,y{:}\varphi_1\cap\varphi_3 \vdash_\cap y : \varphi_1}\,(Ax)}{x{:}\varphi_1\to\varphi_2,y{:}\varphi_1\cap\varphi_3 \vdash_\cap xy : \varphi_2}\,(\to E)}{x{:}\varphi_1\to\varphi_2 \vdash_\cap \lambda y.xy : \varphi_1\cap\varphi_3\to\varphi_2}\,(\to I)}{\vdash_\cap \lambda xy.xy : (\varphi_1\to\varphi_2)\to\varphi_1\cap\varphi_3\to\varphi_2}\,(\to I)
$$

We cannot derive $\vdash_\cap \lambda x.x : (\varphi_1\to\varphi_2)\to\varphi_1\cap\varphi_3\to\varphi_2$, since we cannot transform the type $\varphi_1\to\varphi_2$ into $\varphi_1\cap\varphi_3\to\varphi_2$ using $\leq$. There exists intersection systems that do allow this (see, for example, [3]).

8.4 Rank 2 and ML

It is possible to limit the structure of intersection types, and allow the intersection type constructor only up to a certain *rank* (or depth); for example, *1) in rank 0, no intersection is used; 2) in rank 1, intersection is only allowed on the top; 3) in rank 2, intersection is only allowed on the top, or on the left of the top arrow; etc.* All these variants give decidable restrictions. Moreover, rank 2 is already enough to model ML's let.

Example 8.12 The let is used for the case that we would like to type the redex $(\lambda x.E_2)\,E_1$ whenever the contractum is typeable using Curry types, but cannot:

$$
\begin{array}{c}
\Gamma \vdash E_1 : A[B/\varphi] \qquad \Gamma \vdash E_1 : A[C/\varphi] \\
\hline
\Gamma \vdash E_2[E_1/x] : D
\end{array}
$$

Using rank 2 types, the let-construct is not needed, since we can type the redex $(\lambda x.E_2)\,E_1$ directly (let $\Gamma' = \Gamma,x{:}A[B/\varphi]\cap A[C/\varphi]$):

$$
\dfrac{
\dfrac{\begin{array}{c}\overline{\Gamma' \vdash x : A[B/\varphi]}\,(Ax) \quad \overline{\Gamma' \vdash x : A[C/\varphi]}\,(Ax)\\ \hline \Gamma' \vdash E_2 : D\end{array}}{\Gamma \vdash \lambda x.E_2 : A[B/\varphi]\cap A[C/\varphi]\to D}\,(\to I)
\qquad
\dfrac{\Gamma \vdash E_1 : A[B/\varphi] \quad \Gamma \vdash E_1 : A[C/\varphi]}{\Gamma \vdash E_1 : A[B/\varphi]\cap A[C/\varphi]}\,(\cap I)
}{\Gamma \vdash (\lambda x.E_2)\,E_1 : D}\,(let)
$$

8.5 Approximation results

In this section, we will prove the approximation theorem for '$\vdash_\cap$'; from this result, we can show the characterisation of head-normalisation and normalisation of λ-terms using intersection types, i.e., all terms having a head-normal form are typeable in '$\vdash_\cap$' (with a type not equivalent to $\top$), and all terms having a normal form are typeable with a context and type that do not contain $\top$ at all. Because all these properties can be reduced to the *halting problem*, type assignment with intersection types is undecidable.

This result will be proven here using the reducibility technique [52]; we just state and show the main properties.

We will first focus on the approximation theorem

$$\Gamma \vdash_\cap M : \sigma \iff \exists\, \mathsf{A} \in \mathcal{A}(M)\, (\Gamma \vdash_\cap \mathsf{A} : \sigma)$$

So, for every type we can assign to a term M, we can find an approximant for M, so a redex-free term (that might contain $\bot$) that fits one of the reducts of M, and has that same type. In other words, the assigned type predicts (part of) the shape of the (infinite) normal form of M. For reasons of readability, we will abbreviate $\exists\, \mathsf{A} \in \mathcal{A}(M)\, (\Gamma \vdash_\cap \mathsf{A} : \sigma)$ by $\mathit{Appr}(\Gamma, M, \sigma)$.

The proof itself is rather convoluted and unexpected. We will define a notion of *term computable in a type*, inductively over the structure of types (to be exact, it is defined over triples of contexts, terms and types, or derivable statements), and show that if a term M is computable in a type σ, then there exists an approximant of M that has type σ.

This then means that we need to show that if $\Gamma \vdash_\cap M : \sigma$, then M is computable in σ. The main result for this is in Theorem 8.20, where we show that any computable extension M, where we replace variables by computable terms, yields a computable term of type σ. Since we also show that all variables are computable in any type, these two then imply that M is computable in σ.

A number of auxiliary results is needed to prove these steps. First we show that type assignment is upward closed for '$\sqsubseteq$' (see Definition 1.16); this is a direct consequence of the fact that $\bot$ can only appear in terms that are typed with $\top$.

Lemma 8.13 $\Gamma \vdash_\cap M : \sigma \wedge M \sqsubseteq M' \Rightarrow \Gamma \vdash_\cap M' : \sigma$.

Proof: By easy induction on the definition of '$\sqsubseteq$'; the base case, $\bot \sqsubseteq M'$, follows from the fact that then $\sigma = \top$. $\square$

The following basic properties are needed further on, and are direct consequences of results shown in Section 1.3.

* *Lemma 8.14* *i*) $\mathit{Appr}(\Gamma, xM_1\cdots M_n, \sigma{\to}A) \wedge \mathit{Appr}(\Gamma, N, \sigma) \Rightarrow \mathit{Appr}(\Gamma, xM_1\cdots M_nN, A)$.

ii) $\mathit{Appr}(\Gamma \cup \{z{:}\sigma\}, Mz, A) \wedge z \notin \mathit{fv}(M) \Rightarrow \mathit{Appr}(\Gamma, M, \sigma{\to}A)$.

iii) $\mathit{Appr}(\Gamma, M[N/x]\vec{P}, \sigma) \Rightarrow \mathit{Appr}(\Gamma, (\lambda x.M)N\vec{P}, \sigma)$.

Proof: *i*) $\mathsf{A} \in \mathcal{A}(xM_1\cdots M_n) \wedge \Gamma \vdash_\cap \mathsf{A} : \sigma{\to}A \wedge \mathsf{A}' \in \mathcal{A}(N) \wedge \Gamma \vdash_\cap \mathsf{A}' : \sigma$

$\Rightarrow$ (1.18 (*i*) $\wedge$ $({\to}E)$ $\wedge$ $\mathsf{A} \neq \bot$) $\quad \mathsf{A}\mathsf{A}' \in \mathcal{A}(xM_1\cdots M_nN) \wedge \Gamma \vdash_\cap \mathsf{A}\mathsf{A}' : A$.

ii) $\mathsf{A} \in \mathcal{A}(Mz) \wedge \Gamma, z{:}\sigma \vdash_\cap \mathsf{A} : A \wedge z \notin \mathit{fv}(M) \Rightarrow$ (1.18 (*ii*))

a) $\mathsf{A} \equiv \mathsf{A}'z \wedge z \notin \mathit{fv}(\mathsf{A}') \wedge \mathsf{A}' \in \mathcal{A}(M) \wedge \Gamma, z{:}\sigma \vdash_\cap \mathsf{A}'z : A \Rightarrow$
$\mathsf{A}' \in \mathcal{A}(M) \wedge \Gamma \vdash_\cap \mathsf{A}' : \sigma{\to}A$.

b) $\lambda z.\mathsf{A} \in \mathcal{A}(M) \wedge \Gamma, z{:}\sigma \vdash_\cap \mathsf{A} : A \Rightarrow \lambda z.\mathsf{A} \in \mathcal{A}(M) \wedge \Gamma \vdash_\cap \lambda z.\mathsf{A} : \sigma{\to}A$.

iii) Since $M[N/x]\vec{P} =_\beta (\lambda x.M)N\vec{P}$, the result follows by Lemma 1.21. $\square$

In order to prove that, for each term typeable in '$\vdash_\cap$', an approximant can be found that can be assigned the same type, a notion of computability is introduced.

* *Definition 8.15* (COMPUTABILITY PREDICATE) The predicate $Comp(\Gamma, M, \rho)$ is inductively defined by:

$$\begin{array}{rcl} Comp(\Gamma, M, \varphi) & \iff & Appr(\Gamma, M, \varphi) \\ Comp(\Gamma, M, \sigma \rightarrow A) & \iff & (\, Comp(\Gamma', N, \sigma) \Rightarrow Comp(\cap\{\Gamma, \Gamma'\}, MN, A)\,) \\ Comp(\Gamma, M, \cap_n A_i) & \iff & \forall 1 \leq i \leq n\, (Comp(\Gamma, M, A_i)) \end{array}$$

Notice that $Comp(\Gamma, M, \top)$ holds as special case of the third part.

We will now show that the computability predicate is closed for '$\leq$':

* *Lemma 8.16* *i*) If $Comp(\Gamma, M, \sigma)$, and $\Gamma' \leq \Gamma$, then $Comp(\Gamma', M, \sigma)$.
ii) If $Comp(\Gamma, M, \sigma)$, and $\sigma \leq \tau$, then $Comp(\Gamma, M, \tau)$.

Proof: By straightforward induction on the definition of '$\leq$'. □

We will now show that the computability predicate is closed for β-expansion

* *Lemma 8.17* $Comp(\Gamma, M[N/x]\vec{P}, \sigma) \Rightarrow Comp(\Gamma, (\lambda x.M)N\vec{P}, \sigma)$.

Proof: By induction on the definition of *Comp*.

$(\sigma = \varphi)$: $Comp(\Gamma, M[N/x]\vec{P}, \varphi) \Rightarrow Appr(\Gamma, M[N/x]\vec{P}, \varphi) \Rightarrow$ (8.14(*iii*))
$Appr(\Gamma, (\lambda x.M)N\vec{P}, \varphi) \Rightarrow Comp(\Gamma, (\lambda x.M)N\vec{P}, \varphi)$.

$(\sigma = \tau \rightarrow A)$:

$$\begin{array}{ll} Comp(\Gamma, M[N/x]\vec{P}, \tau \rightarrow A) & \Rightarrow (8.15) \\ (Comp(\Gamma', Q, \tau) \Rightarrow Comp(\cap\{\Gamma, \Gamma'\}, M[N/x]\vec{P}Q, A)) & \Rightarrow (IH) \\ (Comp(\Gamma', Q, \tau) \Rightarrow Comp(\cap\{\Gamma, \Gamma'\}, (\lambda x.M)N\vec{P}Q, A)) & \Rightarrow (8.15) \\ Comp(\Gamma, (\lambda x.M)N\vec{P}, \tau \rightarrow A). & \end{array}$$

$(\sigma = \cap_n A_i)$: By induction. □

The following theorem essentially shows that all term-variables are computable of any type, and that all terms computable of a certain type have an approximant with that same type. These results are interdependent, in that they can only be shown together, using simultaneously induction.

* **Theorem 8.18** *i*) $Appr(\Gamma, xM_1 \cdots M_n, \rho) \Rightarrow Comp(\Gamma, xM_1 \cdots M_n, \rho)$.
ii) $Comp(\Gamma, M, \rho) \Rightarrow Appr(\Gamma, M, \rho)$.

Proof: Simultaneously by induction on the structure of types. The only interesting case is when $\rho = \sigma \rightarrow A$; when ρ is a type-variable, the result is immediate and when it is an intersection type, it is dealt with by induction.

i)

$$\begin{array}{ll} Appr(\Gamma, xM_1 \cdots M_n, \sigma \rightarrow A) & \Rightarrow (IH(ii)) \\ (Comp(\Gamma', N, \sigma) \Rightarrow Appr(\Gamma, xM_1 \cdots M_n, \sigma \rightarrow A) \wedge Appr(\Gamma', N, \sigma)) & \Rightarrow (8.14(i)) \\ (Comp(\Gamma', N, \sigma) \Rightarrow Appr(\cap\{\Gamma, \Gamma'\}, xM_1 \cdots M_n N, A)) & \Rightarrow (IH(i)) \\ (Comp(\Gamma', N, \sigma) \Rightarrow Comp(\cap\{\Gamma, \Gamma'\}, xM_1 \cdots M_n N, A)) & \Rightarrow (8.15) \\ Comp(\Gamma, xM_1 \cdots M_n, \sigma \rightarrow A). & \end{array}$$

ii)

$$\begin{array}{ll} Comp(\Gamma, M, \sigma \rightarrow A) \wedge z \notin fv(M) & \Rightarrow (IH(i)) \\ Comp(\Gamma, M, \sigma \rightarrow A) \wedge Comp(\{z{:}\sigma\}, z, \sigma) \wedge z \notin fv(M) & \Rightarrow (8.15) \\ Comp(\cap\{\Gamma, \{z{:}\sigma\}\}, Mz, A) \wedge z \notin fv(M) & \Rightarrow (IH(ii)) \\ Appr(\cap\{\Gamma, \{z{:}\sigma\}\}, Mz, A) \wedge z \notin fv(M) & \Rightarrow (8.14(ii)) \\ Appr(\Gamma, M, \sigma \rightarrow A). & \end{array}$$

□

Notice that, as a corollary of the first of these two results, we get that term-variables are computable for any type:

* *Corollary 8.19* $Comp(\{x{:}\sigma\}, x, \sigma)$, for all x, σ.

The main result of this section states that a computable extension of a typeable term yields a computable term.

* **Theorem 8.20** (Replacement Theorem) If $x_1{:}\mu_1,\ldots,x_n{:}\mu_n \vdash_\cap M : \sigma$, and, for every $1 \leq i \leq n$, $Comp(\Gamma_i, N_i, \mu_i)$, then $Comp(\cap_n\Gamma_i, M[\overrightarrow{N_i/x_i}], \sigma)$.

Proof: By induction on the structure of derivations; let $x_1{:}\mu_1,\ldots,x_n{:}\mu_n = \Gamma_0$, and $\Gamma' = \cap_n\Gamma_i$.

(Ax): Then $M \equiv x_j$, for some $1 \leq j \leq n$, $\mu_j \leq \sigma$, and $M[\overrightarrow{N_i/x_i}] \equiv x_j[\overrightarrow{N_i/x_i}] \equiv N_j$.

$$\begin{array}{ll} Comp(\Gamma_j, N_j, \mu_j) & \Rightarrow (\mu_j \leq \sigma \wedge 8.16) \\ Comp(\Gamma_j, N_j, \sigma) & \Rightarrow (\Gamma' \leq \Gamma_j \wedge 8.16) \\ Comp(\Gamma', N_j, \sigma). & \end{array}$$

$(\to I)$: Then $M \equiv \lambda y.M'$, $\sigma = \rho\to A$, and $\Gamma_0,y{:}\rho \vdash_\cap M' : A$.

$$\begin{array}{ll} \forall 1 \leq i \leq n\,(Comp(\Gamma_i, N_i, \mu_i)) \wedge \Gamma_0,y{:}\rho \vdash_\cap M' : A & \Rightarrow (IH \wedge 8.16\,(i)) \\ (Comp(\Gamma', N, \rho) \Rightarrow Comp(\cap\{\Gamma',\Gamma''\}, M'[\overrightarrow{N_i/x_i}, N/y], A)) & \Rightarrow (8.17) \\ (Comp(\Gamma', N, \rho) \Rightarrow Comp(\cap\{\Gamma',\Gamma''\}, (\lambda y.M'[\overrightarrow{N_i/x_i}])\,N, A)) & \Rightarrow (8.15) \\ Comp(\Gamma', (\lambda y.M')[\overrightarrow{N_i/x_i}], \rho\to A). & \end{array}$$

$(\to E)$: Then $M \equiv M_1M_2$, $\Gamma_0 \vdash_\cap M_1 : \rho\to\sigma$, and $\Gamma_0 \vdash_\cap M_2 : \rho$.

$$\begin{array}{ll} \forall 1 \leq i \leq n\,(Comp(\Gamma_i, N_i, \mu_i)) \wedge \Gamma_0 \vdash_\cap M_1 : \rho\to\sigma \wedge \Gamma_0 \vdash_\cap M_2 : \rho & \Rightarrow (IH) \\ Comp(\Gamma', M_1[\overrightarrow{N_i/x_i}], \rho\to\sigma) \wedge Comp(\Gamma', M_2[\overrightarrow{N_i/x_i}], \rho) & \Rightarrow (8.15) \\ Comp(\Gamma', (M_1M_2)[\overrightarrow{N_i/x_i}], \sigma). & \end{array}$$

$(\cap I)$: Straightforward by induction. □

Now we can show:

* **Theorem 8.21** $\Gamma \vdash_\cap M : \sigma$ then $Comp(\Gamma, M, \sigma)$

Proof: By Corollary 8.19, for every $x_i{:}\tau_i \in \Gamma$ we know that $Comp(\Gamma, x_i, \tau_i)$, so in particular by Theorem 8.20 we have $Comp(\Gamma, M[\overrightarrow{x_i/x_i}], \sigma)$, and that $M[\overrightarrow{x_i/x_i}] = M$.

We can now show the approximation result.

Theorem 8.22 (Approximation theorem) $\Gamma \vdash_\cap M : \sigma \iff \exists\, \mathsf{A} \in \mathcal{A}(M)\,(\Gamma \vdash_\cap \mathsf{A} : \sigma)$.

Proof:$(\Rightarrow)$: $\Gamma \vdash_\cap M : \sigma \;\Rightarrow (8.21)\;\; Comp(\Gamma, M, \sigma) \;\Rightarrow (8.18\,(ii))\;\; \exists\, \mathsf{A} \in \mathcal{A}(M)\,(\Gamma \vdash_\cap \mathsf{A} : \sigma)$.

$(\Leftarrow)$: Let $\mathsf{A} \in \mathcal{A}(M)$ be such that $\Gamma \vdash_\cap \mathsf{A} : \sigma$. Since $\mathsf{A} \in \mathcal{A}(M)$, there is an M' such that $M' =_\beta M$ and $\mathsf{A} \sqsubseteq M'$. Then, by Lemma 8.13, $\Gamma \vdash_\cap M' : \sigma$ and, by Corollary 8.9, also $\Gamma \vdash_\cap M : \sigma$. □

8.6 Characterisation of (head/strong) normalisation

Using the approximation result, the following head-normalisation result becomes easy to show.

Theorem 8.23 (Head-normalisation) $\exists \Gamma, A\,(\Gamma \vdash_\cap M : A) \iff$ *M has a head-normal form.*

Proof:$(\Rightarrow)$: If $\Gamma \vdash_\cap M : A$, then, by Theorem 8.22, there exists $\mathsf{A} \in \mathcal{A}(M)$ such that $\Gamma \vdash_\cap \mathsf{A} : A$. By Definition 1.15, there exists $M' =_\beta M$ such that $\mathsf{A} \sqsubseteq M$. Since A is strict, $\mathsf{A} \not\equiv \bot$, so A is either $\lambda x.\mathsf{A}_1$ or $x\mathsf{A}_1\cdots\mathsf{A}_n$, with $n \geq 0$. Since $\mathsf{A} \sqsubseteq M'$, M' is either $\lambda x.M_1$, or $xM_1\cdots M_n$. Then M has a head-normal form.

$(\Leftarrow)$: If M has a head-normal form, then there exists $M' =_\beta M$ such that M' is either $\lambda x.M_1$ or $xM_1\cdots M_n$, with $M_i \in \Lambda$, for all $1 \leq i \leq n$. Then either:

a) $M' \equiv \lambda x.M_1$. Since M_1 is in head-normal form, by induction there are Γ, A such that $\Gamma \vdash_\cap M_1 : A$. If $x{:}\tau \in \Gamma$, then $\Gamma\backslash x \vdash_\cap \lambda x.M_1 : \tau\to A$; otherwise $\Gamma \vdash_\cap \lambda x.M_1 : \top\to A$.

b) $M' \equiv xM_1\cdots M_n, (n \geq 0)$. Then $x{:}\top\to\cdots\to\top\to A \vdash_\cap xM_1\cdots M_n : A$.

So there exists Γ, A such that $\Gamma \vdash_\cap M' : A$, and, by Corollary 8.9, we get $\Gamma \vdash_\cap M : A$. $\square$

To prepare the characterisation of normalisability by assignable types, first we prove that a term in $\lambda\bot$-normal form is typeable without $\top$, if and only if it does not contain $\bot$. This forms the basis for the result that all normalisable terms are typeable without $\top$.

* *Lemma 8.24* *i*) If $\Gamma \vdash_\cap \mathsf{A} : \sigma$ and Γ,σ are $\top$-free, then A is $\bot$-free.

ii) If A is $\bot$-free, then there are $\top$-free Γ and σ, such that $\Gamma \vdash_\cap \mathsf{A} : \sigma$.

Proof: By induction on the structure of terms in approximate normal form.

i) As before, only the part $\sigma = A \in \mathcal{T}_s$ is shown.

($\mathsf{A} \equiv \bot$): Impossible, since $\bot$ is only typeable by $\top$.

($\mathsf{A} \equiv \lambda x.\mathsf{A}'$): Then $A = \rho \rightarrow B$, and $\Gamma, x{:}\rho \vdash_\cap \mathsf{A} : B$. Since Γ, A are $\top$-free, so are $\Gamma, x{:}\rho$ and B, so, by induction, A' is $\bot$-free, so also $\lambda x.\mathsf{A}'$ is $\bot$-free.

($\mathsf{A} \equiv x\mathsf{A}_1 \cdots \mathsf{A}_n$): Then, by $(\rightarrow E)$ and (Ax), there are σ_i, τ_i $(1 \leq i \leq n), \psi$, such that $\Gamma \vdash_\cap \mathsf{A}_i : \sigma_i$ for all $1 \leq i \leq n$, $x{:}\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \psi \in \Gamma$, and $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \psi \leq \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \psi$. So, especially, for every $1 \leq i \leq n$, $\sigma_i \leq \tau_i$. By Lemma 8.7, also $\Gamma \vdash_\cap \mathsf{A}_i : \tau_i$, for every $1 \leq i \leq n$. Since each τ_i occurs in Γ, all are $\top$-free, so by induction each A_i is $\bot$-free. Then also $x\mathsf{A}_1 \cdots \mathsf{A}_n$ is $\bot$-free.

ii) ($\mathsf{A} \equiv \lambda x.\mathsf{A}'$): By induction there are Γ, B such that $\Gamma \vdash_\cap \mathsf{A}' : B$ and Γ, B are $\top$-free. If x does not occur in Γ, take an $\top$-free $\sigma \in \mathcal{T}$. Otherwise, there exist $x{:}\tau \in \Gamma$, and τ is $\top$-free. In any case, $\Gamma \backslash x \vdash_\cap \lambda x.\mathsf{A}' : \tau \rightarrow B$, and $\Gamma \backslash x$ and $\tau \rightarrow B$ are $\top$-free.

($\mathsf{A} \equiv x\mathsf{A}_1 \cdots \mathsf{A}_n$, *with* $(n \geq 0)$): By induction there are Γ_i, σ_i $(1 \leq i \leq n)$ such that, for every $1 \leq i \leq n$, $\Gamma_i \vdash_\cap \mathsf{A}_i : \sigma_i$, and Γ_i, σ_i are $\top$-free. Take any A strict, such that $\top$ does not occur in A, and $\Gamma = \cap_n \Gamma_i \cap \{x{:}\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow A\}$. Then $\Gamma \vdash_\cap x\mathsf{A}_1 \cdots \mathsf{A}_n : A$, and Γ and A are $\top$-free. $\square$

By construction of the proof, in part (*ii*), the type constant $\top$ is not used at all in the derivation.

Now it is possible to prove that we can characterise normalisation.

Theorem 8.25 (Normalisation) $\exists \Gamma, \sigma\, (\Gamma \vdash_\cap M : \sigma \wedge \Gamma, \sigma\ \top\text{-}free) \Longleftrightarrow M$ *has a normal form.*

Proof:($\Rightarrow$): If $\Gamma \vdash_\cap M : \sigma$, then, by Theorem 8.22, there exists $\mathsf{A} \in \mathcal{A}(M)$ such that $\Gamma \vdash_\cap \mathsf{A} : \sigma$. Then, by Lemma 8.24 (*i*), this A is $\bot$-free. By Definition 1.15, there exists $M' =_\beta M$ such that $\mathsf{A} \sqsubseteq M'$. Since A is $\bot$-free, we have $\mathsf{A} \equiv M'$, so M' itself is in normal form, so, especially, M has a normal form.

($\Leftarrow$): If M' is the normal form of M, then it is a $\bot$-free approximate normal form. Then, by Lemma 8.24 (*ii*), there are $\top$-free Γ, σ such that $\Gamma \vdash_\cap M' : \sigma$, and $\Gamma \vdash_\cap M : \sigma$ follows by Theorem 8.9. $\square$

Using the technique of computability, we can also show:

Theorem 8.26 *M is strongly normalisable, if and only if there are Γ and A such that $\Gamma \vdash_\cap M : A$, and in this derivation $\top$ is not used at all.*

We would be needing a separate proof, in that this does not follow from the results shown above.

We can now reason that the converse of Corollary 8.9 does not hold: terms that do not have a head-normal form are all only typeable with $\top$, but these cannot all be converted to each other.

8.7 Principal intersection pairs

It is possible to define a notion of principal pair for λ-terms using intersection types [51, 4, 5].

We can approach this problem in the traditional way, by defining an algorithm that constructs a pair of context and (strict) type for a term, but will walk a different path here. Through the approximation result we know that every type for a term correspond to that of one of its approximants, but is this true for the principal pair as well? Which approximant will serve for the principal pair?

If the collection of approximants for a term M, $\mathcal{A}(M)$, is finite, that set contains a largest element with respect to $\sqsubseteq$, A_M, in the sense that for every $\mathsf{A} \in \mathcal{A}(M)$ we can show that $\mathsf{A} \sqsubseteq \mathsf{A}_M$. Then by Lemma 8.13, we can show that if $\Gamma \vdash_\cap \mathsf{A} : B$, and $\mathsf{A} \sqsubseteq \mathsf{A}_M$, then $\Gamma \vdash_\cap \mathsf{A}_M : B$, so the maximal approximant has all the type assignments we have for M. If $\mathcal{A}(M)$ is infinite, then there is no maximal element, and we cannot choose one approximant as a representative that has all types, but need to consider the whole set. We will see this come back in Definition 8.30.

We first define principal pairs first for terms in approximate normal forms.

Definition 8.27 (Principal Pairs) *i*) For $\mathsf{A} \in \mathcal{A}$, we define $pp(\mathsf{A})$, the *principal pair* of A, by:

a) $pp(\bot) = \langle \emptyset, \top \rangle$.

b) $pp(x) = \langle \{x{:}\varphi\}, \varphi \rangle$.

c) If $\mathsf{A} \neq \bot$, and $pp(\mathsf{A}) = \langle \Pi, P \rangle$, then:

1) If x occurs free in A, and $x{:}\sigma \in \Pi$, then $pp(\lambda x.\mathsf{A}) = \langle \Pi \backslash x, \sigma \to P \rangle$.

2) Otherwise $pp(\lambda x.\mathsf{A}) = \langle \Pi, \top \to P \rangle$.

d) If for $1 \leq i \leq n$, $pp(\mathsf{A}_i) = \langle \Pi_i, P_i \rangle$ (disjoint in pairs), then

$$pp(x\mathsf{A}_1 \cdots \mathsf{A}_n) \;=\; \langle \cap_n \Pi_i \cap \{ x{:}P_1 \to \cdots \to P_n \to \varphi \}, \varphi \rangle,$$

where φ is a type-variable that does not occur in $pp(\mathsf{A}_i)$, for $1 \leq i \leq n$.

ii) $\mathcal{P} = \{ \langle \Pi, P \rangle \mid \exists \mathsf{A} \in \mathcal{A}\, (pp(\mathsf{A}) = \langle \Pi, P \rangle) \}$.

Since unification is not used in this definition, the following result is almost immediate:

Lemma 8.28 If $pp(\mathsf{A}) = \langle \Pi, P \rangle$, then $\Pi \vdash_\cap \mathsf{A} : P$.

Proof: Easy. □

Notice that this algorithm always returns a pair. This immediately implies:

Corollary 8.29 For every $\mathsf{A} \in \mathcal{A}$ with $\mathsf{A} \neq \bot$ there exist Γ, B such that $\Gamma \vdash_\cap \mathsf{A} : B$.

We can now define a notion of principal pairs for arbitrary λ-terms:

Definition 8.30 (Principal intersection pairs) *i*) Let M be a term. We define $\mathcal{P}(M)$ as the set of all principal pairs for all approximants of M: $\mathcal{P}(M) = \{ pp(\mathsf{A}) \mid \mathsf{A} \in \mathcal{A}(M) \}$.

ii) $\mathcal{P}(M)$ is an ideal[6] in $\mathcal{P}$, and therefore:

a) If $\mathcal{P}(M)$ is finite, then there exists a pair $\langle \Pi, P \rangle = \bigsqcup \mathcal{P}(M)$, where $\langle \Pi, P \rangle \in \mathcal{P}$. This pair is then called the principal pair of M.

b) If $\mathcal{P}(M)$ is infinite, $\bigsqcup \mathcal{P}(M)$ does not exist in $\mathcal{P}$. The principal pair of M is then the infinite set of pairs $\mathcal{P}(M)$.

---

[6] A subset I of a partially ordered set $(P, \leq)$ is an *ideal* if the following conditions hold:

a) I is *non-empty*,

b) I is *downwards closed*: for every $x \in I$, $y \in P$: if $y \leq x$ then $y \in I$ (I is a lower set), and

c) for all $x, y \in I$, there exists $z \in I$, such that $x \leq z$ and $y \leq z$ (I is a directed set)

In case $\mathcal{P}(M)$ is finite, $\bigsqcup\mathcal{P}(M)$ corresponds to the $pp(\mathsf{A}_l)$, where A_l is the largest approximant of M.

The proof that this notion is well defined, *i.e.* defines a notion of principal pairs, is quite involved and requires a notion of unification that deals with intersection types as well, and needs more operations on types than just substitution of type variables. We will illustrate this by having a look at how unification would have to operate.

Example 8.31 Assume we are trying to type $\Delta\Delta = (\lambda x.xx)\,(\lambda x.xx)$. We have that $pp(\lambda x.xx) = (\varphi_1{\to}\varphi_2)\cap\varphi_1 \to \varphi_2$, so in order to type $\Delta\Delta$ we need to call

$$unify \quad (\varphi_1{\to}\varphi_2)\cap\varphi_1 \to \varphi_2 \quad ((\varphi_3{\to}\varphi_4)\cap\varphi_3 \to \varphi_4) \to \varphi_5.$$

This means that we will unify the arrow type $(\varphi_3{\to}\varphi_4)\cap\varphi_3 \to \varphi_4$ with the intersection type $(\varphi_1{\to}\varphi_2)\cap\varphi_1$.

$$unify \quad (\varphi_1{\to}\varphi_2)\cap\varphi_1 \quad (\varphi_3{\to}\varphi_4)\cap\varphi_3 \to \varphi_4. \tag{1}$$

To be able to do that, we need to copy the arrow type into two fresh copies (this operation is called *expansion*) and unify each with a component of the intersection type. So we get

$$unify \quad (\varphi_1{\to}\varphi_2)\cap\varphi_1 \quad ((\varphi_3{\to}\varphi_4)\cap\varphi_3 \to \varphi_4)\cap((\varphi_6{\to}\varphi_7)\cap\varphi_6 \to \varphi_7)$$

which calls

$$\begin{array}{rcl} S_1 &=& unify \quad \varphi_1{\to}\varphi_2 \quad (\varphi_3{\to}\varphi_4)\cap\varphi_3 \to \varphi_4 \\ S_2 &=& unify \quad (S_1\,\varphi_1) \quad (S_1\,(\varphi_6{\to}\varphi_7)\cap\varphi_6 \to \varphi_7). \end{array}$$

The first returns $S_1 = (\varphi_1 \mapsto (\varphi_3{\to}\varphi_4)\cap\varphi_3,\ \varphi_2 \mapsto \varphi_4)$, so

$$S_2 \;=\; unify \quad (\varphi_3{\to}\varphi_4)\cap\varphi_3 \quad (\varphi_6{\to}\varphi_7)\cap\varphi_6 \to \varphi_7$$

Notice that this is exactly the kind of unification we started with in (1); the unification loops and never returns an answer. Of course this is correct: $\Delta\Delta$ had only $\bot$ as approximant, so can only be typed with $\top$, and unification should not be successful.

A semi-algorithm that calculates principal intersection pairs for λ-terms is defined in [50]. It tries to type all sub-terms, and thereby only returns a result for strongly normalisable terms - it loops on unification when dealing with terms that are not *SN*. Since stating that a term has a principal pair corresponds to being able to say it is typeable, the former is, of course, undecidable.

Exercises

Exercise 8.32 Show

i) $\emptyset \vdash_\cap \lambda xy.xy : (A{\to}B){\to}(C\cap A){\to}B$.

ii) $\emptyset \vdash_\cap \lambda xyz.xz(yz) : (A{\to}B{\to}C){\to}(D{\to}B){\to}(A\cap D){\to}C$.

iii) $\emptyset \vdash_\cap \lambda xyz.xz(yz) : (A{\to}\top{\to}C){\to}\top{\to}A{\to}C$

iv) $\emptyset \vdash_\cap (\lambda xyz.xz(yz))\,(\lambda ab.a) : \top{\to}A{\to}A$ *(use the previous result without repeating the whole structure).*

Exercise 8.33 Compare the principal types of the terms $(\lambda xyz.xz(yz))(\lambda xy.x)(\lambda x.x)$ *and* $\lambda x.x$ *in the Curry system and in the intersection system. Motivate your answer.*

Exercise 8.34 Does $\lambda x.xx$ *have a single principal intersection type? If yes, give a derivation that derives that type; if no, explain why. Show that* $(\top{\to}\varphi){\to}\varphi$ *is a type for* $\lambda x.xx$.

Exercise 8.35 Does $\lambda y.(\lambda x.xx)y$ *have a single principal intersection type? If yes, give a derivation that derives that type; if no, explain why. Show that* $(\top{\to}\varphi){\to}\varphi$ *is a type for* $\lambda y.(\lambda x.xx)y$.

Exercise 8.36 *Does $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ have a single principal intersection type? If yes, give a derivation that derives that type; if no, explain why.*

Show that $(\top \rightarrow \varphi) \rightarrow \varphi$ is a type for $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] 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.

[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 Colloquim 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] R. Bloo and K.H. Rose. Preservation of Strong Normalisation in Named Lambda Calculi with Explicit Substitution and Garbage Collection. In *CSN'95 – Computer Science in the Netherlands*, pages 62–72, 1995.

[13] 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.

[14] 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.

[15] A. Church. A Note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] M. Coppo, M. Dezani-Ciancaglini, and M. Zacchi. Type Theories, Normal Forms and D_∞-Lambda-Models. *Information and Computation*, 72(2):85–116, 1987.

[21] H.B. Curry. Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics*, 52:509–536, 789–834, 1930.

[22] H.B. Curry. Functionality in Combinatory Logic. In *Proc. Nat. Acad. Sci. U.S.A*, volume 20, pages 584–590, 1934.

[23] H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[24] 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.

[25] L.M.M. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, Department of Computer Science, Edinburgh, 1984. Thesis CST-33-85.

[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.Y. Levin, and B.C. Pierce. Recursive subtyping revealed: functional pearl. In M. Odersky and P. Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference On Functional Programming (ICFP '00), Montreal, Canada*, pages 221–231. ACM, September 18-21 2000.

[29] J.-Y. Girard. The System F of Variable Types, Fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[30] C.A. Gunter and D.S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 633–674. North-Holland, 1990.

[31] B Harper. Introduction to Standard ML. Technical report, ECS-LFCS-86-14, Laboratory for the Foundation of Computer Science, Edinburgh University, 1986.

[32] J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] J.W. Klop. Term Rewriting Systems: a tutorial. *EATCS Bulletin*, 32:143–182, 1987.

[38] 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.

[39] 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.

[40] J-L. Krivine. A call-by-name lambda-calculus machine. *Higher Order and Symbolic Computation*, 20(3):199–207, 2007.

[41] S. Lengrand, P. Lescanne, D. Dougherty, M. Dezani-Ciancaglini, and S. van Bakel. Intersection types for explicit substitutions. *Information and Computation*, 189(1):17–42, 2004.

[42] P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$: a Journey Through Calculi of Explicit Substitutions. In *POPL'94*, pages 60–69, 1994.

[43] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[44] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, 1990. Revised edition.

[45] 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.

[46] F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Aanalysis*. Springer Verlag, 1999.

[47] F. Pottier. A modern eye on ML type inference - Old techniques and recent Developments, September 2005.

[48] 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.

[49] J.A. Robinson. A Machine-Oriented Logic Based on Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[50] S. Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59:181–209, 1988.

[51] S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28:151–169, 1984.

[52] W. Tait. Intensional Interpretations of Functionals of Finite Type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[53] 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.

[54] C.P. Wadsworth. The Relation Between Computational and Denotational Properties for Scott's D_∞-Models of the Lambda-Calculus. *SIAM Journal on Computing*, 5(3):488–521, 1976.

[55] J.B. Wells. Typeability and type checking in Second order λ-calculus are equal and undecidable. In *Proceedings of the ninth Annual IEEE Symposium on Logic in Computer Science,* Paris, France, 1994.

[56] J.B. Wells. The essence of principal typings. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *Proceedings of ICALP'92. 29$^{\text{th}}$ International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer Verlag, 2002.

Appendix: Answers to exercises

Exercise 1.28 *i*) $(\lambda x.(\lambda y.(\lambda z.(((xz)\,y)\,z))))$

ii) $(\lambda x.(\lambda y.(\lambda z.((xz)\,(yz)))))$

iii) $((\lambda x.(\lambda y.x))\,(\lambda z.(wz)))$

iv) $\lambda x_1.(\lambda x_2.x_1x_2)x_1x_3$

v) $(\lambda x_1.(\lambda x_2.x_1x_2x_1))x_3$

vi) $(\lambda xy.x)\,\lambda z.za$

Exercise 1.29 *Proof:* By induction on the definition of terms.

$(M = x)$: Then $x\,[N/x]\,[L/y] = N[L/y] = x\,[L/y]\,[N\,[L/y]/x]$.

$(M = y)$: Then $y\,[N/x]\,[L/y] = y[L/y] =(x \notin fv\,(L))\; y\,[L/y]\,[N\,[L/y]/x]$.

$(M = z)$: Then $z\,[N/x]\,[L/y] = z[L/y] = z\,[L/y]\,[N\,[L/y]/x]$.

$(M = \lambda z.P)$: Then $(\lambda z.P)\,[N/x]\,[L/y] = \lambda z.(P\,[N/x]\,[L/y]) = (IH)\ \lambda z.(P\,[L/y]\,[N\,[L/y]/x]) = (\lambda z.P)\,[L/y]\,[N\,[L/y]/x])$.

$(M = PQ)$: Then $(PQ)\,[N/x]\,[L/y] = (P\,[N/x]\,[L/y])\ (Q\,[N/x]\,[L/y]) = (IH)$ $(P\,[L/y]\,[N\,[L/y]/x])\ (Q\,[L/y]\,[N\,[L/y]/x]) = (PQ)\,[L/y]\,[N\,[L/y]/x]$.

Exercise 1.30 We run head-reduction on all terms.

i) $(\lambda xyz.xz(yz))\,(\lambda ab.a) \;\to_\beta\; \lambda yz.(\lambda ab.a)z\,(yz) \;\to_\beta\; \lambda yz.(\lambda b.z)\,(yz) \;\to_\beta\; \lambda yz.z$

This is a head-normal form that is a normal form as well.

ii) $(\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx)))\,(\lambda a.a) \;\to_\beta$
$(\lambda x.(\lambda a.a)(xx))\,(\lambda x.(\lambda a.a)(xx)) \quad = \quad (with\ \lambda x.(\lambda a.a)(xx) = P)$
$PP \;\to_\beta\; (\lambda a.a)(PP) \;\to_\beta\; PP\ etc$

No head-normal form, so no normal form.

iii) $(\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx)))\,(\lambda ab.a) \;\to_\beta$
$(\lambda x.(\lambda ab.a)(xx))\,(\lambda x.(\lambda ab.a)(xx)) \quad = \quad (with\ \lambda x.(\lambda ab.a)(xx) = P)$
$PP \;\to_\beta\; (\lambda ab.a)(PP) \;\to_\beta\; \lambda b.PP\ etc \;\to_\beta\; \lambda b.\lambda b'.PP\ etc$

No head-normal form, so no normal form.

iv) $(\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx)))\,(\lambda ab.b) \;\to_\beta$
$(\lambda x.(\lambda ab.b)(xx))\,(\lambda x.(\lambda ab.b)(xx)) \quad = \quad (with\ \lambda x.(\lambda ab.b)(xx) = P)$
$PP \;\to_\beta\; (\lambda ab.b)(PP) \;\to_\beta\; \lambda b.b$

Head-normal form, and normal form.

v) $(\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx)))\,(\lambda ab.ab) \;\to_\beta$
$(\lambda x.(\lambda ab.ab)(xx))\,(\lambda x.(\lambda ab.ab)(xx)) \quad = \quad (with\ \lambda x.(\lambda ab.ab)(xx) = P)$
$PP \;\to_\beta\; (\lambda ab.ab)(PP) \;\to_\beta\; \lambda b.PPb \;\to_\beta\; \lambda b.(\lambda b'.PPb')b \;\to_\beta\; \lambda b.PPb\ etc$

No head-normal form, so no normal form.

Exercise 1.31 By induction on the definition of $=_\beta$.

$(M \to_\beta^* N \Rightarrow M =_\beta N)$: Immediate.

$(M =_\beta N \Rightarrow N =_\beta M)$: By induction there exist $M_1, M_2, \ldots, M_n, M_{n+1}$ such that $M \equiv M_1$, $N \equiv M_{n+1}$, and, for all $1 \le i \le n$, either $M_i \to_\beta^* M_{i+1}$, or $M_{i+1} \to_\beta^* M_i$. This same sequence serves for the reversed equation.

$(M =_\beta L \wedge L =_\beta N \Rightarrow M =_\beta N)$: By induction there exist $M_1, M_2, \ldots, M_n, M_{n+1}$ such that $M \equiv M_1$, $L \equiv M_{n+1}$, and, for all $1 \le i \le n$, either $M_i \to_\beta^* M_{i+1}$, or $M_{i+1} \to_\beta^* M_i$, and there exist $L_1, L_2, \ldots, L_m, L_{m+1}$ such that $L \equiv L_1$, $N \equiv L_{n+1}$, and, for all $1 \le i \le n$, either $L_i \to_\beta^* L_{i+1}$, or

$L_{i+1} \to_\beta^* L_i$. Then the sequence $M \equiv M_1, M_2, \ldots, M_n, M_{n+1} \equiv L \equiv L_1, L_2, \ldots, L_m, L_{m+1} \equiv N$ satisfies the criteria.

Exercise 1.32 We have

$$M((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))M) \;\to_\beta\; M((\lambda x.M(xx))(\lambda x.M(xx)))$$

and

$$\begin{array}{rcl}(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))M & \to_\beta & (\lambda x.M(xx))(\lambda x.M(xx)) \\ & \to_\beta & M((\lambda x.M(xx))(\lambda x.M(xx)))\end{array}$$

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

Exercise 1.33 We have

$$\begin{array}{rcl}(\lambda xy.y(xxy))(\lambda xy.y(xxy))M & \to_\beta & (\lambda z.z((\lambda xy.y(xxy))(\lambda xy.y(xxy))z))M \\ & \to_\beta & M((\lambda xy.y(xxy))(\lambda xy.y(xxy))M)\end{array}$$

Exercise 1.34 By induction on the structure of terms in normal form.

($M \equiv xM_1 \cdots M_n$, *with* $n \geq 0$ *and each* M_i *in normal form*): Notice that each M_i is a λ-term, so $xM_1 \cdots M_n$ is in head-nomal form.

($M \equiv \lambda x.N$, *with* N *in normal form*): Then, by induction, N is in head-normal form, so so is $\lambda x.N$.

Exercise 1.35 Assume $M \to_\beta^* N_1$ and $M \to_\beta^* N_2$, and both N_1 and N_2 are in normal form. Then, by the Church-Rosser property, we know there exists Q such that $N_1 \to_\beta^* Q$ and $N_2 \to_\beta^* Q$. However, since both are in normal form, these are empty reductions and, in fact, we have $N_1 \equiv Q \equiv N_2$. So the normal form of M, if it exists, is unique.

Exercise 1.36 *i*) If $A \in \mathcal{A}(xM_1 \cdots M_n)$, then there exists $M'_1, \ldots, M'_n$ such that, for all $1 \leq i \leq n$, $M_i \to_\beta^* M'_i$ and $A \sqsubseteq xM'_1 \cdots M'_n$. Since $A \neq \bot$, there exists $A_1, \ldots, A_n$ such that, for all $1 \leq i \leq n$, $A_i \sqsubseteq M'_i$, and $A = xA_1 \cdots A_n$. Since $A' \in \mathcal{A}(N)$, there exists N' such that $N \to_\beta^* N'$ and $A' \sqsubseteq N'$. But then $xM_1 \cdots M_n N \to_\beta^* xM'_1 \cdots M'_n N'$, and $xA_1 \cdots A_n A' \sqsubseteq xM'_1 \cdots M'_n N'$, so $AA' \in \mathcal{A}(xM_1 \cdots M_n N)$.

ii) Let $A \in \mathcal{A}(Mz)$, then there exists N such that $Mz \to_\beta^* N$ and $A \sqsubseteq N$. Now either:

- $N = M'z$ (so the reductions took place inside M), and $A = \bot$ or $A = A'z$ with $A' \sqsubseteq M'$, so $A' \in \mathcal{A}(M)$. Since $z \notin M$, we have $z \notin M'$, and since A' matches M', also $z \notin A'$.
- During reduction of Mz towards N, M has run into an abstraction, $M \to_\beta^* \lambda x.M'$, and $(\lambda x.M')z \to_\beta M'[z/x] \to_\beta^* N$. Then $A \in \mathcal{A}(M'[z/x])$, so $\lambda z.A \in \mathcal{A}(\lambda z.M'[z/x]) = \mathcal{A}(\lambda x.M') = \mathcal{A}(M)$.

Exercise 1.37 *i*) By induction on the structure of approximants. If $A \sqsubseteq M$ and $M \to_\beta^* N$, then either:

($A = \bot$): Then $A \sqsubseteq N$.

($A = \lambda x.A'$, $A' \neq \bot$): Then $M = \lambda x.M'$ with $A' \sqsubseteq M'$, and $\lambda x.M' \to_\beta^* \lambda x.N'$, so $M' \to_\beta^* N'$. By induction, we have $A' \sqsubseteq N'$, so $\lambda x.A' \sqsubseteq \lambda x.N'$.

($A = x\mathsf{A}_1 \cdots \mathsf{A}_n$): Then $M = xM_1 \cdots M_n$, and, for every $1 \leq i \leq n$, we have $A_i \sqsubseteq M_i$. Since $M \to_\beta^* N$, there are $N_1, \ldots, N_n$ such that $N = xN_1 \cdots N_n$, and, for every $1 \leq i \leq n$, we have $M_i \to_\beta^* N_i$. Then, by induction, for every $1 \leq i \leq n$, we have $A_i \sqsubseteq N_i$, so $x\mathsf{A}_1 \cdots \mathsf{A}_n \sqsubseteq xN_1 \cdots N_n$.

ii) If $A \in \mathcal{A}(M)$, then there exists P such that $M \to_\beta^* P$ and $A \sqsubseteq P$. Since $M \to_\beta^* N$ and $M \to_\beta^* P$ and reduction is confluent, there exists L such that $N \to_\beta^* L$ and $P \to_\beta^* L$. By part (*iii*) we have $A \sqsubseteq L$, so also $A \in \mathcal{A}(N)$.

iii) If $A \in \mathcal{A}(N)$, then there exists P such that $N \to_\beta^* P$ and $A \sqsubseteq P$. Since $M \to_\beta^* N$ and $N \to_\beta^* P$, we have $M \to_\beta^* P$, so $A \in \mathcal{A}(M)$.

Exercise 1.38 By induction on the structure of approximants. If $M_1 \sqsubseteq M$, and $M_2 \sqsubseteq M$, then either

$(M_1 = \bot)$: Notice that $\bot \sqcup M_2 = M_2$, so we have $\bot \sqsubseteq M_2$, $M_2 \sqsubseteq M_2$, and $M_2 \sqsubseteq M$.

$(M_2 = \bot)$: Similar.

$(M_1 = \lambda x.M_1')$: Then $M = \lambda x.M'$, $M_1' \sqsubseteq M'$, and either:

- $(M_2 = \bot)$: Then $M_1 \sqcup M_2 = \lambda x.M_1'$, and $\lambda x.M_1' \sqsubseteq \lambda x.M_1'$, $\bot \sqsubseteq \lambda x.M_1'$, and $\lambda x.M_1' \sqsubseteq \lambda x.M'$.
- $(M_2 = \lambda x.M_2')$: Then $M_2' \sqsubseteq M'$, and by induction $M_1' \sqsubseteq M_1' \sqcup M_2'$, $M_2' \sqsubseteq M_1' \sqcup M_2'$, and $M_1' \sqcup M_2' \sqsubseteq M'$. But then also $\lambda x.M_1' \sqsubseteq \lambda x.M_1' \sqcup \lambda x.M_2'$, $\lambda x.M_2' \sqsubseteq \lambda x.M_1' \sqcup \lambda x.M_2'$, and $\lambda x.M_1' \sqcup \lambda x.M_2' \sqsubseteq \lambda x.M'$.

$(M_1 = xM_1^1 \cdots M_1^n)$: Similar to the previous case.

Exercise 1.39
$$\begin{array}{llllll} \mathsf{Cond\ True}\ M\ N & \triangleq & (\lambda btf.btf)\ (\lambda xy.x)\ M\ N & \to_\beta & (\lambda xy.x)\ M\ N & \to_\beta\ M \\ \mathsf{Cond\ False}\ M\ N & \triangleq & (\lambda btf.btf)\ (\lambda xy.y)\ M\ N & \to_\beta & (\lambda xy.y)\ M\ N & \to_\beta\ N \end{array}$$

Exercise 1.40 $\mathsf{First}\,\langle M,N\rangle \ \triangleq\ (\lambda p.p\,(\lambda xy.x))\,(\lambda z.zMN) \ \to_\beta\ (\lambda z.zMN)\,(\lambda xy.x) \ \to_\beta\ (\lambda xy.x)\,MN \ \to_\beta\ M$

$\mathsf{Second}\,\langle M,N\rangle \ \triangleq\ (\lambda p.p\,(\lambda xy.y))\,(\lambda z.zMN) \ \to_\beta\ (\lambda z.zMN)\,(\lambda xy.y) \ \to_\beta\ (\lambda xy.y)\,MN \ \to_\beta\ N$

Exercise 1.41 $\mathsf{Pred}\ (\mathsf{Succ}\ k) \ \triangleq\ (\langle \mathbf{K},\mathbf{I}\rangle)\,((\lambda nxy.yn)\,k) \ \triangleq\ (\lambda p.p\mathbf{KI})\,((\lambda nxy.yn)\,k) \ \to_\beta\ (\lambda nxy.yn)\ k\ \mathbf{K}\ \mathbf{I}) \ \to_\beta^*\ \mathbf{I}\ k \ \triangleq\ (\lambda x.x)\ k \ \to_\beta\ k$

$\mathsf{Succ}\ (\mathsf{Pred}\ k) \ \triangleq\ (\lambda nxy.yn)\,(\langle \mathbf{K},\mathbf{I}\rangle\ k) \ \to_\beta\ \lambda xy.y\ (\langle \mathbf{K},\mathbf{I}\rangle\ k) \ \triangleq\ \lambda xy.y\ ((\lambda p.p\mathbf{KI})\ k) \ \triangleq\ \lambda xy.y(k\mathbf{KI})$

So no.

Exercise 2.17

$$\dfrac{\overline{x{:}A \vdash_c x : A}\;(Ax)}{\emptyset \vdash_c \lambda x.x : A\to A}\;(\to I)$$

$$\dfrac{\dfrac{\overline{x{:}A,y{:}B \vdash_c x : A}\;(Ax)}{x{:}A \vdash_c \lambda y.x : B\to A}\;(\to I)}{\emptyset \vdash_c \lambda xy.x : A\to B\to A}\;(\to I)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash_c x : A\to B\to C}\;(Ax) \quad \overline{\Gamma \vdash_c z : A}\;(Ax)}{\Gamma \vdash_c xz : B\to C}\;(\to E) \quad \dfrac{\overline{\Gamma \vdash_c y : A\to B}\;(Ax) \quad \overline{\Gamma \vdash_c z : A}\;(Ax)}{\Gamma \vdash_c yz : B}\;(\to E)}{\Gamma = x{:}A\to B\to C, y{:}A\to B, z{:}A \vdash_c xz(yz) : C}\;(\to E)}{x{:}A\to B\to C, y{:}A\to B \vdash_c \lambda z.xz(yz) : A\to C}\;(\to I)}{x{:}A\to B\to C \vdash_c \lambda yz.xz(yz) : (A\to B)\to A\to C}\;(\to I)}{\emptyset \vdash_c \lambda xyz.xz(yz) : (A\to B\to C)\to(A\to B)\to A\to C}\;(\to I)$$

$$\dfrac{\dfrac{\overline{b{:}A,c{:}B \vdash_c c : B}\;(Ax)}{b{:}A \vdash_c \lambda c.c : B\to B}}{\emptyset \vdash_c \lambda bc.c : A\to B\to B}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma,y{:}A \vdash_c c : B}\;(Ax)}{\Gamma \vdash_c \lambda y.c : A\to B}\;(\to I) \quad \dfrac{\overline{\Gamma \vdash_c b : B\to A}\;(Ax) \quad \overline{\Gamma \vdash_c c : B}\;(Ax)}{\Gamma \vdash_c bc : A}\;(\to E)}{\Gamma = b{:}B\to A, c{:}B \vdash_c (\lambda y.c)\,(bc) : B}\;(\to E)}{b{:}B\to A \vdash_c \lambda c.(\lambda y.c)\,(bc) : B\to B}\;(\to I)}{\emptyset \vdash_c \lambda bc.(\lambda y.c)\,(bc) : (B\to A)\to B\to B}\;(\to I)$$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\Gamma,x{:}B,y{:}A \vdash_{\mathrm{c}} x : B}\,(Ax)
}{\Gamma,x{:}B \vdash_{\mathrm{c}} \lambda y.x : A\to B}\,(\to I)
}{\Gamma \vdash_{\mathrm{c}} \lambda xy.x : B\to A\to B}\,(\to I)
\quad
\dfrac{}{\Gamma \vdash_{\mathrm{c}} c : B}\,(Ax)
}{\Gamma \vdash_{\mathrm{c}} (\lambda xy.x)\,c : A\to B}\,(\to I)
\quad
\dfrac{
\dfrac{}{\Gamma \vdash_{\mathrm{c}} b : B\to A}\,(Ax)
\quad
\dfrac{}{\Gamma \vdash_{\mathrm{c}} c : B}\,(Ax)
}{\Gamma \vdash_{\mathrm{c}} bc : A}\,(\to E)
}{\Gamma = b{:}B\to A, c{:}B \vdash_{\mathrm{c}} (\lambda xy.x)c\,(bc) : B}\,(\to E)
}{b{:}B\to A \vdash_{\mathrm{c}} \lambda c.((\lambda xy.x)\,c\,(bc) : B\to B}\,(\to I)
}{\emptyset \vdash_{\mathrm{c}} \lambda bc.(\lambda xy.x)\,c\,(bc) : (B\to A)\to B\to B}\,(\to I)
$$

The last derivation is obtained from the third and the second, applying $(\to E)$ and taking $C = A$.

Exercise 2.18 *i*) By induction on the structure of derivations.

(Ax): Then $M \equiv x$ and $x{:}A \in \Gamma$; since $\Gamma' \supseteq \Gamma$, also $x{:}A \in \Gamma'$, by (Ax) also $\Gamma' \vdash_{\mathrm{c}} x : A$.

$(\to I)$: Then $M \equiv \lambda x.N$, $A = B\to C$, and $\Gamma,x{:}B \vdash_{\mathrm{c}} N : C$; since $\Gamma' \supseteq \Gamma$, also $\Gamma',x{:}B \supseteq \Gamma,x{:}B$, so by induction $\Gamma',x{:}B \vdash_{\mathrm{c}} N : C$. Then $\Gamma' \vdash_{\mathrm{c}} \lambda x.N : B\to C$ follows by $(\to I)$.

$(\to E)$: Straightforward by induction.

ii) By induction on the structure of derivations.

(Ax): Then $M \equiv x$ and $x{:}A \in \Gamma$; notice that $\{\, y{:}D \in \Gamma \mid y \in fv\,(M)\,\} = \{x{:}A\}$, and that by (Ax) also $x{:}A \vdash_{\mathrm{c}} x : A$.

$(\to I)$: Then $M \equiv \lambda x.N$, $A = B\to C$, and $\Gamma,x{:}B \vdash_{\mathrm{c}} N : C$; let $\Gamma' = \{\, y{:}D \in \Gamma \mid y \in fv\,(N)\,\}$ then by induction $\Gamma' \vdash_{\mathrm{c}} N : C$. Now either:

$(x \in fv\,(N))$: Then x appears in Γ'; since $\Gamma \supseteq \Gamma'$, in fact $x{:}B \in \Gamma'$; let $\Gamma'' = \Gamma' \backslash x$.

$(x \notin fv\,(N))$: Then x does not appear in Γ'; by part (*i*), also $\Gamma',x{:}B \vdash_{\mathrm{c}} N : C$; let $\Gamma'' = \Gamma'$.

In either case: $\Gamma'' \vdash_{\mathrm{c}} \lambda x.N : B\to C$ follows by $(\to I)$.

Also: $\Gamma'' = \Gamma' \backslash x = \{\, y{:}D \in \Gamma \mid y \in fv\,(N)\,\} \backslash x = \{\, y{:}D \in \Gamma \mid y \in fv\,(\lambda x.N)\,\}$.

$(\to E)$: Then $M \equiv PQ$ and there exists B such that $\Gamma \vdash_{\mathrm{c}} P : B\to A$ and $\Gamma \vdash_{\mathrm{c}} Q : B$. let $\Gamma_1 = \{\, y{:}D \in \Gamma \mid y \in fv\,(P)\,\}$ and $\Gamma_2 = \{\, y{:}D \in \Gamma \mid y \in fv\,(Q)\,\}$ then by induction $\Gamma_1 \vdash_{\mathrm{c}} P : B\to A$ and $\Gamma_2 \vdash_{\mathrm{c}} N : B$. Since $\Gamma_i \subseteq \Gamma$ for $i = 1,2$, also $\Gamma_1 \cup \Gamma_2 \subseteq \Gamma$; moreover, $\Gamma_i \subseteq \Gamma_1 \cup \Gamma_2$, so by part (*i*) we have $\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{c}} P : B\to A$ and $\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{c}} N : B$, so by $(\to E)$ also $\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{c}} PQ : A$.

$$
\begin{array}{rcl}
\text{Also: } \Gamma_1 \cup \Gamma_2 & = & \{\, y{:}D \in \Gamma \mid y \in fv\,(P)\,\} \cup \{\, y{:}D \in \Gamma \mid y \in fv\,(Q)\,\} \\
& = & \{\, y{:}D \in \Gamma \mid y \in fv\,(P) \vee y \in fv\,(Q)\,\} \\
& = & \{\, y{:}D \in \Gamma \mid y \in fv\,(PQ)\,\}
\end{array}
$$

iii) By induction on the structure of derivations.

(Ax): Then $M \equiv x$ and $x{:}A \in \Gamma$; notice that $x \in fv\,(x)$.

$(\to I)$: Then $M \equiv \lambda x.N$, $A = B\to C$, and $\Gamma,x{:}B \vdash_{\mathrm{c}} N : C$; by induction, for every $y \in fv\,(N)$ there exists D such that $y{:}D \in \Gamma,x{:}B$. Since $fv\,(\lambda x.N) = fv\,(N) \backslash x$, for every $y \in fv\,()Lx.(M)$ there exists D such that $y{:}D \in \Gamma$.

$(\to E)$: Then $M \equiv PQ$ and there exists B such that $\Gamma \vdash_{\mathrm{c}} P : B\to A$ and $\Gamma \vdash_{\mathrm{c}} Q : B$; by induction, for every $y \in fv\,(P)$ there exists D such that $y{:}D \in \Gamma$ and for every $y \in fv\,(Q)$ there exists D such that $y{:}D \in \Gamma$. Since $fv\,(PQ) = fv\,(P) \cup fv\,(Q)$, for every $y \in fv\,(PQ)$ there exists D such that $y{:}D \in \Gamma$.

Exercise 2.19 $(S = (\varphi \mapsto C))$: By Definition 2.10.

$(S = S_1 \circ S_2)$: Since $S_1 \circ S_2(A\to B) = S_1(S_2(A\to B)) = (IH)\ S_1(S_2 A \to S_2 B) = (IH)$ $S_1(S_2 A) \to S_1(S_2 B) = S_1 \circ S_2(A) \to S_1 \circ S_2(B)$.

Exercise 2.20 *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), $S\,\Gamma \vdash_c x : S\,A$.

($\to I$): Then there are N, A, C such that $M \equiv \lambda x.N$, $A = C \to D$, and $\Gamma, x{:}C \vdash_c N : D$. Since this statement is derived in a sub-derivation, we know that $S\,(\Gamma, x{:}C) \vdash_c N : S\,D$ follows by induction. Since $S\,(\Gamma, x{:}C) = S\,\Gamma, x{:}S\,C$, we also have $S\,\Gamma, x{:}S\,C \vdash_c N : S\,D$. So there is a derivation that shows this, to which we can apply rule ($\to I$), to obtain $S\,\Gamma \vdash_c \lambda x.N : S\,C \to S\,D$. Since $S\,C \to S\,D = S\,(C \to D) = S\,A$, by definition of substitutions, we get $S\,\Gamma \vdash_c \lambda x.M' : S\,A$.

($\to E$): Then there are P, Q, and B such that $M \equiv PQ$, $\Gamma \vdash_c P : B \to A$, and $\Gamma \vdash_c Q : B$. Since these two statements are derived in a sub-derivation, by induction both $S\,\Gamma \vdash_c P : S\,(B \to A)$ and $S\,\Gamma \vdash_c Q : S\,B$. Since $S\,(B \to A) = S\,B \to S\,A$ by definition of substitution, we also have $S\,\Gamma \vdash_c P : S\,B \to S\,A$, and we can apply rule ($\to E$) to obtain $S\,\Gamma \vdash_c PQ : S\,A$. □

Exercise 2.21 By induction on the structure of terms.

($M = x$): Then $pp_c\,x = \langle x{:}\varphi;\ \varphi \rangle$. Notice that we have

$$\dfrac{}{x{:}\varphi \vdash_c x : \varphi}$$

($M = \lambda x.N$): Let $pp_c\,N = \langle \Pi;\ P \rangle$; then either:

($x \in \Pi$): Then there exists A such that $\Pi = \Pi', x{:}A$, and by Definition 2.15, $pp_c\,\lambda x.N = \langle \Pi';\ A \to P \rangle$. By induction (there exist a derivation for) $\Pi', x{:}A \vdash_c N : P$. To this derivation we can apply rule ($\to I$) and obtain:

$$\dfrac{\Pi', x{:}A \vdash_c N : P}{\Pi' \vdash_c \lambda x.N : A \to P}\ (\to I)$$

($x \notin \Pi$): Take φ fresh; by Definition 2.15, $pp_c\,\lambda x.N = \langle \Pi;\ \varphi \to P \rangle$, and by induction $\Pi \vdash_c N : P$. Then by Lemma 2.6 also $\Pi, x{:}\varphi \vdash_c N : P$. We can apply rule ($\to I$) and obtain:

$$\dfrac{\Pi, x{:}\varphi \vdash_c N : P}{\Pi \vdash_c \lambda x.N : \varphi \to P}\ (\to I)$$

($M = M_1 M_2$): Let $\langle \Pi_1;\ P_1 \rangle = pp_c\,M_1$ and $\langle \Pi_2;\ P_2 \rangle = pp_c\,M_2$, and φ fresh; by Definition 2.15, $pp_c\,M_1 M_2 = S_2 \circ S_1\,\langle \Pi_1 \cup \Pi_2;\ \varphi \rangle$, where

$$\begin{array}{lcl} \varphi & = & \textit{fresh} \\ S_1 & = & \textit{unify}\ P_1\ P_2 \to \varphi \qquad\qquad \textit{and} \\ S_2 & = & \textit{UnifyContexts}\ (S_1\,\Pi_1)\ (S_1\,\Pi_2) \end{array}$$

By induction we have both $\Pi_1 \vdash_c M_1 : P_1$ and $\Pi_2 \vdash_c M_2 : P_2$; by Lemma 2.11, also

$$S_2 \circ S_1\,\Pi_1 \vdash_c M_1 : S_2 \circ S_1\,P_1 \ \text{ and } \ S_2 \circ S_1\,\Pi_2 \vdash_c M_2 : S_2 \circ S_1\,P_2,$$

and $S_2 \circ S_1\,P_1 = S_2 \circ S_1\,P_2 \to \varphi = S_2 \circ S_1\,P_2 \to S_2 \circ S_1\,\varphi$. By Lemma 2.6 we also have

$$S_2 \circ S_1\,\Pi_1, S_2 \circ S_1\,\Pi_2 \vdash_c M_1 : S_2 \circ S_1\,P_1 \ \text{ and } \ S_2 \circ S_1\,\Pi_1, S_2 \circ S_1\,\Pi_2 \vdash_c M_2 : S_2 \circ S_1\,P_2,$$

so also

$$S_2 \circ S_1\,\Pi_1, \Pi_2 \vdash_c M_1 : S_2 \circ S_1\,P_1 \ \text{ and } \ S_2 \circ S_1\,\Pi_1, \Pi_2 \vdash_c M_2 : S_2 \circ S_1\,P_2.$$

To these we can apply rule ($\to E$) and obtain:

$$\dfrac{S_2 \circ S_1\,\Pi_1, \Pi_2 \vdash_c M_1 : S_2 \circ S_1\,P_2 \to S_2 \circ S_1\,\varphi \qquad S_2 \circ S_1\,\Pi_1, \Pi_2 \vdash_c M_2 : S_2 \circ S_1\,P_2}{S_2 \circ S_1\,\Pi_1, \Pi_2 \vdash_c M_1 M_2 : S_2 \circ S_1\,\varphi}\ (\to E)$$

This completes the proof.

Exercise 2.22
$$\begin{aligned}
pp_c\,\lambda xyz.xz(yz) &= \langle \emptyset, (\varphi_1\to\varphi_2\to\varphi_3)\to(\varphi_1\to\varphi_2)\to\varphi_1\to\varphi_3\rangle,\\
pp_c\,\lambda xy.x &= \langle \emptyset, \varphi_4\to\varphi_5\to\varphi_4\rangle,\\
pp_c\,\lambda xy.xy &= \langle \emptyset, (\varphi_6\to\varphi_7)\to\varphi_6\to\varphi_7\rangle,\\
pp_c\,\lambda x.x &= \langle \emptyset, \varphi_8\to\varphi_8\rangle,
\end{aligned}$$

Let $\Gamma = x{:}\varphi_1\to\varphi_2\to\varphi_3, y{:}\varphi_1\to\varphi_2, z{:}\varphi_1$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\vdash_c x:\varphi_1\to\varphi_2\to\varphi_3}\,(Ax)\quad \dfrac{}{\Gamma\vdash_c z:\varphi_1}\,(Ax)}{\Gamma\vdash_c xz:\varphi_2\to\varphi_3}\,(\to E)\quad \dfrac{\dfrac{}{\Gamma\vdash_c y:\varphi_1\to\varphi_2}\,(Ax)\quad \dfrac{}{\Gamma\vdash_c z:\varphi_1}\,(Ax)}{\Gamma\vdash_c yz:\varphi_2}\,(\to I)}{\Gamma\vdash_c xz(yz):\varphi_3}\,(\to E)}{x{:}\varphi_1\to\varphi_2\to\varphi_3, y{:}\varphi_1\to\varphi_2\vdash_c \lambda z.xz(yz):\varphi_1\to\varphi_3}\,(\to I)}{x{:}\varphi_1\to\varphi_2\to\varphi_3\vdash_c \lambda yz.xz(yz):(\varphi_1\to\varphi_2)\to\varphi_1\to\varphi_3}\,(\to I)}{\vdash_c \lambda xyz.xz(yz):(\varphi_1\to\varphi_2\to\varphi_3)\to(\varphi_1\to\varphi_2)\to\varphi_1\to\varphi_3}\,(\to I)$$

$$\dfrac{\dfrac{}{x{:}\phi_8\vdash_c x:\phi_8}\,(Ax)}{\vdash_c \lambda x.x:\phi_8\to\phi_8}\,(\to I)$$

$$\dfrac{\dfrac{\dfrac{}{x{:}\phi_4, y{:}\phi_5\vdash_c x:\phi_4}\,(Ax)}{x{:}\phi_4\vdash_c \lambda y.x:\phi_5\to\phi_4}\,(\to I)}{\vdash_c \lambda xy.x:\phi_4\to\phi_5\to\phi_4}\,(\to I)
\qquad
\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}\phi_6\to\phi_7, y{:}\phi_6\vdash_c x:\phi_6\to\phi_7}\,(Ax)\quad \dfrac{}{x{:}\phi_6\to\phi_7, y{:}\phi_6\vdash_c y:\phi_6}\,(Ax)}{x{:}\phi_4\vdash_c xy:\phi_7}\,(\to E)}{\vdash_c \lambda y..xy:\phi_6\to\phi_7}\,(\to I)}{\vdash_c \lambda xy.xy:(\phi_6\to\phi_7)\to\phi_6\to\phi_7}\,(\to I)$$

Using *unify* $(\varphi_1\to\varphi_2\to\varphi_3)\to(\varphi_1\to\varphi_2)\to\varphi_1\to\varphi_3\ (\varphi_4\to\varphi_5\to\varphi_4)\to\varphi_6$

$$pp_c\,(\lambda xyz.xz(yz))(\lambda xy.x) \;=\; \langle \emptyset, (\varphi_1\to\varphi_2)\to\varphi_1\to\varphi_1\rangle$$

so, using *unify* $(\varphi_1\to\varphi_2)\to\varphi_1\to\varphi_1\ (\varphi_8\to\varphi_8)\to\varphi_9$

$$pp_c\,(\lambda xyz.xz(yz))(\lambda xy.x)(\lambda x.x) \;=\; \langle \emptyset, \varphi_1\to\varphi_1\rangle$$

Also, using *unify* $(\varphi_6\to\varphi_7)\to\varphi_6\to\varphi_7\ (\varphi_8\to\varphi_8)\to\varphi_{10}$

$$pp_c\,(\lambda xy.xy)(\lambda x.x) \;=\; \langle \emptyset, \varphi_1\to\varphi_1\rangle$$

So $(\lambda xyz.xz(yz))(\lambda xy.x)(\lambda x.x)$ and $(\lambda xy.xy)(\lambda x.x)$ have the same principal types, so the set of assignable types is the same.

Exercise 2.23 No. $(\lambda xyz.xz(yz))(\lambda x.x)(\lambda x.x)((\lambda xyz.xz(yz))(\lambda x.x)(\lambda x.x))\to$
$(\lambda z.(\lambda x.x)z((\lambda x.x)z))(\lambda z.(\lambda x.x)z((\lambda x.x)z))\to$
$(\lambda z.zz)(\lambda z.zz)\to(\lambda z.zz)(\lambda z.zz)\to\cdots$

So the first term is not strongly normalising (in fact, it is unsolvable), and in Curry's system only the strongly normalisable terms receive a type. Also, the term $\lambda z.zz$ contains self-application, which the type system cannot deal with.

Exercise 3.6 *i*) Take the environment and the substitutions

$$\begin{aligned}
\mathcal{E}\mathsf{I} &= 1\to1,\\
\mathcal{E}\mathsf{K} &= 1\to2\to1,\\
\mathcal{E}\mathsf{B} &= (1\to2)\to(3\to1)\to3\to2,\\
\mathcal{E}\mathsf{S} &= (1\to2\to3)\to(1\to2)\to1\to3
\end{aligned}$$

$$\begin{aligned}
S_1 &= 1\mapsto(4\to4)\to5,\ 2\mapsto4\to4,\ 3\mapsto5\\
S_2 &= 1\mapsto5,\ 2\mapsto5,\ 3\mapsto4\to4\\
S_3 &= 1\mapsto5\\
S_4 &= 1\mapsto4\to4,\ 2\mapsto(4\to4)\to5\\
S_5 &= 1\mapsto4
\end{aligned}$$

then we can derive

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{S}:S_1(\mathcal{E}\mathsf{S})}\,(Call)\quad \dfrac{\dfrac{}{\vdash \mathsf{B}:S_2(\mathcal{E}\mathsf{B})}\,(Call)\quad \dfrac{}{\vdash \mathsf{I}:S_3(\mathcal{E}\mathsf{I})}\,(Call)}{\vdash \mathsf{B\,I}:((4\to4)\to5)\to(4\to4)\to5}\,(\to E)}{\vdash \mathsf{S\,(B\,I)}:(((4\to4)\to5)\to4\to4)\to((4\to4)\to5)\to5}\,(\to E)\quad \dfrac{\dfrac{}{\vdash \mathsf{K}:S_4(\mathcal{E}\mathsf{K})}\,(Call)\quad \dfrac{}{\vdash \mathsf{I}:S_5(\mathcal{E}\mathsf{I})}\,(Call)}{\vdash \mathsf{K\,I}:((4\to4)\to5)\to4\to4}\,(\to E)}{\vdash \mathsf{S\,(B\,I)\,(K\,I)}:((4\to4)\to5)\to5}\,(\to E)$$

$$\begin{array}{lrcl@{\quad}l}
ii) & \mathsf{S}\,(\mathsf{B}\,\mathsf{I})\,(\mathsf{K}\,\mathsf{I}) & = & (\lambda xyz.xz(yz))\,(\mathsf{B}\,\mathsf{I})\,(\mathsf{K}\,\mathsf{I}) & \to\ \lambda z.((\mathsf{B}\,\mathsf{I})\,z)\,((\mathsf{K}\,\mathsf{I})\,z) \\
 & & = & \lambda z.(\lambda abc.a(bc))\,\mathsf{I}\,z\,(\mathsf{K}\,\mathsf{I}\,z) & \to\ \lambda z.\mathsf{I}\,(z\,(\mathsf{K}\,\mathsf{I}\,z)) \\
 & & = & \lambda z.(\lambda c.c)\,(z\,(\mathsf{K}\,\mathsf{I}\,z)) & \to\ \lambda z.z\,(\mathsf{K}\,\mathsf{I}\,z) \\
 & & = & \lambda z.z\,((\lambda ab.a)\,\mathsf{I}\,z) & \to\ \lambda z.z\,\mathsf{I}.
\end{array}$$

iii)

$$\dfrac{\dfrac{\dfrac{}{z{:}(4\to4)\to5 \vdash z:(4\to4)\to5}\,(Ax) \qquad \dfrac{}{z{:}(4\to4)\to5 \vdash \mathsf{I}:4\to4}\,(Call)}{z{:}(4\to4)\to5 \vdash z\,\mathsf{I}:5}\,(\to E)}{\vdash \lambda z.z\,\mathsf{I}:((4\to4)\to5)\to5}\,(\to I)$$

Exercise 3.7 The problem is easy when considering just the rules that are part of the definition of $\to_\beta$: they either follow immediately, or by induction. The only addition is the rule '$name \to M$, if $name = M \in Defs$'. Notice that $\langle name\rangle_\lambda = \langle M\rangle_\lambda = M$, and this step follows immediately, since $M \to_\beta^* M$.

Exercise 3.8 Since reduction is defined as an extension of $\to_\beta$, we can copy the proof of Theorem 2.4, and add the case:

($name \to M$, *if* $name = M \in Defs$): By rule $(Call)$, A is a substitution instance of B, where $name{:}B \in \mathcal{E}$, so $A = S\,B$, for some substitution S. By rule $(Defs)$, we have that $\emptyset \vdash_c M : A$, and by Lemma 2.11 also $\emptyset \vdash_c M : S\,A$. Since the derivation rules for $\vdash_c$ have a similar corresponding rule the system for Λ^{N}, also $\emptyset;\mathcal{E} \vdash M : S\,A$, and by weakening $\Gamma;\mathcal{E} \vdash M : S\,A$.

Exercise 4.7 Take

$$\begin{array}{rcrcl}
\mathcal{E} & = & \mathsf{S} & : & (\varphi_1\to\varphi_2\to\varphi_3)\to(\varphi_1\to\varphi_2)\to\varphi_1\to\varphi_3, \\
 & & \mathsf{K} & : & \varphi_1\to\varphi_2\to\varphi_1, \\
 & & \mathsf{I} & : & \varphi_1\to\varphi_1, \\
 & & \mathsf{Y} & : & (\varphi_1\to\varphi_1)\to\varphi_1
\end{array}$$

and $\mathcal{E}'$ as $\mathcal{E}$, but for rec $\mathsf{Y} : (\varphi_1\to\varphi_1)\to\varphi_1$.

We have (with $\Gamma = x{:}1\to2\to3, y{:}1\to2, z{:}1$):

$$D_{\mathsf{S}} = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;\mathcal{E} \vdash x:1\to2\to3}\,(Ax) \quad \dfrac{}{\Gamma;\mathcal{E} \vdash z:1}\,(Ax)}{\Gamma;\mathcal{E} \vdash xz:2\to3}\,(\to E) \quad \dfrac{\dfrac{}{\Gamma;\mathcal{E} \vdash y:1\to2}\,(Ax) \quad \dfrac{}{\Gamma;\mathcal{E} \vdash z:1}\,(Ax)}{\Gamma;\mathcal{E} \vdash yz:2}\,(\to E)}{\Gamma;\mathcal{E} \vdash xz(yz):3}\,(\to E)}{\Gamma\backslash z;\mathcal{E} \vdash \lambda z.xz(yz):1\to3}\,(\to I)}{\Gamma\backslash y,z;\mathcal{E} \vdash \lambda yz.xz(yz):(1\to2)\to1\to3}\,(\to I)}{\emptyset;\mathcal{E} \vdash \lambda xyz.xz(yz):(1\to2\to3)\to(1\to2)\to1\to3}\,(\to I)$$

$$D_{\mathsf{K}} = \dfrac{\dfrac{\dfrac{}{x{:}1,y{:}2;\mathcal{E} \vdash x:1}\,(Ax)}{x{:}1;\mathcal{E} \vdash \lambda y.x:2\to1}\,(\to I)}{\emptyset;\mathcal{E} \vdash \lambda xy.x:1\to2\to1}\,(\to I)$$

$$D_{\mathsf{Y}} = \dfrac{\dfrac{\dfrac{}{m{:}1\to1;\mathcal{E}' \vdash m:1\to1}\,(Ax) \quad \begin{array}{c}\dfrac{\dfrac{}{m{:}1\to1;\mathcal{E}' \vdash \mathsf{Y}:(1\to1)\to1}\,(Rec\ Call) \quad \dfrac{}{m{:}1\to1;\mathcal{E}' \vdash m:1\to1}\,(Ax)}{m{:}1\to1;\mathcal{E}' \vdash \mathsf{Y}m:1}\,(\to E)\\ \vdots\end{array}}{m{:}1\to1;\mathcal{E}' \vdash m(\mathsf{Y}m):1}\,(\to E)}{\emptyset;\mathcal{E}' \vdash \lambda m.m(\mathsf{Y}m):(1\to1)\to1}\,(\to I)$$

$$D_{\mathsf{I}} = \dfrac{\begin{array}{c}\dfrac{\dfrac{}{\emptyset;\mathcal{E} \vdash \mathsf{S}:(1\to(2\to1)\to1)\to(1\to2\to1)\to1\to1}\,(Call) \quad \dfrac{}{\emptyset;\mathcal{E} \vdash \mathsf{K}:1\to(2\to1)\to1}\,(Call)}{\emptyset;\mathcal{E} \vdash \mathsf{SK}:(1\to2\to1)\to1\to1}\,(\to E)\\ \vdots\end{array} \quad \dfrac{}{\emptyset;\mathcal{E} \vdash \mathsf{K}:1\to2\to1}\,(Call)}{\emptyset;\mathcal{E} \vdash \mathsf{SKK}:1\to1}\,(\to E)$$

$$D = \dfrac{\dfrac{}{\emptyset;\mathcal{E} \vdash \mathsf{Y} : (\varphi\to\varphi)\to\varphi}\,(Call) \qquad \dfrac{}{\emptyset;\mathcal{E}\vdash \mathsf{I} : \varphi\to\varphi}\,(Call)}{\emptyset;\mathcal{E}\vdash \mathsf{YI} : \varphi}\,(\to E)$$

Take $A = (1\to2\to3)\to(1\to2)\to1\to3$, $B = 1\to2\to1$, $C = 1\to1$, and $D = (1\to1)\to1$, the types derived above, then we can construct:

$$D' = \dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset;\mathcal{E}\vdash\epsilon:\Diamond}\,(\epsilon) \quad \dfrac{\boxed{D_{\mathsf{S}}}}{\emptyset;\mathcal{E}\vdash \lambda xyz.xz(yz):A}}{\emptyset;\mathcal{E}\vdash \mathsf{S} = \lambda xyz.xz(yz):\Diamond}\,(Def) \quad \dfrac{\boxed{D_{\mathsf{K}}}}{\emptyset;\mathcal{E}\vdash\lambda xy.x : B}}{\emptyset;\mathcal{E}\vdash \mathsf{S}=\lambda xyz.xz(yz);\ \mathsf{K}=\lambda xy.x:\Diamond}\,(Def) \quad \dfrac{\boxed{D_{\mathsf{I}}}}{\emptyset;\mathcal{E}\vdash \mathsf{SKK}:C}}{\emptyset;\mathcal{E}\vdash \mathsf{S}=\lambda xyz.xz(yz);\ \mathsf{K}=\lambda xy.x;\ \mathsf{I}=\mathsf{SKK}:\Diamond}\,(Def)$$

$$\dfrac{\dfrac{\dfrac{\boxed{D'}}{\emptyset;\mathcal{E}\vdash \mathsf{S}=\lambda xyz.xz(yz);\ \mathsf{K}=\lambda xy.x;\ \mathsf{I}=\mathsf{SKK}:\Diamond} \quad \dfrac{\boxed{D_{\mathsf{Y}}}}{\emptyset;\mathcal{E}'\vdash \lambda m.m(\mathsf{Y}m):D}}{\emptyset;\mathcal{E}\vdash \mathsf{S}=\lambda xyz.xz(yz);\ \mathsf{K}=\lambda xy.x;\ \mathsf{I}=\mathsf{SKK};\ \mathsf{rec}\ \mathsf{Y}=\lambda m.m(\mathsf{Y}m):\Diamond}\,(Rec\ Def) \quad \dfrac{\boxed{D}}{\emptyset;\mathcal{E}\vdash \mathsf{YI}:\varphi}}{\emptyset;\mathcal{E}\vdash \mathsf{S}=\lambda xyz.xz(yz);\ \mathsf{K}=\lambda xy.x;\ \mathsf{I}=\mathsf{SKK};\ \mathsf{rec}\ \mathsf{Y}=\lambda m.m(\mathsf{Y}m):\mathsf{YI}:\varphi}\,(Prog)$$

As for the final part:

$$\begin{array}{llllll} \mathsf{Y\,I} & \to & (\lambda m.m(\mathsf{Y}m))\mathsf{I} & \to & \mathsf{I}\,(\mathsf{Y\,I}) & \\ & \to & \mathsf{S\,K\,K}\,(\mathsf{Y\,I}) & \to & (\lambda xyz.xz(yz))\,\mathsf{K\,K}\,(\mathsf{Y\,I}) & \\ & \to^* & \mathsf{K}\,(\mathsf{Y\,I})\,(\mathsf{K}\,(\mathsf{Y\,I})) & \to & (\lambda xy.x)\,(\mathsf{Y\,I})\,(\mathsf{K}\,(\mathsf{Y\,I})) & \to^*\ \mathsf{Y\,I} \end{array}$$

Exercise 4.8 Take
$$\mathcal{E} = \begin{array}{lll} \mathsf{E} & : & (\varphi_1\to\varphi_2)\to\varphi_1\to\varphi_2 = E, \\ \mathsf{S} & : & (\varphi_1\to\varphi_2\to\varphi_3)\to(\varphi_1\to\varphi_2)\to\varphi_1\to\varphi_3 = F, \\ \mathsf{Y} & : & (\varphi_1\to\varphi_1)\to\varphi_1 = G \end{array}$$
and $\mathcal{E}'$ as $\mathcal{E}$, but for $\mathsf{rec}\ \mathsf{Y} : (\varphi_1\to\varphi_1)\to\varphi_1$. Take D_{Y} and D_{S} as before, and

$$D_E = \dfrac{\dfrac{\dfrac{\dfrac{}{x{:}1\to2,y{:}1;\mathcal{E}\vdash x:1\to2}\,(Ax) \quad \dfrac{}{x{:}1\to2,y{:}1;\mathcal{E}\vdash y:1}\,(Ax)}{x{:}1\to2,y{:}2;\mathcal{E}\vdash xy:2}\,(\to E)}{x{:}1\to2;\mathcal{E}\vdash \lambda y.xy:1\to2}\,(\to I)}{\emptyset;\mathcal{E}\vdash\lambda xy.xy:(1\to2)\to1\to2}\,(\to I)$$

Take $C = (A\to A)\to A$, $E = (1\to2)\to1\to2$, $F = (1\to2\to3)\to(1\to2)\to1\to3$, $G = (1\to1)\to1$, and $D =$

$$\dfrac{\dfrac{}{\vdash \mathsf{Y}:(C\to C)\to C}\,(Call) \quad \dfrac{\dfrac{}{\emptyset;\mathcal{E}\vdash \mathsf{S}:((A\to A)\to A\to A)\to C\to C}\,(Call) \quad \begin{array}{c}\dfrac{\dfrac{}{\emptyset;\mathcal{E}\vdash \mathsf{E}:((A\to A)\to A\to A)\to(A\to A)\to A\to A}\,(Call) \quad \dfrac{}{\emptyset;\mathcal{E}\vdash\mathsf{E}:(A\to A)\to A\to A}\,(Call)}{\emptyset;\mathcal{E}\vdash \mathsf{E\,E}:(A\to A)\to A\to A}\,(\to E)\\ \vdots\end{array}}{\emptyset;\mathcal{E}\vdash \mathsf{S}\,(\mathsf{E\,E}):C\to C}\,(\to E)}{\emptyset;\mathcal{E}\vdash \mathsf{Y}(\mathsf{S}\,(\mathsf{E\,E})):C}\,(\to E)$$

then we can construct:

$$\dfrac{\begin{array}{c}\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset;\mathcal{E}\vdash\epsilon:\Diamond}\,(\epsilon) \quad \dfrac{\boxed{D_E}}{\emptyset;\mathcal{E}\vdash\lambda xy.xy:E}}{\emptyset;\mathcal{E}\vdash \mathsf{E}=\lambda xy.xy:\Diamond}\,(Def) \quad \dfrac{\boxed{D_{\mathsf{S}}}}{\emptyset;\mathcal{E}\vdash\lambda xyz.xz(yz):F}}{\emptyset;\mathcal{E}\vdash \mathsf{E}=\lambda xy.xy;\ \mathsf{S}=\lambda xyz.xz(yz):\Diamond}\,(Def) \quad \dfrac{\boxed{D_{\mathsf{Y}}}}{\emptyset;\mathcal{E}'\vdash\lambda m.m(\mathsf{Y}m):G}}{\emptyset;\mathcal{E}\vdash \mathsf{E}=\lambda xy.xy;\ \mathsf{S}=\lambda xyz.xz(yz);\ \mathsf{rec}\ \mathsf{Y}=\lambda m.m(\mathsf{Y}m):\Diamond}\,(Rec\ Def)\\ \vdots\end{array} \quad \dfrac{\boxed{D}}{\emptyset;\mathcal{E}\vdash \mathsf{Y}(\mathsf{S}\,(\mathsf{E\,E})):C}}{\emptyset;\mathcal{E}\vdash \mathsf{E}=\lambda xy.xy;\ \mathsf{S}=\lambda xyz.xz(yz);\ \mathsf{rec}\ \mathsf{Y}=\lambda m.m(\mathsf{Y}m):\mathsf{Y}(\mathsf{S}\,(\mathsf{E\,E})):C}\,(Prog)$$

Exercise 5.15 To show: If $\Gamma \vdash_{\text{ML}} E : \psi$, then, for every substitution S, $S\Gamma \vdash_{\text{ML}} E : S\psi$. We prove this by induction on the structure of derivations:

(Ax): Then $E \equiv x$, and $x{:}\psi \in \Gamma$. Then $x{:}S\psi \in S\Gamma$, so $S\Gamma \vdash_{\text{ML}} x : S\psi$, by rule (Ax).

$(\to I)$: Then $E \equiv \lambda x.E'$, there are $A,B \in \mathcal{T}_{\text{C}}$ such that $\psi = A{\to}B$, and $\Gamma,x{:}A \vdash_{\text{ML}} E' : B$. By induction, $S(\Gamma,x{:}A) \vdash_{\text{ML}} E' : SB$. Since $S(\Gamma,x{:}A) = S\Gamma,x{:}SA$, also $S\Gamma,x{:}SA \vdash_{\text{ML}} E' : SB$. Then, by rule $(\to I)$, we obtain $S\Gamma \vdash_{\text{ML}} \lambda x.E' : SA{\to}SB$. Since $SA{\to}SB = S(A{\to}B)$, and $S(A{\to}B) = S\psi$, we get $S\Gamma \vdash_{\text{ML}} \lambda x.E' : S\psi$.

$(\to E)$: Then $E \equiv E_1E_2$, and there exists $A \in \mathcal{T}_{\text{C}}$ such that $\Gamma \vdash_{\text{ML}} E_1 : A{\to}\psi$, and $\Gamma \vdash_{\text{ML}} E_2 : A$ (also $\psi \in \mathcal{T}_{\text{C}}$). Then, by induction, $S\Gamma \vdash_{\text{ML}} E_1 : S(A{\to}\psi)$ and $S\Gamma \vdash_{\text{ML}} E_2 : SA$. Since $S(A{\to}\psi) = SA{\to}S\psi$, so we also have $S\Gamma \vdash_{\text{ML}} E_1 : SA{\to}S\psi$, and we can apply rule $(\to E)$ to obtain $S\Gamma \vdash_{\text{ML}} E_1E_2 : S\psi$.

$(\mathcal{C})$: Then $E = c$, and $\tau = \nu c$. By rule $(\mathcal{C})$, $\Gamma \vdash_{\text{ML}} c : \tau$. Since τ has no free variables, substitution does not affect that type.

(let): Then $E \equiv \mathsf{let}\ x = E_2\ \mathsf{in}\ E_1, \psi \in \mathcal{T}_{\text{C}}$, and there exist χ such that $\Gamma,x{:}\chi \vdash_{\text{ML}} E_1 : \psi$ and $\Gamma \vdash_{\text{ML}} E_2 : \chi$. By induction, $S(\Gamma,x{:}\chi) \vdash_{\text{ML}} E_1 : S\psi$ and $S\Gamma \vdash_{\text{ML}} E_2 : S\chi$. As before, since $S(\Gamma,x{:}\chi) = S\Gamma,x{:}S\chi$, also $S\Gamma,x{:}S\chi \vdash_{\text{ML}} E_1 : S\psi$. Then, by applying rule (let), $S\Gamma \vdash_{\text{ML}} \mathsf{let}\ x = E_2\ \mathsf{in}\ E_1 : S\psi$.

(fix): Then $E \equiv \mathsf{fix}\,g.E$, $\psi \in \mathcal{T}_{\text{C}}$, and $\Gamma,g{:}\psi \vdash_{\text{ML}} E : \psi$. Then $S(\Gamma,g{:}\psi) \vdash_{\text{ML}} E : S\psi$ by induction, and since $S(\Gamma,g{:}\psi) = S\Gamma,g{:}S\psi$, we also have $S\Gamma,Sg{:}\psi \vdash_{\text{ML}} E : S\psi$. So, by rule (fix), also $S\Gamma \vdash_{\text{ML}} \mathsf{fix}\,g.E : S\psi$.

$(\forall I)$: Then $\psi = \forall\varphi.\chi$, and $\Gamma \vdash_{\text{ML}} E : \chi$, provided φ does not occur free in Γ. By induction, $S\Gamma \vdash_{\text{ML}} E : S\chi$. Without loss of generality, we can assume that φ does not occur in the range of S, so does not occur free in $S\Gamma$. So, by rule $(\forall I)$, $S\Gamma \vdash_{\text{ML}} E : \forall\varphi.S\chi$. Since $\forall\varphi.S\chi = S(\forall\varphi.\chi) = S\psi$, we get $S\Gamma \vdash_{\text{ML}} E : S\psi$.

$(\forall E)$: Then $\psi = \chi[A/\varphi]$, and $\Gamma \vdash_{\text{ML}} E : \forall\varphi.\chi$, and, by induction, $S\Gamma \vdash_{\text{ML}} E : S\forall\varphi.\chi$. Then, by rule $(\forall E)$, also $S\Gamma \vdash_{\text{ML}} E : (S\chi)[(SA)/\varphi]$. Without loss of generality, we can assume that φ does not occur in the domain of S, so if it occurs in χ, then it occurs in $S\chi$. So $(S\chi)[(SA)/\varphi] = S(\chi[A/\varphi]) = S\psi$, and we get $S\Gamma \vdash_{\text{ML}} E : S\psi$.

Exercise 5.16 Again, this is an extension of Theorem 2.4. We only show the added cases; before we come to this, however, we need to show:

Lemma If $\Gamma,x{:}\tau \vdash_{\text{ML}} E_1 : A$ and $\Gamma \vdash_{\text{ML}} E_2 : \tau$, then $\Gamma \vdash_{\text{ML}} E_2[E_1/x] : A$.

The added cases are:

$(\mathit{let}\ x = E_1\ \mathsf{in}\ E_2 \to_{\text{ML}} E_2[E_1/x])$: Assume $\Gamma \vdash_{\text{ML}} \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : A$, then by Lemma 5.8 there exists B,τ such that $\Gamma,x{:}\tau \vdash_{\text{ML}} E_1 : B$, and $\Gamma \vdash_{\text{ML}} E_2 : \tau$, and $B = \forall\overrightarrow{\varphi_i}.A$, with each φ_i not in Γ. Then by Lemma 8, we get $\Gamma \vdash_{\text{ML}} E_2[E_1/x] : A$, and by applying rule $(\forall I)$ repeatedly, we get $\Gamma \vdash_{\text{ML}} E_2[E_1/x] : \sigma$.

$(\mathit{fix}\,g.E \to_{\text{ML}} E[(\mathit{fix}\,g.E)/g])$: Assume $\Gamma \vdash_{\text{ML}} \mathsf{fix}\,g.E : \sigma$, then by Lemma 5.8 there exists A such that $\Gamma,g{:}A \vdash_{\text{ML}} E : A$, and $\sigma = \forall\overrightarrow{\varphi_i}.A$, with each φ_i not in Γ. Applying rule (fix) to $\Gamma,g{:}A \vdash_{\text{ML}} E : A$ gives us a derivation for $\Gamma \vdash_{\text{ML}} \mathsf{fix}\,g.E : A$; so we have both $\Gamma,g{:}A \vdash_{\text{ML}} E : A$ and $\Gamma \vdash_{\text{ML}} \mathsf{fix}\,g.E : A$, so by Lemma 8, we get $\Gamma \vdash_{\text{ML}} E[\mathsf{fix}\,g.E/g] : A$, and by applying rule $(\forall I)$ repeatedly, we get $\Gamma \vdash_{\text{ML}} E[\mathsf{fix}\,g.E/g] : \sigma$

Exercise 5.17 $\mathsf{Times} = \mathsf{fix}\,t.\lambda nm.\mathsf{Cond}(\mathsf{IsZero}\ n)\ 0\ (\mathsf{Add}\ (t\ (\mathsf{MinusOne}\ n)\ m)\ m)$

Let $\Gamma = t{:}I{\to}I{\to}I, n{:}I, m{:}I$, $C = \mathsf{Cond}$, and

$$\mathcal{D}' = \dfrac{\dfrac{\dfrac{}{\Gamma\vdash\mathsf{Add}:I\to I\to I}\quad\dfrac{\dfrac{\dfrac{}{\Gamma\vdash t:I\to I\to I}\quad\dfrac{\dfrac{}{\Gamma\vdash\mathsf{MinusOne}:I\to I}\quad\dfrac{}{\Gamma\vdash n:I}}{\Gamma\vdash\mathsf{MinusOne}\ n:I}}{\Gamma\vdash t\ (\mathsf{MinusOne}\ n):I\to I}\quad\dfrac{}{\Gamma\vdash m:I}}{\Gamma\vdash t\ (\mathsf{MinusOne}\ n)\ m:I}}{\Gamma\vdash\mathsf{Add}\ (t\ (\mathsf{MinusOne}\ n)\ m):I\to I}\quad\dfrac{}{\Gamma\vdash m:I}}{\Gamma\vdash\mathsf{Add}\ (t\ (\mathsf{MinusOne}\ n)\ m)\ m:I}\ .$$

$$\mathcal{D} = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\vdash C\ :\mathsf{Bool}\to I\to I\to I}\quad\dfrac{\dfrac{}{\Gamma\vdash\mathsf{IsZero}:I\to\mathsf{Bool}}\quad\dfrac{}{\Gamma\vdash n:I}}{\Gamma\vdash\mathsf{IsZero}\ n:\mathsf{Bool}}}{\Gamma\vdash C\ (\mathsf{IsZero}\ n):I\to I\to I}\quad\dfrac{}{\Gamma\vdash 0:I}}{\Gamma\vdash C\ (\mathsf{IsZero}\ n)\ 0:I\to I}\quad\begin{array}{c}\boxed{\mathcal{D}'}\\ \Gamma\vdash\mathsf{Add}\ (t\ (\mathsf{MinusOne}\ n)\ m)\ m:I\end{array}}{\Gamma\vdash C\ (\mathsf{IsZero}\ n)\ 0\ (\mathsf{Add}\ (t\ (\mathsf{MinusOne}\ n)\ m)\ m):I}}{t{:}I\to I\to I,n{:}I\vdash\lambda m.C\ (\mathsf{IsZero}\ n)\ 0\ (\mathsf{Add}\ (t\ (\mathsf{MinusOne}\ n)\ m)\ m):I\to I}}{t{:}I\to I\to I\vdash\lambda nm.C\ (\mathsf{IsZero}\ n)\ 0\ (\mathsf{Add}\ (t\ (\mathsf{MinusOne}\ n)\ m)\ m):I\to I\to I}}{\emptyset\vdash\mathsf{fix}\,t.\lambda nm.C\ (\mathsf{IsZero}\ n)\ 0\ (\mathsf{Add}\ (t\ (\mathsf{MinusOne}\ n)\ m)\ m):I\to I\to I}$$

Exercise 5.18 $\mathsf{Squares}\ =\ (\mathsf{fix}\,s.\lambda n.\mathsf{Cons}\ (\mathsf{Times}\ n\ n)\ (s\ (\mathsf{Add}\ n\ 1)))\ 1$

Let $\Gamma = s{:}I\to[I],n{:}I,$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\vdash C:I\to[I]\to[I]}\quad\dfrac{\dfrac{\begin{array}{c}\boxed{\mathcal{D}}\\ \Gamma\vdash\mathsf{Times}:I\to I\to I\end{array}\quad\dfrac{}{\Gamma\vdash n:I}}{\Gamma\vdash\mathsf{Times}\ n:I\to I}\quad\dfrac{}{\Gamma\vdash n:I}}{\Gamma\vdash\mathsf{Times}\ n\ n:I}}{\Gamma\vdash C\ (\mathsf{Times}\ n\ n):[I]\to[I]}\quad\dfrac{\dfrac{}{\Gamma\vdash s:I\to[I]}\quad\dfrac{\dfrac{\dfrac{}{\Gamma\vdash\mathsf{Add}:I\to I\to I}\quad\dfrac{}{\Gamma\vdash n:I}}{\Gamma\vdash\mathsf{Add}\ n:I\to I}\quad\dfrac{}{\Gamma\vdash 1:I}}{\Gamma\vdash\mathsf{Add}\ n\ 1:I}}{\Gamma\vdash s\ (\mathsf{Add}\ n\ 1):[I]}}{\Gamma\vdash C\ (\mathsf{Times}\ n\ n)\ (s\ (\mathsf{Add}\ n\ 1)):[I]}}{s{:}I\to[I]\vdash\lambda n.C\ (\mathsf{Times}\ n\ n)\ (s\ (\mathsf{Add}\ n\ 1)):I\to[I]}}{\vdash\mathsf{fix}\,s.\lambda n.C\ (\mathsf{Times}\ n\ n)\ (s\ (\mathsf{Add}\ n\ 1)):I\to[I]}\quad\dfrac{}{\vdash 1:I}}{\vdash(\mathsf{fix}\,s.\lambda n.C\ (\mathsf{Times}\ n\ n)\ (s\ (\mathsf{Add}\ n\ 1)))\ 1:[I]}$$

Exercise 5.19 $\mathsf{fix}\,f\,n.\mathsf{Cond}\ (\mathsf{IsZero}\ n)\ \mathsf{then}\ 1\ \mathsf{else}\ (\mathsf{Times}\ n\ (f\ (\mathsf{MinusOne}\ n)))$. Take $\Gamma = f{:}I\to I,n{:}I$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\vdash{=}0:I\to\mathsf{Bool}}\quad\dfrac{}{\Gamma\vdash n:I}}{\Gamma\vdash{=}0\ n:\mathsf{Bool}}\quad\dfrac{}{\Gamma\vdash 1:I}\quad\begin{array}{c}\dfrac{\dfrac{\dfrac{}{\Gamma\vdash\times:I\to I\to I}\quad\dfrac{}{\Gamma\vdash n:I}}{\Gamma\vdash\times\ n:I\to I}\quad\dfrac{\dfrac{}{\Gamma\vdash f:I\to I}\quad\dfrac{\dfrac{}{\Gamma\vdash -1:I\to I}\quad\dfrac{}{\Gamma\vdash n:I}}{\Gamma\vdash -1\ n:I}}{\Gamma\vdash f\ (-1\ n):I}}{\Gamma\vdash\times\ n\ (f\ n):I}\\ \vdots\end{array}}{\Gamma\vdash\mathsf{Cond}\ ({=}0\ n)\ \mathsf{then}\ 1\ \mathsf{else}\ (\times\ n\ (f\ n)):I}}{f{:}I\to I\vdash\lambda n\ .\ \mathsf{Cond}\ ({=}0\ n)\ \mathsf{then}\ 1\ \mathsf{else}\ (\times\ n\ (f\ n)):I\to I}}{\emptyset\vdash\mathsf{fix}\ f\ .\ \lambda n\ .\ \mathsf{Cond}\ ({=}0\ n)\ \mathsf{then}\ 1\ \mathsf{else}\ (\times\ n\ (f\ n)):I\to I}$$

Exercise 5.20 Take $\mathsf{Fac}\ =\ \mathsf{fix}\ f\ .\ \lambda n\ .\ \mathsf{Cond}\ (\mathsf{IsZero}\ n)\ \mathsf{then}\ 1\ \mathsf{else}\ (\mathsf{Times}\ n\ (f\ (\mathsf{Pred}\ n)))$.
Then let $\mathsf{Facs} = \mathsf{fix}\,g.\lambda n.\mathsf{Cons}\ (\mathsf{Fac}\ (\mathsf{Succ}\,n))\ (g\ n)$ in $\mathsf{Facs}\ 0$. Take $\Gamma = g{:}I\to[\mathsf{Int}],n{:}I$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\vdash \mathsf{Cons}:I\to[I]\to[I]} \quad \dfrac{\dfrac{}{\Gamma\vdash\mathsf{Fac}:I\to I}\quad\dfrac{}{\Gamma\vdash n:I}}{\Gamma\vdash \mathsf{Fac}\;n:I}}{\Gamma\vdash\mathsf{Cons}\;(\mathsf{Fac}\;n):[I]\to[I]} \quad \dfrac{\dfrac{}{\Gamma\vdash g:I\to[I]}\quad \dfrac{\dfrac{}{\Gamma\vdash \mathsf{Succ}:I\to I}\quad\dfrac{}{\Gamma\vdash n:I}}{\Gamma\vdash \mathsf{Succ}\,n:I}}{\Gamma\vdash g\;(\mathsf{Succ}\,n):[I]}}{\Gamma = g{:}I\to[I],n{:}I\vdash \mathsf{Cons}\;(\mathsf{Fac}\;n)\;(g\;(\mathsf{Succ}\,n)):[I]}}{g{:}I\to[I]\vdash \lambda n\;.\;\mathsf{Cons}\;(\mathsf{Fac}\;n)\;(g\;(\mathsf{Succ}\,n)):I\to[I]}}{\emptyset\vdash \mathsf{fix}g.\lambda n.\mathsf{Cons}\;(\mathsf{Fac}\;n)\;(g\;(\mathsf{Succ}\,n)):I\to[I]} \quad \dfrac{\dfrac{}{f{:}I\to[I]\vdash f:I\to[I]} \quad \begin{array}{c}\dfrac{}{f{:}I\to[I]\vdash 0:I}\\ \vdots\end{array}}{f{:}I\to[I]\vdash f\;0:[I]}}{\emptyset\vdash \mathsf{let}\;f = \mathsf{fix}g.\lambda n.\mathsf{Cons}\;(\mathsf{Fac}\;n)\;(g\;(\mathsf{Succ}\,n))\;\mathsf{in}\;f\;0:[I]}$$

Exercise 5.21 Take $A = \varphi_4\to\varphi_5\to\varphi_4$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\vdash r:A\to A\to A}\,(Ax) \quad \dfrac{\dfrac{\dfrac{}{\Gamma\vdash r:A\to A\to A}\,(Ax)\quad\dfrac{}{\Gamma\vdash y:A}\,(Ax)}{\Gamma\vdash r\,y:A\to A}\,(\to E) \quad \dfrac{\dfrac{\dfrac{}{\Gamma,a{:}\varphi_4,b{:}\varphi_5\vdash a:\varphi_4}\,(Ax)}{\Gamma,a{:}\varphi_4\vdash \lambda b.a:\varphi_5\to\varphi_4}\,(\to I)}{\Gamma\vdash\lambda ab.a:A}\,(\to I)}{\Gamma\vdash r\,y\,(\lambda ab.a):A}\,(\to E)}{\Gamma\vdash r\,(r\,y\,(\lambda ab.a)):A\to A}\,(\to E) \quad \dfrac{}{\Gamma\vdash x:A}\,(Ax)}{\Gamma = r{:}A\to A\to A,x{:}A,y{:}A\vdash r\,(r\,y\,(\lambda ab.a))\,x:A}\,(\to E)}{r{:}A\to A\to A,x{:}A\vdash \lambda y.r\,(r\,y\,(\lambda ab.a))\,x:A\to A}\,(\to I)}{r{:}A\to A\to A\vdash \lambda xy.r\,(r\,y\,(\lambda ab.a))\,x:A\to A\to A}\,(\to I)}{\emptyset\vdash \mathsf{fix}\,r\,.\,\lambda xy.r\,(r\,y\,(\lambda ab.a))\,x:A\to A\to A}\,(\mathsf{fix})$$

Exercise 6.27 *i*) We need to check that the principal pair for the left-hand side is a valid pair for the right-hand side, whilst the defining occurrence is typed with its environment type. For the first rule, we calculate *pp* $\mathsf{S}\,(\mathsf{K}\,x)\,(\mathsf{K}\,y)\;\mathcal{E}$ and get $\langle x{:}2\to3,y{:}2;\,1\to3\rangle$. Taking $\Gamma_1 = x{:}2\to3,y{:}2$ we can derive

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1\vdash\mathsf{S}:\mathcal{E}\,\mathsf{S}}\quad\dfrac{\dfrac{}{\Gamma_1\vdash\mathsf{K}:(2\to3)\to1\to2\to3}\quad\dfrac{}{\Gamma_1\vdash x:2\to3}}{\Gamma_1\vdash\mathsf{K}\,x:1\to2\to3}}{\Gamma_1\vdash\mathsf{S}\,(\mathsf{K}\,x):(1\to2)\to1\to3}\quad\dfrac{\dfrac{}{\Gamma_1\vdash\mathsf{K}:2\to1\to2}\quad\dfrac{}{\Gamma_1\vdash y:2}}{\Gamma_1\vdash\mathsf{K}\,y:1\to2}}{\Gamma_1\vdash\mathsf{S}\,(\mathsf{K}\,x)\,(\mathsf{K}\,y):1\to3}$$

Notice that S is typed with $\mathcal{E}\,\mathsf{S}$, and that the types for the two occurrences of K are instances of $\mathcal{E}\,\mathsf{K}$. We can now type the right-hand side using Γ_1 and $1\to3$:

$$\dfrac{\dfrac{}{\Gamma_1\vdash\mathsf{K}:3\to1\to3}\quad\dfrac{\dfrac{}{\Gamma_1\vdash x:2\to3}\quad\dfrac{}{\Gamma_1\vdash y:2}}{\Gamma_1\vdash x\,y:3}}{\Gamma_1\vdash\mathsf{K}\,(x\,y):1\to3}$$

so the rule is typeable.

For the second rule, *pp* $\mathsf{S}\,(\mathsf{K}\,x)\,y\;\mathcal{E} = \langle x{:}2\to3;\,y{:}1\to2,1\to3\rangle$; take $\Gamma_2 = x{:}2\to3,y{:}1\to2$

$$\dfrac{\dfrac{\dfrac{}{\Gamma_2\vdash\mathsf{S}:\mathcal{E}\,\mathsf{S}}\quad\dfrac{\dfrac{}{\Gamma_2\vdash\mathsf{K}:(2\to3)\to1\to2\to3}\quad\dfrac{}{\Gamma_2\vdash x:2\to3}}{\Gamma_2\vdash\mathsf{K}\,x:1\to2\to3}}{\Gamma_2\vdash\mathsf{S}\,(\mathsf{K}\,x):(1\to2)\to1\to3}\quad\dfrac{}{\Gamma_2\vdash y:1\to2}}{\Gamma_2\vdash\mathsf{S}\,(\mathsf{K}\,x)\,y:1\to3}$$

and for the right-hand side:

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash \mathsf{B} : (2\to3)\to(1\to2)\to1\to3} \quad \dfrac{}{\Gamma_1 \vdash x : 2\to3}}{\Gamma_1 \vdash \mathsf{B}\,x : (1\to2)\to1\to3} \quad \dfrac{}{\Gamma_1 \vdash y : 1\to2}}{\Gamma_1 \vdash \mathsf{B}\,x\,y : 1\to3}$$

so the rule is typeable.

For the third rule, take $\Gamma_3 = x{:}1\to2\to3, y{:}2$

$$\dfrac{\dfrac{\dfrac{}{\Gamma_3 \vdash \mathsf{S} : \mathcal{E}\,\mathsf{S}} \quad \dfrac{}{\Gamma_3 \vdash x : 1\to2\to3}}{\Gamma_3 \vdash \mathsf{S}\,x : (1\to2)\to1\to3} \quad \dfrac{\dfrac{}{\Gamma_3 \vdash \mathsf{K} : 2\to1\to2} \quad \dfrac{}{\Gamma_3 \vdash y : 2}}{\Gamma_3 \vdash \mathsf{K}\,y : 1\to2}}{\Gamma_3 \vdash \mathsf{S}\,x\,(\mathsf{K}\,y) : 1\to3}$$

and for the right-hand side

$$\dfrac{\dfrac{\dfrac{}{\Gamma_3 \vdash \mathsf{C} : (1\to2\to3)\to2\to1\to3} \quad \dfrac{}{\Gamma_3 \vdash x : 1\to2\to3}}{\Gamma_3 \vdash \mathsf{C}\,x : 2\to1\to3} \quad \dfrac{}{\Gamma_3 \vdash y : 2}}{\Gamma_3 \vdash \mathsf{C}\,x\,y : 1\to3}$$

so the rule is typeable.

ii) We would need to type the left-hand side using *exactly* the type $\mathcal{E}\,S$, but that is not possible: the second argument is I, which creates a problem when trying to type the left-hand side as specified in Definition 6.13.

$$\dfrac{\dfrac{\dfrac{}{x{:}2\to3 \vdash \mathsf{S} : \mathcal{E}\,\mathsf{S}} \quad \dfrac{\dfrac{}{x{:}2\to3 \vdash \mathsf{K} : (2\to3)\to1\to2\to3} \quad \dfrac{}{x{:}2\to3 \vdash x : 2\to3}}{x{:}2\to3 \vdash \mathsf{K}\,x : 1\to2\to3}}{x{:}2\to3 \vdash \mathsf{S}\,(\mathsf{K}\,x) : (1\to2)\to1\to3} \quad \dfrac{}{x{:}2\to3 \vdash I : 4\to4}}{x{:}2\to3 \vdash \mathsf{S}\,(\mathsf{K}\,x)\,I : ??}$$

We can only type this if we change the environment type for S, making $\varphi_1 = \varphi_2$, thus defining

$$\mathcal{E}\,\mathsf{S} \;=\; (1\to1\to3)\to(1\to1)\to1\to3$$

iii) Then we can only well-type applications of S when the second argument has an 'identity' type, and this rules out an application like $\mathsf{S}\,x\,\mathsf{K}\,z$.

Exercise 6.28 Proof: By induction on the structure of derivations.

(Ax): Then $t = x$, and $x{:}A \in \Gamma$; then also $x{:}SA \in S\Gamma$, so by rule (Ax), $S\,\Gamma;\mathcal{E} \vdash x : S\,A$.

$(Call)$: Then $t = F$, and there exists a substitution S' such that $S'(\mathcal{E}F) = A$; notice that $S{\circ}S'$ is also a substitution, so by rule $(Call)$, $S\,\Gamma;\mathcal{E} \vdash x : S\,A$.

$(\to E)$: Then $t = t_1t_2$, and there exists B such that $\Gamma;\mathcal{E} \vdash t_1 : B\to A$ and $\Gamma;\mathcal{E} \vdash t_2 : B$. By induction, we have both $S\Gamma;\mathcal{E} \vdash t_1 : S(B\to A)$ and $S\Gamma;\mathcal{E} \vdash t_2 : S\,B$; since $S(B\to A) = S\,B\to S\,A$, by rule $(\to E)$ we get $S\Gamma;\mathcal{E} \vdash t_1t_2 : S\,A$.

Exercise 6.29 Proof: By induction on the definition of $pp\ t\ \mathcal{E}$, using Lemma 6.11.

$(pp\ x\ \mathcal{E} = \langle x{:}\varphi;\ \varphi\rangle)$: Evidently $x{:}\varphi;\mathcal{E} \vdash x : \varphi$ follows by rule (Ax).

$(pp\ F\ \mathcal{E} = \langle \phi;\ \mathit{FreshInstance}\,(\mathcal{E}\,F)\rangle)$: Since *FreshInstance* is a substitution, we have that $\phi;\mathcal{E} \vdash F : \mathit{FreshInstance}\,(\mathcal{E}\,F)$ follows by rule $(Call)$.

$(pp\ t_1t_2\ \mathcal{E} = S\,\langle \Pi_1 \cup \Pi_2;\ \varphi\rangle$, *with* φ *fresh,* $\langle \Pi_1;\ P_1\rangle = pp\ \ t_1,\ E\langle \Pi_2;\ P_2\rangle = pp\ \ t_2\ \ \mathcal{E}$, *and* $S = \mathit{unify}\ P_1\ P_2\to\varphi)$: Then by induction we have $\Pi_1;\mathcal{E} \vdash t_1 : P_1$ and $\Pi_2;\mathcal{E} \vdash t_2 : P_2$. By Lemma 6.11, also $S\Pi_1;\mathcal{E} \vdash t_1 : SP_1$ and $S\Pi_2;\mathcal{E} \vdash t_2 : SP_2$; by weakening we get that both $S(\Pi_1 \cup \Pi_2);\mathcal{E} \vdash t_1 : SP_1$ and $S(\Pi_1 \cup \Pi_2);\mathcal{E} \vdash t_2 : SP_2$. Since $S = \mathit{unify}\ P_1\ P_2\to\varphi$, in fact $SP_1 = S(P_2\to\varphi) = SP_2\to S\varphi$, so by rule $(\to E)$ we have $S(\Pi_1 \cup \Pi_2);\mathcal{E} \vdash t_1t_2 : S\varphi$.

Exercise 6.30 Take

$$\begin{array}{lcl} \mathcal{E}\ 0 & = & I \\ \mathcal{E}\ \mathsf{Succ} & = & I{\to}I \\ \mathcal{E}\ \mathsf{Fac} & = & I{\to}I \\ \mathcal{E}\ \mathsf{Times} & = & I{\to}I{\to}I \end{array}$$

i)

$$\begin{array}{lcl} \mathsf{Fac}\ \ 0 & \to & \mathsf{Succ}\ 0 \\ \mathsf{Fac}\ \ (\mathsf{Succ}\ n) & \to & \mathsf{Times}\ (\mathsf{Succ}\ n)\ (\mathsf{Fac}\ n) \end{array}$$

Then (where $I = Int$)

$$\dfrac{\dfrac{}{\emptyset \vdash \mathsf{Fac} : I{\to}I} \quad \dfrac{}{\emptyset \vdash 0 : I}}{\emptyset \vdash \mathsf{Fac}\ 0 : I} \qquad \dfrac{\dfrac{}{\emptyset \vdash \mathsf{Succ} : I{\to}I} \quad \dfrac{}{\emptyset \vdash 0 : I}}{\emptyset \vdash \mathsf{Succ}\ \ 0 : I} \qquad \dfrac{\dfrac{}{n{:}I \vdash \mathsf{Fac} : I{\to}I} \quad \dfrac{\dfrac{}{n{:}I \vdash \mathsf{Succ} : I{\to}I} \quad \dfrac{}{n{:}I \vdash n : I}}{n{:}I \vdash \mathsf{Succ}\ \ n : I}}{n{:}I \vdash \mathsf{Fac}\ (\mathsf{Succ}\ \ n) : I}$$

$$\dfrac{\dfrac{\dfrac{}{n{:}I \vdash \mathsf{Times} : I{\to}I{\to}I} \quad \dfrac{\dfrac{}{n{:}I \vdash \mathsf{Succ} : I{\to}I} \quad \dfrac{}{n{:}I \vdash n : I}}{n{:}I \vdash \mathsf{Succ}\ \ n : I}}{n{:}I \vdash \mathsf{Times}\ (\mathsf{Succ}\ \ n) : I{\to}I} \quad \dfrac{\dfrac{}{n{:}I \vdash \mathsf{Fac} : I{\to}I} \quad \dfrac{}{n{:}I \vdash n : I}}{n{:}I \vdash \mathsf{Fac}\ n : I}}{n{:}I \vdash \mathsf{Times}\ (\mathsf{Succ}\ \ n)\ (\mathsf{Fac}\ n) : I}$$

Notice that the first and third derivation are principal and use the environment type for the defined symbol.

ii)

$$\begin{array}{lcl} \mathcal{E}\ [\,] & = & [\varphi] \\ \mathcal{E}\ \mathsf{Cons} & = & \varphi{\to}[\varphi]{\to}[\varphi] \\ \mathcal{E}\ \mathsf{Facs} & = & I{\to}[I] \end{array} \qquad \begin{array}{lcl} \mathsf{Facs}\ \ n & \to & \mathsf{Cons}\ (\mathsf{Fac}\ n)\ (\mathsf{Facs}\ (\mathsf{Succ}\ \ n)) \\ \mathsf{Facs}\ \ 0 & & \end{array}$$

$$\dfrac{\dfrac{}{\emptyset \vdash \mathsf{Facs} : I{\to}[I]} \quad \dfrac{}{n{:}I \vdash n : I}}{\emptyset \vdash \mathsf{Facs}\ n : [I]}$$

and

$$\dfrac{\dfrac{\dfrac{}{n{:}I \vdash \mathsf{Cons} : I{\to}[I]{\to}[I]} \quad \dfrac{\dfrac{}{n{:}I \vdash \mathsf{Fac} : I{\to}I} \quad \dfrac{}{n{:}I \vdash n : I}}{n{:}I \vdash \mathsf{Fac}\, n : I}}{n{:}I \vdash \mathsf{Cons}\,(\mathsf{Fac}\, n) : [I]{\to}[I]} \quad \dfrac{\dfrac{}{n{:}I \vdash \mathsf{Facs} : I{\to}[I]} \quad \dfrac{\dfrac{}{n{:}I \vdash \mathsf{Succ} : I{\to}I} \quad \dfrac{}{n{:}I \vdash n : I}}{n{:}I \vdash \mathsf{Succ}\, n : I}}{n{:}I \vdash \mathsf{Facs}\,(\mathsf{Succ}\, n) : [I]}}{n{:}I \vdash \mathsf{Cons}\,(\mathsf{Fac}\, n)\,(\mathsf{Facs}\,(\mathsf{Succ}\, n)) : [I]}$$

Exercise 6.31

$$\begin{array}{lcl} \mathcal{E}\,\mathsf{append} & = & [\varphi]{\to}[\varphi]{\to}[\varphi] \\ \mathcal{E}\,\mathsf{nil} & = & [\varphi] \\ \mathcal{E}\,\mathsf{cons} & = & \varphi{\to}[\varphi]{\to}[\varphi] \\ \mathcal{E}\,\mathsf{map} & = & (\varphi_1{\to}\varphi_2){\to}[\varphi_1]{\to}[\varphi_2] \end{array}$$

Let $\Gamma = f{:}\varphi_1{\to}\varphi_2, y{:}\varphi_1, l{:}[\varphi_1]$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{map} : (1{\to}2){\to}[1]{\to}[2]} \quad \dfrac{}{\Gamma \vdash f : 1{\to}2}}{\Gamma \vdash \mathsf{map}\ f : [1]{\to}[2]} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{cons} : 1{\to}[1]{\to}[1]} \quad \dfrac{}{\Gamma \vdash y : 1}}{\Gamma \vdash \mathsf{cons}\, y : [1]{\to}[1]} \quad \dfrac{}{\Gamma \vdash l : [1]}}{\Gamma \vdash \mathsf{cons}\, y\, l : [1]}}{\Gamma \vdash \mathsf{map}\ f\ (\mathsf{cons}\, y\, l) : [2]}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{cons} : 2{\to}[2]{\to}[2]} \quad \dfrac{\dfrac{}{\Gamma \vdash f : 1{\to}2} \quad \dfrac{}{\Gamma \vdash y : 1}}{\Gamma \vdash f\ y : 2}}{\Gamma \vdash \mathsf{cons}\ (f\ y) : [2]{\to}[2]} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{map} : (1{\to}2){\to}[1]{\to}[2]} \quad \dfrac{}{\Gamma \vdash f : 1{\to}2}}{\Gamma \vdash \mathsf{map}\ f : [1]{\to}[2]} \quad \dfrac{}{\Gamma \vdash l : [1]}}{\Gamma \vdash \mathsf{map}\ f\ l : [2]}}{\Gamma \vdash \mathsf{cons}\ (f\ y)\ (\mathsf{map}\ f\ l) : [2]}$$

Exercise 6.32 $\mathsf{B} : (\varphi_1{\to}\varphi_2){\to}(\varphi_3{\to}\varphi_1){\to}\varphi_3{\to}\varphi_2$
$\mathsf{C} : (\varphi_1{\to}\varphi_2{\to}\varphi_3){\to}\varphi_2{\to}\varphi_1{\to}\varphi_3$
$\mathsf{W} : (\varphi_1{\to}\varphi_1{\to}\varphi_2){\to}\varphi_1{\to}\varphi_2$

With $\Gamma = x{:}1{\to}2,\ y{:}3{\to}1,\ z{:}3$:

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma\vdash B:(1{\to}2){\to}(3{\to}1){\to}3{\to}2}\quad \overline{\Gamma\vdash x:1{\to}2}}{\Gamma\vdash B\,x\,y:(3{\to}1){\to}3{\to}2}\quad \overline{\Gamma\vdash y:3{\to}1}}{\Gamma\vdash B\,x\,y:3{\to}2}\quad \overline{\Gamma\vdash z:3}}{\Gamma\vdash B\,x\,y\,z:2}$$

$$\dfrac{\overline{\Gamma\vdash x:1{\to}2}\quad \dfrac{\overline{\Gamma\vdash y:3{\to}1}\quad \overline{\Gamma\vdash z:3}}{\Gamma\vdash y\,z:1}}{\Gamma\vdash x\,(y\,z):2}$$

With $\Gamma = x{:}1{\to}2{\to}3,\ y{:}2,\ z{:}1$:

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma\vdash C:(1{\to}2{\to}3){\to}2{\to}1{\to}3}\quad \overline{\Gamma\vdash x:1{\to}2{\to}3}}{\Gamma\vdash C\,x\,y:2{\to}1{\to}3}\quad \overline{\Gamma\vdash y:2}}{\Gamma\vdash C\,x\,y:1{\to}3}\quad \overline{\Gamma\vdash z:1}}{\Gamma\vdash C\,x\,y\,z:3} \qquad \dfrac{\dfrac{\overline{\Gamma\vdash x:1{\to}2{\to}3}\quad \overline{\Gamma\vdash z:1}}{\Gamma\vdash x\,z:2{\to}3}\quad \overline{\Gamma\vdash y:2}}{\Gamma\vdash x\,z\,y:3}$$

With $\Gamma = x{:}1{\to}1{\to}2,\ y{:}1$:

$$\dfrac{\dfrac{\overline{\Gamma\vdash C:(1{\to}1{\to}2){\to}1{\to}2}\quad \overline{\Gamma\vdash x:1{\to}1{\to}2}}{\Gamma\vdash W\,x:1{\to}2}\quad \overline{\Gamma\vdash y:1}}{\Gamma\vdash W\,x\,y:2} \qquad \dfrac{\dfrac{\overline{\Gamma\vdash x:1{\to}1{\to}2}\quad \overline{\Gamma\vdash y:1}}{\Gamma\vdash x\,y:1{\to}2}\quad \overline{\Gamma\vdash y:1}}{\Gamma\vdash x\,y\,y:2}$$

Exercise 6.33 We check that, for all four rules, supplying left and right-hand side with an extra argument gives two reductions that lead to the same result:

$\mathsf{S}\,(\mathsf{K}\,t_1)\,(\mathsf{K}\,t_2)\,t_3 \;\to\; \mathsf{K}\,t_1\,t_3\,(\mathsf{K}\,t_2\,t_3) \;\to\; t_1\,(\mathsf{K}\,t_2\,t_3) \;\to\; t_1\,t_2$ and $\mathsf{K}\,(t_1\,t_2)\,t_3 \;\to\; t_1\,t_2$.

$\mathsf{S}\,(\mathsf{K}\,t_1)\,\mathsf{I}\,t_3 \;\to\; \mathsf{K}\,t_1\,t_3\,(\mathsf{I}\,t_3) \;\to\; t_1\,(\mathsf{I}\,t_3) \;\to\; t_1\,t_3$.

$\mathsf{S}\,(\mathsf{K}\,t_1)\,t_2\,t_3 \;\to\; \mathsf{K}\,t_1\,t_3\,(t_2\,t_3) \;\to\; t_1\,(t_2\,t_3)$ and $\mathsf{B}\,t_1\,t_2\,t_3 \;\to\; t_1\,(t_2\,t_3)$.

$\mathsf{S}\,t_1\,(\mathsf{K}\,t_2)\,t_3 \;\to\; t_1\,t_3\,(\mathsf{K}\,t_2\,t_3) \;\to\; t_1\,t_3\,t_2$ and $\mathsf{C}\,t_1\,t_2\,t_3 \;\to\; t_1\,t_3\,t_2$.

Exercise 6.34 *i*) By induction on the structure of terms.

$(t = x,\ y \neq x)$: Notice that then $\mathsf{Fun}\ y\ x = \mathsf{K}\,x$. Assume $\Gamma,y{:}A \vdash_{\mathcal{E}_{\mathrm{CL}}} x : B$, then $x{:}B \in \Gamma$, and we can construct:

$$\dfrac{\overline{\Gamma\vdash \mathsf{K}:B{\to}A{\to}B}\,(\mathsf{K})\quad \overline{\Gamma\vdash x:B}\,(Ax)}{\Gamma\vdash \mathsf{K}\,x:B}\,({\to}E)$$

$(t = y)$: Notice that then $\mathsf{Fun}\ y\ y = \mathsf{I}$. Assume $\Gamma,y{:}A \vdash_{\mathcal{E}_{\mathrm{CL}}} y : B$, then $A = B$; we can construct:

$$\overline{\Gamma\vdash \mathsf{I}:A{\to}A}\,(\mathsf{I})$$

$(t = t_1t_2)$: Notice that $\mathsf{Fun}\ t_1\ t_2 = \mathsf{S}\,(\mathsf{Fun}\ y\ t_1)\,(\mathsf{Fun}\ y\ t_2)$. The derivation for $\Gamma,y{:}A \vdash_{\mathcal{E}_{\mathrm{CL}}} t_1t_2 : B$ is shaped as follows:

$$\dfrac{\Gamma,y{:}A\vdash t_1:C{\to}B\qquad \Gamma,y{:}A\vdash t_2:C}{\Gamma,y{:}A\vdash t_1t_2:B}\,({\to}E)$$

By induction, we have (derivations for) $\Gamma \vdash_{\mathcal{E}_{\mathrm{CL}}} \mathsf{Fun}\ y\ t_1 : A{\to}C{\to}B$ and $\Gamma \vdash_{\mathcal{E}_{\mathrm{CL}}} \mathsf{Fun}\ y\ t_2 : A{\to}C$, and can construct:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{S} : (A{\to}C{\to}B){\to}(A{\to}C){\to}A{\to}B}\,(\mathsf{S}) \qquad \dfrac{\boxed{\qquad\qquad\qquad}}{\Gamma \vdash \mathsf{Fun}\ y\ t_1 : A{\to}C{\to}B}}{\Gamma \vdash \mathsf{S}\,(\mathsf{Fun}\ y\ t_1) : A{\to}C{\to}B}\,(\to E) \qquad \dfrac{\boxed{\qquad\qquad\qquad}}{\Gamma \vdash \mathsf{Fun}\ y\ t_2 : A{\to}C}}{\Gamma \vdash \mathsf{S}\,(\mathsf{Fun}\ y\ t_1)\,(\mathsf{Fun}\ y\ t_2) : A{\to}B}\,(\to E)$$

ii) By induction on the structure of terms.

$(M = x)$: Then $x{:}A \in \Gamma$; $\llbracket x \rrbracket_{\mathrm{CL}} = x$, and we can construct:

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

$(M = \lambda x.N)$: Then $A = B{\to}C$, and $\Gamma,x{:}B \vdash_{\mathrm{c}} N : C$; notice that $\llbracket M \rrbracket_{\mathrm{CL}} = \llbracket \lambda x.N \rrbracket_{\mathrm{CL}} = \mathsf{Fun}\ x\ N$. By induction, we have $\Gamma,x{:}B \vdash_{\mathcal{E}_{\mathrm{CL}}} \llbracket N \rrbracket_{\mathrm{CL}} : C$; by part (*i*) we get $\Gamma \vdash_{\mathcal{E}_{\mathrm{CL}}} \mathsf{Fun}\ x\ N : B{\to}C$, so we have $\Gamma \vdash_{\mathcal{E}_{\mathrm{CL}}} \llbracket M \rrbracket_{\mathrm{CL}} : A$.

$(M = PQ)$: Then there exists B such that $\Gamma \vdash_{\mathrm{c}} P : B{\to}A$ and $\Gamma \vdash_{\mathrm{c}} Q : B$. By induction, we have both $\Gamma \vdash_{\mathcal{E}_{\mathrm{CL}}} \llbracket P \rrbracket_{\mathrm{CL}} : B{\to}A$ and $\Gamma \vdash_{\mathcal{E}_{\mathrm{CL}}} \llbracket Q \rrbracket_{\mathrm{CL}} : B$; observing that $\llbracket PQ \rrbracket_{\mathrm{CL}} = \llbracket P \rrbracket_{\mathrm{CL}}\, \llbracket Q \rrbracket_{\mathrm{CL}}$, applying $(\to E)$ gives the desired result.

iii) This follows by straightforward induction.

Exercise 6.35 *i*)
$$\begin{array}{rcl}
\llbracket \lambda xy.xy \rrbracket_{\mathrm{CL}} & = & \mathsf{Fun}\ x\ (\mathsf{Fun}\ y\ (x\ y)) \\
& = & \mathsf{Fun}\ x\ (\mathsf{S}(\mathsf{Fun}\ y\ x)\ (\mathsf{Fun}\ y\ y)) \\
& = & \mathsf{Fun}\ x\ (\mathsf{S}\ (\mathsf{K}x)\ \mathsf{I}) \\
& = & \mathsf{S}\ (\mathsf{Fun}\ x\ (\mathsf{S}\ (\mathsf{K}x)))\ (\mathsf{Fun}\ x\ \mathsf{I}) \\
& = & \mathsf{S}\ (\mathsf{S}\ (\mathsf{Fun}\ x\ \mathsf{S})\ (\mathsf{Fun}\ x\ (\mathsf{K}x)))\ (\mathsf{KI}) \\
& = & \mathsf{S}\ (\mathsf{S}\ (\mathsf{KS})\ (\mathsf{S}\ (\mathsf{Fun}\ x\ \mathsf{K})\ (\mathsf{Fun}\ x\ x)))\ (\mathsf{KI}) \\
& = & \mathsf{S}\ (\mathsf{S}\ (\mathsf{KS})\ (\mathsf{S}\ (\mathsf{KK})\ \mathsf{I}))\ (\mathsf{KI})
\end{array}$$

ii) We can show this using the result in the notes, that states that $\langle \llbracket M \rrbracket_{\mathrm{CL}} \rangle_\lambda \to_\beta^* M$, for all M. Otherwise, let $S = \langle \mathsf{S} \rangle_\lambda, K = \langle \mathsf{K} \rangle_\lambda$, and $I = \langle \mathsf{I} \rangle_\lambda$

$$\begin{array}{rclrl}
\langle \llbracket \lambda xy.xy \rrbracket_{\mathrm{CL}} \rangle_\lambda & = & \langle \mathsf{S}(\mathsf{S}(\mathsf{KS})(\mathsf{S}(\mathsf{KK})\mathsf{I}))(\mathsf{KI}) \rangle_\lambda & = & S(S(KS)(S(KK)I))(KI) \\
& = & (\lambda xyz.xz(yz))\ (S(KS)(S(KK)I))\ (KI) & \to_\beta^* & \lambda z.S\ (KS)\ (S(KK)I)z\ (KIz) \\
& \to_\beta^* & \lambda z.K\,S\,z\ (S(KK)Iz)\ (KIz) & \to_\beta^* & \lambda z.S\ (KKz(Iz))\ I \\
& \to_\beta^* & \lambda z.S\ (Kz)\ I & \to_\beta^* & \lambda z.\lambda c.K\,z\,c\ (Ic) \\
& \to_\beta^* & \lambda z.\lambda c.zc & =_\alpha & \lambda xy.xy
\end{array}$$

iii) Let $\Gamma = x{:}A{\to}B, y{:}A$. Since $\llbracket \lambda xy.xy \rrbracket_{\mathrm{CL}} = \mathsf{S}(\mathsf{S}(\mathsf{KS})(\mathsf{S}(\mathsf{KK})\mathsf{I}))(\mathsf{KI})$,

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\mathrm{c}} x : A{\to}B} \qquad \dfrac{}{\Gamma \vdash_{\mathrm{c}} y : A}}{\Gamma \vdash_{\mathrm{c}} xy : B}}{x{:}A{\to}B \vdash_{\mathrm{c}} \lambda y.xy : A{\to}B}}{\emptyset \vdash_{\mathrm{c}} \lambda xy.xy : (A{\to}B){\to}A{\to}B}$$

and, by a result in the notes, $\Gamma \vdash_{\mathrm{c}} \lambda xy.xy : A$ implies $\Gamma \vdash_{\mathcal{E}_{\mathrm{CL}}} \llbracket \lambda xy.xy \rrbracket_{\mathrm{CL}} : A$.

Exercise 6.36 No, it is not. It contains a non-terminating sub-term

$$(\mathsf{S\,I\,I})\,(\mathsf{S\,I\,I}) \;\to\; \mathsf{I}\,(\mathsf{S\,I\,I})\,(\mathsf{I}\,(\mathsf{S\,I\,I})) \;\to\; \mathsf{S\,I\,I}\,(\mathsf{I}\,(\mathsf{S\,I\,I})) \;\to\; \mathsf{S\,I\,I}\,(\mathsf{S\,I\,I}) \;\to\; \dots$$

and only strongly normalisable terms are typeable.

Exercise 7.16 Using this derivation (twice), we can even type non-terminating terms:

$$\dfrac{\dfrac{\begin{array}{c}\boxed{\qquad\qquad}\\ \vdash \lambda x.xx : \mu X.X\to\varphi\end{array}}{\vdash \lambda x.xx : (\mu X.X\to\varphi)\to\varphi}\,(\mu) \qquad \begin{array}{c}\boxed{\qquad\qquad}\\ \vdash \lambda x.xx : \mu X.X\to\varphi\end{array}}{\vdash (\lambda x.xx)\,(\lambda x.xx) : \varphi}$$

Exercise 7.17 Take $\Gamma = x{:}\mu X.X\to A, f{:}A\to A$; notice that $\mu X.X\to A =_\mu (\mu X.X\to A)\to A$.

$$\mathcal{D} :: \quad \dfrac{\dfrac{\dfrac{}{\Gamma\vdash f : A\to A} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma\vdash x : \mu X.X\to A}}{\Gamma\vdash x : (\mu X.X\to A)\to A} \qquad \dfrac{}{\Gamma\vdash x : \mu X.X\to A}}{\Gamma\vdash xx : A}}{\Gamma\vdash f(xx) : A}}{f{:}A\to A\vdash \lambda x.f(xx) : (\mu X.X\to A)\to A}$$

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\boxed{\quad\mathcal{D}\quad}\\ f{:}A\to A\vdash \lambda x.f(xx) : (\mu X.X\to A)\to A\end{array}}{f{:}A\to A\vdash \lambda x.f(xx) : ((\mu X.X\to A)\to A)\to A} \qquad \begin{array}{c}\boxed{\quad\mathcal{D}\quad}\\ \vdash \lambda x.f(xx) : (\mu X.X\to A)\to A\end{array}}{f{:}A\to A\vdash (\lambda x.f(xx))(\lambda x.f(xx)) : A}}{\vdash \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) : (A\to A)\to A}$$

This is possible for any A.

Exercise 7.18 Take $\Gamma = a{:}\varphi,\ b{:}[\varphi]$; $\Gamma' = \Gamma,\ l{:}[\phi],\ ll{:}[\phi]\to I$; $\Gamma'_x = \Gamma',\ x{:}\phi\times[\phi]$.

$$\text{Let } \mathcal{D} = \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma'_x\vdash + : I\to I\to I} \qquad \dfrac{}{\Gamma'_x\vdash 1 : I}}{\Gamma'_x\vdash +\,1 : I\to I} \qquad \dfrac{\dfrac{}{\Gamma'_x\vdash ll : [\phi]\to I} \qquad \dfrac{\dfrac{}{\Gamma_x\vdash x : \phi\times[\phi]}}{\Gamma'_x\vdash \mathit{right}\ (x) : [\phi]}}{\Gamma'_x\vdash ll(\mathit{right}\ (x)) : I}}{\Gamma'_x\vdash +\,1\ (ll(\mathit{right}\ (x))) : I}}{\Gamma'\vdash \lambda x.+\,1\ (ll(\mathit{right}\ (x))) : (\phi\times[\phi])\to I}\ ,$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma'\vdash l : [\phi]}}{\Gamma'\vdash \mathsf{unfold}\,(l) : \mathsf{unit}+\phi\times[\phi]} \qquad \dfrac{\dfrac{}{\Gamma',x{:}\mathsf{unit}\vdash 0 : I}}{\Gamma'\vdash \lambda x.0 : \mathsf{unit}\to I} \qquad \begin{array}{c}\boxed{\quad\mathcal{D}\quad}\\ \Gamma'\vdash \lambda x.+\,1\ (ll(\mathit{right}\ (x))) : (\phi\times[\phi])\to I\\ \vdots\end{array}}{\Gamma'\vdash \mathsf{case}\ (\mathsf{unfold}\,(l),\lambda x.0,\lambda x.+\,1\ (ll(\mathit{right}\ (x)))) : I}}{\Gamma'\setminus l\vdash \lambda l.\mathsf{case}\ (\mathsf{unfold}\,(l),\lambda x.0,\lambda x.+\,1\ (ll(\mathit{right}\ (x)))) : [\phi]\to I}}{\Gamma\vdash \mathsf{fix}\,ll.\lambda l.\mathsf{case}\ (\mathsf{unfold}\,(l),\lambda x.0,\lambda x.+\,1\ (ll(\mathit{right}\ (x)))) : [\phi]\to I} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\vdash a : \varphi} \qquad \dfrac{}{\Gamma\vdash b : [\varphi]}}{\Gamma\vdash \langle a,b\rangle : B\times[B]}}{\Gamma\vdash \mathsf{inj{\cdot}r}\,\langle a,b\rangle : \mathsf{unit}+(B\times[B])}}{\Gamma\vdash \mathsf{fold}\,(\mathsf{inj{\cdot}r}\,\langle a,b\rangle) : [B]}}{\Gamma\vdash (\mathsf{fix}\,ll.\lambda l.\mathsf{case}\ (\mathsf{unfold}\,(l),\lambda x.0,\lambda x.+\,1\ (ll(\mathit{right}\ (x)))))\,(\mathsf{fold}\,(\mathsf{inj{\cdot}r}\,\langle a,b\rangle)) : I}$$

Exercise 7.19 Take $B = \mu X.X\to(\varphi\to\varphi)\to\varphi$ (then $B =_\mu B\to(\varphi\to\varphi)\to\varphi$), $\Gamma = x{:}B, y{:}\varphi\to\varphi$.

$$\mathcal{D} :: \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\vdash y : \varphi\to\varphi}\,(Ax) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\vdash x : B}\,(Ax)}{\Gamma\vdash x : B\to(\varphi\to\varphi)\to\varphi}\,(\mu) \qquad \dfrac{}{\Gamma\vdash x : B}\,(Ax)}{\Gamma\vdash xx : (\varphi\to\varphi)\to\varphi}\,(\to E) \qquad \dfrac{}{\Gamma\vdash y : \varphi\to\varphi}\,(Ax)}{\Gamma\vdash xxy : \varphi}\,(\to E)}{\Gamma\vdash y(xxy) : \varphi}\,(\to E)}{x{:}\mu X.X\to\varphi\vdash \lambda y.y(xxy) : (\varphi\to\varphi)\to\varphi}\,(\to I)}{\vdash \lambda xy.y(xxy) : B\to(\varphi\to\varphi)\to\varphi}\,(\to I)$$

$$\dfrac{\begin{array}{c}\boxed{\quad\mathcal{D}\quad}\\ \vdash \lambda xy.y(xxy) : B{\to}(\varphi{\to}\varphi){\to}\varphi\end{array} \qquad \dfrac{\begin{array}{c}\boxed{\quad\mathcal{D}\quad}\\ \vdash \lambda xy.y(xxy) : B{\to}(\varphi{\to}\varphi){\to}\varphi\end{array}}{\vdash \lambda xy.y(xxy) : B}\,(\mu)}{\vdash (\lambda xy.y(xxy))(\lambda xy.y(xxy)) : (\varphi{\to}\varphi){\to}\varphi}\,({\to}E)$$

Take $T = (\lambda xy.y((\mathsf{unfold}\,x)xy))(\mathsf{fold}\,(\lambda xy.y((\mathsf{unfold}\,x)xy)))$;

$$\begin{array}{ll}
(\lambda xy.y((\mathsf{unfold}\,x)xy))(\mathsf{fold}\,(\lambda xy.y((\mathsf{unfold}\,x)xy)))\,M & \to\\
(\lambda y.y((\mathsf{unfold}\,(\mathsf{fold}\,(\lambda xy.y((\mathsf{unfold}\,x)xy))))(\mathsf{fold}\,(\lambda xy.y((\mathsf{unfold}\,x)xy)))y))\,M & \to\\
M\,((\mathsf{unfold}\,(\mathsf{fold}\,(\lambda xy.y((\mathsf{unfold}\,x)xy))))(\mathsf{fold}\,(\lambda xy.y((\mathsf{unfold}\,x)xy)))\,M) & \to\\
M\,((\lambda xy.y((\mathsf{unfold}\,x)xy))(\mathsf{fold}\,(\lambda xy.y((\mathsf{unfold}\,x)xy)))\,M) &
\end{array}$$

so $TM \to M(TM)$

Exercise 8.32 *i*) Take $\Gamma = x{:}A{\to}B, y{:}C\cap A$.

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash_\cap x : A{\to}B} \qquad \overline{\Gamma \vdash_\cap y : A}}{\Gamma \vdash_\cap xy : B}}{\Gamma\backslash y \vdash_\cap \lambda y.xy : (C\cap A){\to}B}}{\emptyset \vdash_\cap \lambda xy.xy : (A{\to}B){\to}(C\cap A){\to}B}$$

ii) Take $\Gamma = x{:}A{\to}B{\to}C, y{:}D{\to}B, z{:}A\cap D$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash_\cap x : A{\to}B{\to}C} \quad \overline{\Gamma \vdash_\cap z : A}}{\Gamma \vdash_\cap xz : B{\to}C} \qquad \dfrac{\overline{\Gamma \vdash_\cap y : D{\to}B} \quad \overline{\Gamma \vdash_\cap z : D}}{\Gamma \vdash_\cap yz : B}}{\Gamma \vdash_\cap xz(yz) : C}}{\Gamma\backslash z \vdash_\cap \lambda z.xz(yz) : (A\cap D){\to}C}}{\Gamma\backslash y{,}z \vdash_\cap \lambda yz.xz(yz) : (D{\to}B){\to}(A\cap D){\to}C}}{\emptyset \vdash_\cap \lambda xyz.xz(yz) : (A{\to}B{\to}C){\to}(D{\to}B){\to}(A\cap D){\to}C}$$

iii) Take $\Gamma = x{:}A{\to}\top{\to}C, y{:}\top, z{:}A$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash_\cap x : A{\to}\top{\to}C} \quad \overline{\Gamma \vdash_\cap z : A}}{\Gamma \vdash_\cap xz : \top{\to}C} \qquad \overline{\Gamma \vdash_\cap yz : \top}}{\Gamma \vdash_\cap xz(yz) : C}}{\Gamma\backslash z \vdash_\cap \lambda z.xz(yz) : A{\to}C}}{\Gamma\backslash y{,}z \vdash_\cap \lambda yz.xz(yz) : \top{\to}A{\to}C}}{\emptyset \vdash_\cap \lambda xyz.xz(yz) : (A{\to}\top{\to}C){\to}\top{\to}A{\to}C}$$

iv) From the item above, we also have (taking $C = A$),

$$\emptyset \vdash_\cap \lambda xyz.xz(yz) : (A{\to}\top{\to}A){\to}\top{\to}A{\to}A$$

Since

$$\dfrac{\dfrac{\overline{a{:}A, b{:}\top \vdash_\cap a : A}}{a{:}A \vdash_\cap \lambda b.a : \top{\to}A}}{\emptyset \vdash_\cap \lambda ab.a : A{\to}\top{\to}A}$$

by $({\to}E)$ we get the desired result.

Exercise 8.33 For the intersection system they are the same, since the terms are β-equal:

$$\begin{array}{rcl}
(\lambda xyz.xz(yz))(\lambda xy.x)(\lambda x.x) & \to_\beta^* & \lambda x.x \text{ and}\\
(\lambda xy.xy)(\lambda x.x) & \to_\beta^* & \lambda x.x
\end{array}$$

The principal type in Curry's system for $(\lambda xyz.xz(yz))(\lambda xy.x)$ is $(\varphi_1{\to}\varphi_2){\to}\varphi_1{\to}\varphi_1$, that for $\lambda x.x$ is $\varphi_3{\to}\varphi_3$. Calculating the principal type for $(\lambda xyz.xz(yz))(\lambda xy.x)(\lambda x.x)$ will involve unifying $(\varphi_1{\to}\varphi_2){\to}\varphi_1{\to}\varphi_1$ with $(\varphi_3{\to}\varphi_3){\to}\varphi_4$. This will succeed and return $(\varphi_4 \mapsto \varphi_3{\to}\varphi_3)\,\varphi_4 = \varphi_3{\to}\varphi_3$, so $(\lambda xyz.xz(yz))(\lambda xy.x)(\lambda x.x)$ and $\lambda x.x$ have the same principal types, so the set of assignable types is the same.

Exercise 8.34 The derivation for the principal type is on the left; the second part of the question is shown by the derivation on the right.

$$\dfrac{\dfrac{\dfrac{}{x{:}(1{\to}2)\cap 1 \vdash x : 1{\to}2} \quad \dfrac{}{x{:}(1{\to}2)\cap 1 \vdash x : 1}}{x{:}(1{\to}2)\cap 1 \vdash x\,x : 2}}{\emptyset \vdash \lambda x.xx : (1{\to}2)\cap 1{\to}2} \qquad\qquad \dfrac{\dfrac{\dfrac{}{x{:}\top{\to}\varphi \vdash x : \top{\to}\varphi} \quad \dfrac{}{x{:}\top{\to}\varphi \vdash x : \top}}{x{:}\top{\to}\varphi \vdash x\,x : \varphi}}{\emptyset \vdash \lambda x.xx : (\top{\to}\varphi){\to}\varphi}$$

Exercise 8.35 $\Gamma' = y{:}(1{\to}2)\cap 1$, $\Gamma = \Gamma', x{:}(1{\to}2)\cap 1$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}(1{\to}2)\cap 1 \vdash x : 1{\to}2} \quad \dfrac{}{x{:}(1{\to}2)\cap 1 \vdash x : 1}}{\Gamma \vdash x\,x : 2}}{\Gamma' \vdash \lambda x.xx : (1{\to}2)\cap 1{\to}2} \quad \dfrac{\dfrac{}{\Gamma' \vdash y : 1{\to}2} \quad \dfrac{}{\Gamma' \vdash y : 1}}{\Gamma' \vdash y : (1{\to}2)\cap 1}}{\Gamma' \vdash (\lambda x.xx)\,y : 2}}{\Gamma' \vdash \lambda y.(\lambda x.xx)\,y : (1{\to}2)\cap 1{\to}2}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}\top{\to}\varphi, y{:}\top{\to}\varphi \vdash x : \top{\to}\varphi} \quad \dfrac{}{x{:}\top{\to}\varphi, y{:}\top{\to}\varphi \vdash x : \top}}{x{:}\top{\to}\varphi, y{:}\top{\to}\varphi \vdash x\,x : \varphi}}{y{:}\top{\to}\varphi \vdash \lambda x.xx : (\top{\to}\varphi){\to}\varphi} \quad \dfrac{}{y{:}\top{\to}\varphi \vdash y : \top{\to}\varphi}}{y{:}\top{\to}\varphi \vdash (\lambda x.xx)\,y : \varphi}}{\emptyset \vdash \lambda y.(\lambda x.xx)\,y : (\top{\to}\varphi){\to}\varphi}$$

Exercise 8.36 No. Since the set of approximants of $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ is infinite, the set of principal types is as well.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f{:}\top{\to}\varphi \vdash f : \top{\to}\varphi} \quad \dfrac{}{f{:}\top{\to}\varphi \vdash xx : \top}}{f{:}\top{\to}\varphi \vdash f(xx) : \varphi}}{f{:}\top{\to}\varphi \vdash \lambda x.f(xx) : \top{\to}\varphi} \quad \dfrac{}{f{:}\top{\to}\varphi \vdash \lambda x.f(xx) : \top}}{f{:}\top{\to}\varphi \vdash (\lambda x.f(xx))\,(\lambda x.f(xx)) : \varphi}}{\emptyset \vdash \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) : (\top{\to}\varphi){\to}\varphi}$$