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

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

Contents

1 Lambda Calculus 3
  1.1 \( \lambda \)-terms ........................................... 3
  1.2 \( \beta \)-conversion ........................................ 5
  1.3 Making substitution explicit ............................... 8
  1.4 Example: a numeral system ................................ 9

2 The Curry type assignment system 11
  2.1 Curry type assignment ..................................... 11
  2.2 Subject Reduction ......................................... 12
  2.3 The principal type property ................................ 14

3 Combinatory Logic 20
  3.1 The relation between CL and the Lambda Calculus .......... 20
  3.2 Extending CL ........................................... 22
  3.3 Type Assignment for CL ................................... 22
  3.4 Exercises .............................................. 23

4 Dealing with polymorphism 23
  4.1 The language \( \Lambda^N \) .................................. 24
  4.2 Type assignment for \( \Lambda^N \) ........................... 25

5 Dealing with recursion 27
  5.1 The language \( \Lambda^{NR} \) .................................. 27
  5.2 Expressing recursion in the Lambda Calculus .............. 27
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 error free, it is safe to run: “Typed programs cannot go wrong” (Milner [44]). 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 quick look at intersection types.

1 Lambda Calculus

The Lambda Calculus [16, 11] 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 [54], Haskell [33], and CaML [28]. It gets its name from the Greek character λ (lambda). Church defined the Lambda 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 $V = \{x, y, z, x_1, x_2, x_3, \ldots\}$ and two operations, application and abstraction. It is formally defined by:

Definition 1.1 (λ-terms) $\Lambda$, the set of λ-terms is defined by the following grammar:
\[
M, N ::= \text{\(x\)} \mid (\lambda x. M) \mid (M \cdot N)
\]

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 Lambda Calculus, we write \(\lambda x.x^2\) rather than \(sqx = x^2\), and \((\lambda x.x^2)3\) rather than \(sq 3\), so function definition and application are not separated.

Since ‘.’ is the only operation between terms, it is normally omitted, so we write \((MN)\) rather than \((M \cdot N)\). Also, leftmost, outermost brackets are omitted, so \(MN(PQ)\) stands for \(((MN) \cdot (P \cdot Q))\). The omitted brackets are sometimes re-introduced to avoid confusion.

Also, to avoid writing many abstractions, repeated abstractions are abbreviated, so \(\lambda xyz.M\) stands for \((\lambda x. (\lambda y. (\lambda z. M)))\).

The notion of free and bound term-variables of \(\lambda\)-terms will turn out to be important and is defined as follows.

**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:

\[
\begin{align*}
fv(x) &= \{x\} & bv(x) &= \emptyset \\
fv(MN) &= fv(M) \cup fv(N) & bv(MN) &= bv(M) \cup bv(N) \\
fv(\lambda y. M) &= fv(M) \setminus \{y\} & bv(\lambda y. M) &= bv(M) \cup \{y\}
\end{align*}
\]

We write \(x \not\in M\) for \(x \not\in fv(M) \cup bv(M)\).

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. This feature returns in the Lambda Calculus and is at the basis of the computational step; its correct definition is hindered by the binding of variables.

On \(\Lambda\) the replacement of a variable \(x\) by a term \(N\), denoted by \(\langle N/x \rangle\), could be defined by:

\[
\begin{align*}
\langle x/N/x \rangle &= N \\
\langle y/N/x \rangle &= y, & \text{if } y \neq x \\
\langle (PQ)/N/x \rangle &= P\langle N/x \rangle Q\langle N/x \rangle \\
\langle (\lambda y. M)/N/x \rangle &= \lambda y. M, & \text{if } y = x \\
\langle (\lambda y. M)/N/x \rangle &= \lambda y. (M\langle N/x \rangle), & \text{if } y \neq x
\end{align*}
\]

However, notice that, in the last alternative, problems can arise, since \(N\) can obtain free occurrences of \(y\) that would become bound during substitution; this is called a variable capture.

To avoid this, the notion of substitution needs to be defined like:

\[
\begin{align*}
\langle x/N/x \rangle &= N \\
\langle y/N/x \rangle &= y, & \text{if } y \neq x \\
\langle (PQ)/N/x \rangle &= P\langle N/x \rangle Q\langle N/x \rangle \\
\langle (\lambda y. M)/N/x \rangle &= \lambda y. (M\langle N/x \rangle), & \text{if } y \not\in fv(N) \& y \neq x \\
\langle (\lambda y. M)/N/x \rangle &= \lambda z. (M\langle z/y \rangle)\langle N/x \rangle, & \text{if } y \in fv(N) \& y \neq x, z \text{ new} \\
\langle (\lambda y. M)/N/x \rangle &= \lambda y. M, & \text{if } y = x
\end{align*}
\]

a rather complicated definition, and the one used in implementation (rather CPU-heavy because of the additional ‘renaming’ \(\langle z/y \rangle\)).
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 as in the fifth alternative above. It corresponds to the mathematical idea that the functions \( fx = x^2 \) and \( gy = y^2 \) are identical. Essentially, this relation, called ‘\( \alpha \)-conversion’, is defined from:

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

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

**Definition 1.3 (Term substitution)** The substitution of the term variable \( x \) by the term \( N \) is defined inductively over the structure of terms by:

\[
\begin{align*}
x[N/x] &= N \\
y[N/x] &= y, \quad \text{if } y \neq x \\
(PQ)[N/x] &= P[N/x]Q[N/x] \\
(\lambda x.M)[N/x] &= \lambda x.(M[N/x])
\end{align*}
\]

Since \( y \) is bound in \( \lambda y.M \), it is not free in \( N \), so variable capture is impossible.

### 1.2 \( \beta \)-conversion

On \( \Lambda \), 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 (\( \beta \)-conversion)** i) The binary one-step reduction relation ‘\( \rightarrow_\beta \)’ on \( \lambda \)-terms is defined by the \( \beta \)-reduction rule (we will write \( M \rightarrow_\beta N \) rather than \( \langle M, N \rangle \in \rightarrow_\beta \)):

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

and the ‘contextual closure’ rules

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

ii) 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.

iii) The relation ‘\( \rightarrow_\beta \)’ (or ‘\( \rightarrow_{\beta^+} \)’) is defined as the reflexive, transitive closure of ‘\( \rightarrow_\beta \)’:

\[
M \rightarrow_\beta N \Rightarrow M \rightarrow_{\beta^+} N
\]

\[
M \rightarrow_{\beta^+} M
\]

\[
M \rightarrow_{\beta^+} N \land N \rightarrow_{\beta^+} P \Rightarrow M \rightarrow_{\beta^+} P
\]

iv) ‘\( =_\beta \)’ is the equivalence relation generated by ‘\( \rightarrow_{\beta^+} \)’:  

5
\[
M \rightarrow^* N \Rightarrow M =_\beta N \\
M =_\beta N \Rightarrow N =_\beta M \\
M =_\beta N \& N =_\beta P \Rightarrow M =_\beta P
\]

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 \( f 3 \) is the same as \( (\lambda x . x^2) 3 \), which reduces by the rule above to \( 3^2 = 9 \).

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

\[
(\lambda xy . xy) (\lambda xy . xy) \rightarrow (\lambda y . xy) \left[ (\lambda xy . xy) / x \right]
\]

\[
= \lambda y . (\lambda xy . xy) y
\]

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

\[
\lambda y . (\lambda xy . xy) y \rightarrow \lambda y . (\lambda y . xy) [y / x]
\]

\[
= \lambda y . (\lambda y . y y)
\]

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

\[
(\lambda xy . xy) (\lambda xy . xy) \rightarrow (\lambda y . xy) \left[ (\lambda xy . xy) / x \right]
\]

\[
= \lambda y . (\lambda z . z y) y
\]

\[
\rightarrow \lambda y . (\lambda z . z y) [y / x]
\]

\[
= \lambda y . (\lambda z . y z)
\]

\[
= \lambda y z . y z
\]

a reduction that preserves the convention.

The following terms will reappear frequently in this course:

\[
I = \lambda x . x
\]

\[
K = \lambda xy . x
\]

\[
S = \lambda xyz . xz(yz)
\]

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

**Definition 1.5** (\( \eta \)-reduction) Let \( x \notin \text{fv}(M) \), then \( \lambda x . Mx \rightarrow^\eta M \).

As mentioned above, we can see ‘\( \rightarrow^\beta \)’ as the ‘one step’ execution, and ‘\( \rightarrow^\ast \beta \)’ as a ‘many step’ execution. We can then view the relation ‘\( \rightarrow^\beta \)’ as ‘executing the same function’.

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

**Property 1.6** If \( M \rightarrow^\beta N \) and \( M \rightarrow^\beta P \), then there exists a term \( Q \) such that \( N \rightarrow^\beta Q \) and \( P \rightarrow^\beta Q \).

So diverging computations can always be joined. Or, in a diagram:
A popular variant of \( \beta \)-reduction that stands, for example, at the basis of reduction in Haskell [33], is that of lazy reduction (often called call-by-need), where a computation is delayed until the result is required. It can be defined by:

**Definition 1.7** We define *lazy reduction* \( \rightarrow_L \) on terms in \( \Lambda \) as a restriction of \( \rightarrow_\beta \) by:

\[
(\lambda x.M)N \rightarrow_L M[N/x] \\
M \rightarrow_L N \Rightarrow ML \rightarrow_L NL
\]

Notice that the two other contextual rules are omitted, so in lazy 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.

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

**Example 1.8 i)** Take \( (\lambda x.xx) (\lambda x.xx) \). This term reduces as follows:

\[
(\lambda x.xx) (\lambda x.xx) \rightarrow_\beta (xx)[(\lambda x.xx)/x] \\
= (\lambda x.xx) (\lambda x.xx)
\]

so this term reduces only to itself.

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

\[
\lambda f.(\lambda x.f(xx)) (\lambda x.f(xx)) \rightarrow_\beta \lambda f.(f(xx))[(\lambda x.f(xx))/x] \\
= \lambda f.f((\lambda x.f(xx))(\lambda x.f(xx))) \\
\rightarrow_\beta \lambda f.f(f((\lambda x.f(xx))(\lambda x.f(xx)))) \\
\rightarrow_\beta \lambda f.f(f(f((\lambda x.f(xx))(\lambda x.f(xx))))) \\
\vdots \\
\rightarrow_\beta \lambda f.f(f(f(f(f(f(\ldots)))))))
\]

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 5.

Of course, it is also possible to give \( \lambda \)-terms for which \( \beta \)-reduction is terminating.

**Definition 1.9** i) A term is *in normal form* if it does not contain a redex. Terms in normal form are defined by:

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

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

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

iii) A term \( M \) is *head-normalisable* if 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
v) A term $M$ is strongly normalisable if all reduction sequences starting from $M$ are finite.

**Example 1.10** 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 $\lambda f. ((\lambda x. f(xx)) \ (\lambda x. f(xx)))$ 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 $\lambda x y z. xz(yz) \ (\lambda ab. a)$ is a redex, so not in normal form. It has only one reduct, $(\lambda y z.(\lambda ab. a)z(yz))$, which has only one redex $(\lambda ab. a)z$. Contracting this redex gives $\lambda y z.(\lambda b. z)(yz)$, which again has only one redex, which reduces to $\lambda y z.z$. 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 Lambda Calculus. A consequence of this result is that we have a ‘halting problem’ also in the Lambda 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 Making substitution explicit

In implementations (and semantics) of the $\lambda$-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 comes with a cost; many approaches to implement substitution efficiently exist, varying from string reduction, $\lambda$-graphs, and Krivine’s machine [41].

Normally, a calculus of explicit substitutions [13, 1, 43, 42], 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 $\lambda x$, the calculus of explicit substitution with explicit names, defined by Bloo and Rose [13]. $\lambda x$ 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 1.1** (Explicit $\lambda$-calculus $\lambda x$ cf. [13]) i) The syntax of the explicit $\lambda$-calculus $\lambda x$ is defined by:

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

The notion of bound variable 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 $\rightarrow_x$ on terms in $\lambda x$ is defined by the following rules:
\[(\beta) : \quad (\lambda x . M)N \rightarrow M (x := N)\]

\[(\text{Abs}) : \quad (\lambda y . M) \langle x := L \rangle \rightarrow \lambda y . (M \langle x := L \rangle)\]

\[(\text{App}) : \quad (MN) \langle x := L \rangle \rightarrow (M \langle x := L \rangle) \langle N \langle x := L \rangle \rangle\]

\[(\text{Var}) : \quad x \langle x := L \rangle \rightarrow L \quad \text{if} \quad x \notin \text{fv}(M)\]

\[(\text{GC}) : \quad M \langle x := L \rangle \rightarrow M \quad \text{if} \quad x \notin \text{fv}(M)\]

M \rightarrow N \Rightarrow \begin{cases} 
ML \rightarrow NL \\
LM \rightarrow LN \\
\lambda x . M \rightarrow \lambda x . N \\
M \langle x := L \rangle \rightarrow N \langle x := L \rangle \\
L \langle x := M \rangle \rightarrow L \langle x := N \rangle 
\end{cases}

Notice that we allow reduction inside the substitution.

iii) We write \(\rightarrow_{\beta}\) if the rule \((\beta)\) 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 \(\rightarrow_{x}\) is confluent. It is easy to show that \(\rightarrow_{\beta}\) is (on its own) a terminating reduction, i.e. there are no infinite reduction sequences in \(\rightarrow_{\beta}\). Thereby, if no reduction starting from \(M\) terminates, then every reduction sequence in \(\rightarrow_{x}\) starting from \(M\) has an infinite number of \((b\text{-})\text{steps.}\)

The rules express reduction as a term rewriting system [38] (see Section 7). Explicit substitution describes explicitly the process of executing a \(\beta\)-reduction, i.e. expresses syntactically the details of the computation as a succession of atomic, constant-time steps, where the implicit substitution of the \(\beta\)-reduction step is split into several steps. Therefore, the following is easy to show:

**Proposition 1.2 (\(\lambda x\) implements \(\beta\)-reduction)**

i) \(M \rightarrow_{\beta} N \Rightarrow M \rightarrow_{x}^{*} N\).

ii) \(M \in \Lambda \wedge M \rightarrow_{x}^{*} N \Rightarrow \exists L \in \Lambda \left[ N \rightarrow_{x}^{*} L \wedge M \rightarrow_{x}^{*} L \right]\).

### 1.4 Example: a numeral system

We will now show the expressivity of the Lambda Calculus by encoding numbers and some basic operations on them as \(\lambda\)-terms; because of Church’s Thesis, all computable functions are, in fact, encodable, but we will not go there.

**Definition 1.11** We define the **booleans** True and False as:

\[
\text{True} = \lambda xy.x \\
\text{False} = \lambda xy.y
\]

Then the **conditional** is defined as:

\[\text{cond} = \lambda btf.btf\]

In fact, we can see cond as syntactic sugar, since also true \(MN \rightarrow M\) and false \(MN \rightarrow N\).

**Definition 1.12** The operation of **pairing** is defined via

\[\langle M, N \rangle = \lambda z.zMN\]

(or \(\text{pair} = \lambda xyz.zxy\)) and the first and second projection functions are defined by

\[\text{First} = \lambda p. p \text{True} \]
\[\text{Second} = \lambda p. p \text{False}\]
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 1.13** The (Scott) Numerals are defined by:

\[
\begin{align*}
\llbracket 0 \rrbracket &= K \\
\text{Succ} &= \lambda nxy.yn \\
\text{Pred} &= \lambda p.pKI
\end{align*}
\]

so, for example, \( \llbracket 3 \rrbracket = \llbracket S(S(S(0))) \rrbracket \)

\[
= \text{Succ} (\text{Succ} (\text{Succ} \llbracket 0 \rrbracket)) = (\lambda nxy.yn)((\lambda nxy.yn)((\lambda nxy.yn)(K))) \rightarrow_\beta \lambda xy.\lambda y.(\lambda y.\lambda y.K).
\]

Notice that \( \llbracket 0 \rrbracket = \text{True} \). It is now easy to check that

\[
\text{cond } \llbracket n \rrbracket f g \rightarrow_\beta \begin{cases} 
  f & \text{if } \llbracket n \rrbracket = 0 \\
  g (\llbracket n - 1 \rrbracket) & \text{otherwise}
\end{cases}
\]

which implies that, in this system, we can define the test \( \text{IsZero} \) as identity, \( \lambda x.x \).

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

**Definition 1.14** Addition and multiplication are now defined by:

\[
\begin{align*}
\text{Add} &= \lambda xy.\text{cond } (\text{IsZero } x) y (\text{Succ } \text{Add } (\text{Pred } x) y) \\
\text{Mult} &= \lambda xy.\text{cond } (\text{IsZero } x) \text{Zero } (\text{Add } \text{Mult } (\text{Pred } x) y) y
\end{align*}
\]

Notice that these definitions are recursive; we will see in Section 5 that we can express recursion in the Lambda Calculus.

There are alternative ways of encoding numbers in the Lambda Calculus, like the Church Numerals \( n = \lambda fn.f^n x \) which would give an elegant encoding of addition and multiplication, but has a very complicated definition of predecessor.

**Exercises**

**Exercise 1.15** Write the following terms in the original, full notation:

i) \( \lambda xyz.\lambda z.\lambda w.z \).

ii) \( \lambda xz.(\lambda yz). \).

iii) \( (\lambda x. (\lambda yz).) (\lambda w.az) \).

Remove the omissible brackets in the following terms:

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

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

vi) \( (\lambda x.\lambda y.x)(\lambda z.az) \).

* Exercise 1.16 Show that, for all M, fo (M) \( \cap \) bv(M) = \( \varnothing \).

* Exercise 1.17 Show that \( M [N/x] [P/y] = M [P/y] [N [P/y] / x] \).

**Exercise 1.18** Show that, when \( M =_{\beta} N \), then there are terms \( M, N, \ldots, M_n, M_{n+1} \) such that \( M \equiv M, N \equiv M_{n+1}, \) 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 1.19** 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 1.20** Show that, for all terms \(M\),
\[ M((\lambda xy.y(xxy)) (\lambda xy.y(xxy)))M =_\beta ((\lambda xy.y(xxy)) (\lambda xy.y(xxy)))M \]

**Exercise 1.21** Show that all terms in normal form are in head-normal form.

**Exercise 1.22** Show that, if \(M\) has a normal form, it is unique (hint: use the Church-Rosser property).

# 2 The Curry type assignment system

In this section, we will present the basic notion of type assignment for the Lambda Calculus, as first studied by H.B. Curry in [23] (see also [24]). 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 \(\Gamma \vdash C: A\) that appears in the bottom line), which states that given the assumptions in \(\Gamma\), \(A\) is the type for the term \(M\).

Type assignment assigns types to \(\lambda\)-terms, where types are defined as follows:

**Definition 2.1** i) \(T_c\), the set of types, ranged over by \(A, B, \ldots\), is defined over a set of type variables \(\Phi\), ranged over by \(\varphi\), by:
\[ A, B ::= \varphi | (A \rightarrow B) \]

ii) A statement is an expression of the form \(M : A\), where \(M \in \Lambda\) and \(A \in T_c\). \(M\) is called the subject and \(A\) the predicate of \(M : A\).

iii) A context \(\Gamma\) is a set of statements with only distinct variables as subjects; we use \(\Gamma, x : A\) for the context defined as \(\Gamma \cup \{x : A\}\) where either \(x : A \in \Gamma\) or \(x\) does not occur in \(\Gamma\), and \(x : A\) for \(\varnothing, x : A\). 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 \((\varphi_1 \rightarrow \varphi_2) \rightarrow \varphi_3 \rightarrow \varphi_4\) stands for \(((\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_3 \rightarrow \varphi_4))\).

We will now give the definition of Curry type assignment.
Definition 2.2 (cf. [23, 24])

i) Curry type assignment and derivations are defined by the following derivation rules that define a natural deduction system.

\[
\begin{align*}
(Ax) : \quad & \Gamma, x:A \vdash c : A \\
(\rightarrow I) : \quad & \Gamma, x:A \vdash c : B \\
(\rightarrow E) : \quad & \Gamma, \Delta \vdash \lambda x.M : A \rightarrow B \\
\end{align*}
\]

ii) We will write \( \Gamma \vdash c : M : A \) 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.

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

\[
\Gamma, x:A \vdash B \vdash c : x : A \rightarrow B \\
\Gamma, x:A \vdash \lambda x.c : A \\
\Gamma, x:A \vdash c : A \\
\Gamma, x:A \vdash ? : B
\]

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 \( \text{ppC} \) that tries to construct a type for terms as defined below (see Definition 2.14) will fail on \( xx \).

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 \( \text{Integer} \), then we can safely put it in a context that demands an \( \text{Integer} \), as in \( 1 + M \), because running \( M \) will return an \( \text{Integer} \). 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 If \( \Gamma \vdash c : M : A \) and \( M \rightarrow_{\beta} N \), then \( \Gamma \vdash c : N : A \).

Notice that this result states that if a derivation exists for the first result, a derivation will 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 \( \Gamma \vdash c : (\lambda x.M)N : A \); since this is derived by \( (\rightarrow E) \), there exists \( B \) such that \( \Gamma \vdash \lambda x.M : B \rightarrow A \) and \( \Gamma \vdash c : N : B \). Then \( (\rightarrow I) \) has to be the last step performed for the first result, and there are sub-derivations for \( \Gamma, x:B \vdash c : M : A \) and \( \Gamma \vdash c : N : B \), so the full derivation looks like the left-hand derivation below. Then a derivation for \( \Gamma \vdash c : M[N/x] : A \) can be obtained by replacing in the derivation for \( \Gamma, x:B \vdash c : M : A \), the sub-derivation \( \Gamma, x:B \vdash c : x : B \) (consisting of just rule \( Ax \)) by the derivation for \( \Gamma \vdash c : N : B \), as in the right-hand derivation.

\[
\begin{align*}
\Gamma, x:B \vdash x : B \\
\Gamma \vdash \lambda x.M : B \rightarrow A \\
\Gamma \vdash c : N : B
\end{align*}
\]

Notice that we then also need to systematically replace \( x \) by \( N \) throughout \( D_1 \).
We will need the following result below:

**Lemma 2.6** 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 $\{ x : B \mid x : B \in \Gamma \land x \notin \text{fo}(M) \} \vdash_c M : A$.

iii) If $\Gamma \vdash_c M : A$ and $x \in \text{fo}(M)$, then there exists $B$ such that $x : B \in \Gamma$.

We will leave the proof of this result as an exercise.

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

**Lemma 2.7** $\exists C [\Gamma, x : C \vdash_c M : A \& \Gamma \vdash_c N : C] \Rightarrow \Gamma \vdash_c M[N/x] : A$.

**Proof:** By induction on the structure of terms.

\[(M \equiv x) : \exists C [\Gamma, x : C \vdash_c x : A \& \Gamma \vdash_c N : C] \Rightarrow (Ax) \]
\[\Gamma, x : A \vdash_c x : A \& \Gamma \vdash_c N : A \Rightarrow \Gamma \vdash_c x [N/x] : A.\]

\[(M \equiv y \neq x) : \exists C [\Gamma, x : C \vdash_c y : A \& \Gamma \vdash_c N : C] \Rightarrow (2.6(ii)) \Gamma \vdash_c y : A.\]

\[(M \equiv \lambda y.M') \exists C [\Gamma, x : C \vdash_c \lambda y. M' : A \& \Gamma \vdash_c N : C] \Rightarrow (\rightarrow I)\]
\[\exists C, A', B' [\Gamma, x : C, y : A' \vdash_c M' : B' \& A = A' \rightarrow B' \& \Gamma \vdash_c N : C] \Rightarrow (IH)\]
\[\exists C, A', B' [\Gamma, x : C, y : A' \vdash_c M' : [N/x] : B' \& A = A' \rightarrow B'] \Rightarrow (\rightarrow I)\]
\[\Gamma \vdash_c \lambda y.M'[N/x] : A.\]
\[\Gamma \vdash_c (\lambda y.M')[N/x] : A.\]

\[(M \equiv PQ) : \exists C [\Gamma, x : C \vdash_c PQ : A \& \Gamma \vdash_c N : C] \Rightarrow (\rightarrow E)\]
\[\exists B [\Gamma, x : C \vdash_c P : B \rightarrow A \& \Gamma, x : C \vdash_c Q : B \& \Gamma \vdash_c N : C] \Rightarrow (IH)\]
\[\exists B [\Gamma, x : C \vdash_c P[N/x] : B \rightarrow A \& \Gamma \vdash_c Q[N/x] : B] \Rightarrow (\rightarrow E)\]
\[\Gamma \vdash_c P[N/x] Q[N/x] : A\]
\[\Gamma \vdash_c (PQ)[N/x] : A.\]

The proof for Theorem 2.4 then becomes:

**Proof:** By induction on the definition of $\rightarrow_{\beta}$; we only show some of the cases.

- **Single-step reduction**: $(M \equiv (\lambda x.P)Q \rightarrow_{\beta} P[Q/x])$: We have to show that $\Gamma \vdash_c (\lambda x.P)Q : A$ implies $\Gamma \vdash_c P[Q/x] : A$. Notice that, if $\Gamma \vdash_c (\lambda x.P)Q : A$, then, by $(\rightarrow E)$ and $(\rightarrow I)$, there exists $C$ such that $\Gamma, x : C \vdash_c P : A$ and $\Gamma, x : C \vdash_c Q : C$. The result then follows from Lemma 2.7.

- **Multi-step reduction**: $(M \equiv PQ, N \equiv RQ, and P \rightarrow_{\beta} R)$: If $\Gamma \vdash_c PQ : A$, then, by $(\rightarrow E)$ there exists $C$ such that $\Gamma \vdash_c P : C \rightarrow A$ and $\Gamma \vdash_c Q : C$. Since $P \rightarrow_{\beta} R$ by induction on the definition of reduction, we can assume that $\Gamma \vdash_c R : C \rightarrow A$; then by $(\rightarrow E)$ we obtain $\Gamma \vdash_c RQ : A$.

We can also show that type assignment is closed for $\eta$-reduction:

**Theorem 2.8** If $\Gamma \vdash_c M : A$ and $M \rightarrow_{\eta} N$, then $\Gamma \vdash_c N : A$.

**Proof:** By induction on the definition of $\rightarrow_{\eta}$, of which only the part $\lambda x. Mx \rightarrow_{\eta} M$ is shown, where $x$ does not occur free in $M$. The other parts are dealt with by straightforward induction. Assume $x \notin \text{fo}(M)$, then
Definition 2.10 (Type substitution) defined as the operation on types that replaces type variables by types created from it. For the system as defined in this section, the ‘creation’ of types is done by a term, there is one that can be called ‘principal’ in the sense that all other types can be for terms in the Lambda Calculus.

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:

Example 2.9 We cannot derive ⊢ λx. Mx: A, so we lose the converse of the Subject Reduction property (see Theorem 2.4), i.e. Subject Expansion: If ⊢ M: A and N →β M, then ⊢ N: A. The counter example is in Exercise 2.18: take M = λbc.(λy.c) (bc), then it is easy to check that N →β M, ⊢ λbc.(λy.c) (bc): A → B → B, but not ⊢ ∅ λbc.(λy.c) (bc): A → B → 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 [22] (see Definition 2.9), 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 (ϕ → C) : T_c → T_c, where ϕ is a type variable and C ∈ T_c, is inductively defined by:

(ϕ → C) ϕ = C
(ϕ → C) ϕ' = ϕ', if ϕ' ≠ ϕ
(ϕ → C) A → B = ((ϕ → C) A) → ((ϕ → C) B)

ii) If S_1, S_2 are substitutions, then so is S_1◦ S_2, where S_1◦ S_2 A = S_1(S_2 A).

iii) SΓ = {x: SB | x: B ∈ Γ}.

iv) S(Γ, A) = (SΓ, SA).

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

So, for Curry’s system, the principal type property is expressed by: for each typeable term M, there exist a principal pair of context and type P such that ⊢ P: M: P, and for all context Γ, and types A, if ⊢ P: M: A, then there exists a substitution S such that S(P, A) = (Γ, A).

The principal type property for type assignment systems plays an important role in programming languages that model polymorphic functions, where a function is called polymorphic if it can be correctly applied to objects of various types. In fact, the principal type there 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 ⊢ M: A, then SΓ ⊢ M: SA.

Proof: By induction on the structure of derivations.

(Ax): Then M ≡ x, and x: A ∈ Γ. Notice that then x: SA ∈ SΓ, so, by rule (Ax), SΓ ⊢ x: SA.
Proposition 2.13: Then there are $N, A, C$ such that $M \equiv \lambda x \cdot N$, $A = C \rightarrow D$, 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 $S(\Gamma, x : C) = S \Gamma, x : SC$, we also have $S \Gamma, x : SC \vdash C : N : S D$. So there is a derivation that shows this, to which we can apply rule $(\rightarrow I)$, to obtain $S \Gamma \vdash C : N : S C \rightarrow S D$. Since $S C \rightarrow S D = S(C \rightarrow D) = S A$, by definition of substitutions, we get $S \Gamma \vdash C : N : S C \rightarrow S D$.

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

Principal types for $\lambda$-terms are defined using the notion of unification of types that was defined by Robinson in [51]. 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** Let $id_s$ be the substitution that replaces all type variables by themselves.

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

- $\text{unify } \varphi \quad \varphi = (\varphi \rightarrow \varphi)$,
- $\text{unify } \varphi \quad B = (\varphi \rightarrow B)$, if $\varphi$ does not occur in $B$,
- $\text{unify } A \quad \varphi = \text{unify } \varphi \quad A$,
- $\text{unify } (A \rightarrow B) \quad (C \rightarrow D) = S_2 \circ S_1$,
  where $S_1 = \text{unify } A \quad C$
  $S_2 = \text{unify } (S_1 \quad B) \quad (S_1 \quad D)$

ii) The operation $\text{UnifyContexts}$ generalises $\text{unify}$ to contexts:

- $\text{UnifyContexts } (\Gamma_0, x : A) \quad (\Gamma_1, x : B) = S_2 \circ S_1$,
  where $S_1 = \text{unify } A \quad B$
  $S_2 = \text{UnifyContexts } (S_1 \quad \Gamma_0) \quad (S_1 \quad \Gamma_1)$
- $\text{UnifyContexts } (\Gamma_0, x : A) \quad \Gamma_1 = \text{UnifyContexts } \Gamma_0 \quad \Gamma_1$, if $x$ does not occur in $\Gamma_1$.
- $\text{UnifyContexts } \emptyset \quad \Gamma_1 = id_s$.

The following property of Robinson’s unification is very important for all systems that depend on it, and formulates that $\text{unify}$ 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 ([51])** 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

$$S_2 = \text{unify } A \quad B$$
$$S_1 A = S_3 \circ S_2 A = S_3 \circ S_2 B = S_1 B.$$

or, in a diagram:
We say that $S_3$ extends $S_2$.

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

\[
\text{unify} ((\text{unify} A B) A) C = \text{unify} ((\text{unify} A C) A) B
\]

\[
= \text{unify} ((\text{unify} B C) B) A
\]

\[
= \text{unify} A ((\text{unify} B C) B)
\]

\[
= \text{unify} B ((\text{unify} A C) A)
\]

\[
= \text{unify} C ((\text{unify} A B) A)
\]

which justifies that we can introduce a ‘higher-order notation’ and write $\text{unify} A B C \ldots$ for any number of types.

The definition of principal pairs for $\lambda$-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 $\text{pp}_C M = \langle \Pi, P \rangle$ inductively by:

i) For all $x, \varphi$: $\text{pp}_C x = \langle \{x: \varphi \}, \varphi \rangle$.

ii) If $\text{pp}_C M = \langle \Pi, P \rangle$, then:

a) If $x \in \text{fv}(M)$, then, by Lemma 2.6 (iii), there is a $A$ such that $x:A \in \Pi$, and $\text{pp}_C \lambda x.M = \langle \Pi \setminus x, A \rightarrow P \rangle$.

b) otherwise $\text{pp}_C (\lambda x. M) = \langle \Pi, \varphi \rightarrow P \rangle$, where $\varphi$ does not occur in $\langle \Pi, P \rangle$.

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

\[
S_1 = \text{unify } P_1 (P_2 \rightarrow \varphi)
\]

\[
S_2 = \text{UnifyContexts } (S_1 \Pi_1) (S_1 \Pi_2),
\]

then $\text{pp}_C (M_1 M_2) = S_2 \circ S_1 (\Pi_1 \cup \Pi_2, \varphi)$.

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

This definition in fact gives an algorithm that finds the principal pair for $\lambda$-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 $\text{unify}$) 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 $\text{fresh}$ 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** ($\text{pp}_C$, principal pair algorithm for $\lambda$)
\[ \begin{align*}
p p_C \ x & = \langle x; \varphi, \varphi \rangle \\
& \text{where } \varphi = \text{fresh} \\
p p_C \ (\lambda x. M) & = \langle \Pi, A \rightarrow P \rangle, \quad \text{if } p p_C M = \langle \Pi \cup \{ x; A \}, P \rangle \\
& \langle \Pi, \varphi \rightarrow P \rangle, \quad \text{if } p p_C M = \langle \Pi, P \rangle \\
& \text{if } x \notin \Pi \\
& \text{where } \varphi = \text{fresh} \\
p p_C \ MN & = S_2 \circ S_1 \langle \Pi_1 \cup \Pi_2, \varphi \rangle \\
& \text{where } \varphi = \text{fresh} \\
& \langle \Pi_1, P_1 \rangle = p p_C M \\
& \langle \Pi_2, P_2 \rangle = p p_C N \\
& S_1 = \text{unify } P_1 P_2 \rightarrow \varphi \\
& S_2 = \text{UnifyContexts } (S_1 \Pi_1) (S_1 \Pi_2)
\end{align*} \]

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 ‘\(p p_C\)’ 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 \(\Gamma \vdash_C M: A\), then there are context \(\Pi\), type \(P\) and a substitution \(S\) such that: \(p p_C M = \langle \Pi, P \rangle\), and both \(S \Pi \subseteq \Gamma\) and \(SP = \Lambda\).

**Proof:** By induction on the structure of terms in \(\Lambda\).

\((M \equiv x)\): Then, by rule (Ax), \(x: A \in \Gamma\), and \(p p_C x = \langle \{ x; \varphi \}, \varphi \rangle\) by definition. Take \(S = (\varphi \rightarrow A)\).

\((M \equiv \lambda x. N)\): Then, by rule (\(\rightarrow I\)), there are \(C, D\) such that \(A = C \rightarrow D\), and \(\Gamma, x: C \vdash_C N: D\). Then, by induction, there are \(\Pi', P'\) and \(S'\) such that \(p p_C N = \langle \Pi', P' \rangle\), and \(S' \Pi' \subseteq \Gamma, x: C, S'P' = D\). Then either:

a) \(x\) occurs free in \(N\), and there exists an \(A'\) such that \(x: A' \in \Pi'\), and \(p p_C (\lambda x. N) = \langle \Pi' \setminus x, A' \rightarrow P \rangle\). Since \(S' \Pi' \subseteq \Gamma, x: C\), in particular \(S' A' = C\). Also \(S' (\Pi' \setminus x) \subseteq \Gamma\). Notice that now \(S' (A' \rightarrow P') = C \rightarrow D\). Take \(\Pi = \Pi' \setminus x, P = A' \rightarrow P', \) and \(S = S'\).

b) otherwise \(p p_C (\lambda x. M) = (\Pi', \varphi \rightarrow P')\), \(x\) does not occur in \(\Pi'\), and \(\varphi\) does not occur in \(\langle \Pi', P' \rangle\). Since \(S' \Pi' \subseteq \Gamma, x: C\), in particular \(S' \Pi' \subseteq \Gamma\). Take \(S = S' (\varphi \rightarrow C)\), then, since \(\varphi\) does not occur in \(\Pi'\), also \(S' \Pi' \subseteq \Gamma\). Notice that \(S (\varphi \rightarrow P') = C \rightarrow D\); take \(\Pi = \Pi', P = \varphi \rightarrow P'\).

\((M = M_1 M_2)\): Then, by rule (\(\rightarrow E\)), there exists a \(B\) such that \(\Gamma \vdash_C M_1: B \rightarrow A\) and \(\Gamma \vdash_C M_2: B\). By induction, there are \(S_1, S_2, \langle \Pi_1, P_1 \rangle = p p_C M_1\) and \(\langle \Pi_2, P_2 \rangle = p p_C M_2\) (no type variables shared) such that \(S_1 \Pi_1 \subseteq \Gamma, S_2 \Pi_2 \subseteq \Gamma, S_1 P_1 = B \rightarrow A\) and \(S_2 P_2 = B\). Notice that \(S_1, S_2\) do not interfere. Let \(\varphi\) be a type variable that does not occur in any of the pairs \(\langle \Pi, P \rangle\), and

\[ \begin{align*}
S_u & = \text{unify } P_1 (P_2 \rightarrow \varphi) \\
S_c & = \text{UnifyContexts } (S_u \Pi_1) (S_u \Pi_2)
\end{align*} \]

then, by the definition above, \(p p_C (M_1 M_2) = S_c \circ S_u \langle \Pi_1 \cup \Pi_2, \varphi \rangle\).

We will now argue that \(p p_C M_1 M_2\) is successful: since this can only fail on unification of \(P_1\) and \(P_2 \rightarrow \varphi\), we need to argue that this unification is successful. We know that

\[ \begin{align*}
S_1 \circ S_2 P_1 & = B \rightarrow A, \\
S_1 \circ S_2 P_2 & = B
\end{align*} \]

so \(P_1\) and \(P_2 \rightarrow \varphi\) have a common instance \(B \rightarrow A\), and by Proposition 2.13, \(S_u\) exists.
We now need to argue that a substitution $S$ exists such that $S(S_c \circ S_u(\Pi_1 \cup \Pi_2)) \subseteq \Gamma$. Take $S_5 = S_2 \circ S_1 \circ (\varphi \mapsto A)$, then

$$S_5 \Pi_1 \subseteq \Gamma,$$

and

$$S_5 \Pi_2 \subseteq \Gamma.$$

Since $S_5$ also unifies $P_1$ and $P_2 \rightarrow \varphi$, we know in fact also that $S_6$ exists such that

$$S_6 (S_u \Pi_1) \subseteq \Gamma,$$

and

$$S_6 (S_u \Pi_2) \subseteq \Gamma.$$

So $S_6$ also unifies $S_u \Pi_1$ and $S_u \Pi_2$, so by Proposition 2.13 there exists a substitution $S_7$ such that $S_6 = S_7 \circ S_c \circ S_u$. Take $S = S_7$. ■

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

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

$$\Gamma, x : A \ x = A$$

$$\Gamma, x : A \ y = \Gamma y, \text{ if } x \neq y$$

where we assume that $\Gamma$ is total. Also, we should define

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

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

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

**Definition 2.17** ($pp_C$: Alternative definition)

$$pt_C \ \Gamma \ x = \langle \text{Id}_S, \Gamma x \rangle$$

$$pt_C \ \Gamma \ (\lambda x. M) = \langle S, S (\varphi \rightarrow A) \rangle$$

where $\varphi = \text{fresh}$

$$\langle S, A \rangle = pt_C \ \Gamma, x : \varphi \ M$$

$$pt_C \ \Gamma \ MN = \langle S_1 \circ S_2 \circ S_3, S_1 \varphi \rangle$$

where $\varphi = \text{fresh}$

$$\langle S_3, B \rangle = pt_C \ \Gamma M$$

$$\langle S_2, A \rangle = pt_C \ (S_3 \Gamma) \ N$$

$$S_1 = \text{unify} (S_2 B) \ A \rightarrow \varphi$$

$$pp_C \ M = \langle S \emptyset, A \rangle$$

where $\langle S, A \rangle = pt_C \ \emptyset M$

We will see this approach later in Section 6 when we discuss Milner’s algorithm $W$.

We have seen that there exists an algorithm ‘$pp_C$’ that, given a term $M$, produces its principal
pair \( (\Pi, P) \) 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 \[44\] 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 6.

Exercises

Exercise 2.18 Verify the following results:

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

ii) \( \emptyset \vdash_{c} \lambda x y. x : A \rightarrow B \rightarrow A \).

iii) \( \emptyset \vdash_{c} \lambda x y z. x ( y z) : (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C \).

iv) \( \emptyset \vdash_{c} \lambda b c. c : A \rightarrow B \rightarrow B \).

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

vi) \( \emptyset \vdash_{c} \lambda b c. (\lambda x y. x) (b c) : (B \rightarrow A) \rightarrow B \rightarrow B \).

vii) \( \emptyset \vdash_{c} (\lambda b c. a c (b c)) (\lambda x y. x) : (B \rightarrow A) \rightarrow B \rightarrow B \).

Exercise 2.19 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 \( \{ x : B \mid x : B \in \Gamma \text{ and } x \in \text{fv}(M) \} \vdash_{c} M : A \).

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

so show Lemma 2.6.

* Exercise 2.20 Finish the proof of Theorem 2.4; all added cases follow by straightforward induction.

Exercise 2.21 Show that, for all substitutions \( S \) and types \( A \) and \( B \), \( S(A \rightarrow B) = S A \rightarrow S B \).

* Exercise 2.22 Show that unify (unify \( A B A \)) \( C = \text{unify} A \) (unify \( B C C \)). (Notice that unify \( A B \) is a substitution that gets applied to \( A \) before the second unification on the left takes place (with \( C \)); moreover, we could have applied it to \( B \) rather than \( A \) with no resulting difference.)

Exercise 2.23 Show that, if \( \text{pp}_{c} M = (\Pi, P) \), then \( \Pi \vdash_{c} M : P \). You’ll need Lemma 2.11 here.

Exercise 2.24 Extend the definition of type assignment of Definition 2.2 to \( \lambda x \). Show that also this extended system satisfies subject reduction (as Theorem 2.4).

Exercise 2.25 Extend the definition of the principal type algorithm of Definition 2.14 (or 2.15) to \( \lambda x \).
3 Combinatory Logic

In this section, we will focus on Curry’s Combinatory Logic [22], 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 [39] that we will study in the next section, with the restriction that formal parameters of function symbols are not allowed to have structure, and right-hand sides of term rewriting rules are constructed of term-variables only.

**Definition 3.1 (Combinatory Logic [22])** The original definition of Combinatory Logic defines two rules:

\[
\begin{align*}
K & \quad x \quad y \rightarrow x, \\
S & \quad x \quad y \quad z \rightarrow xz(yz)
\end{align*}
\]

and defines terms as

\[
t ::= K \mid S \mid t_1 \quad t_2
\]

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 SKKx → Kx(Kx) → x. We therefore will consider the following rule to be present as well.

\[
I \quad x \rightarrow x
\]

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.

### 3.1 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 Lambda Calculus, and show that every λ-term can be translated to a cl program. This shows of course that Combinatory Logic is Turing Complete: all computable functions can be expressed in terms of S, K, and I.

**Example 3.2** For cl, the interpretation of terms in Λ is given by:

\[
\begin{align*}
\langle x \rangle_\lambda &= x, & \text{for all } x \in V, \\
\langle S \rangle_\lambda &= (\lambda xyz. xz(yz)), \\
\langle K \rangle_\lambda &= (\lambda xy. x), \\
\langle I \rangle_\lambda &= (\lambda x. x), \\
\langle t_1 t_2 \rangle_\lambda &= \langle t_1 \rangle_\lambda \langle t_2 \rangle_\lambda
\end{align*}
\]

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

**Definition 3.3** The mapping \( \llbracket \cdot \rrbracket : \Lambda \rightarrow T_{cl} \) is defined by:

\[
\begin{align*}
\llbracket x \rrbracket_{cl} &= x, \\
\llbracket \lambda x. M \rrbracket_{cl} &= \text{Fun} \ x \ llbracket M \rrbracket_{cl}, \\
\llbracket MN \rrbracket_{cl} &= \llbracket M \rrbracket_{cl} \llbracket N \rrbracket_{cl}
\end{align*}
\]

where Fun x t,\(^1\) with t \( \in T_{cl} \), is defined by induction on the structure of t:

\(^1\) Fun is normally called \( \lambda^* \) in the literature.
Fun \ x \ x = I, \\
Fun \ x \ t = K \ t, \quad \text{if } x \not\in t, \\
Fun \ x \ (t_1 \ t_2) = S (Fun \ x \ t_1) (Fun \ x \ t_2) \quad \text{otherwise}

Notice that the auxiliary function Fun, that takes a variable and a term in \( \mathcal{T}_cl \) and returns a term in \( \mathcal{T}_cl \), is only evaluated in the definition of \( \llbracket . \rrbracket _{cl} \) 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 3.4** (Fun x t) \( \mapsto t[v/x] \).

**Proof:** By induction on the definition of Fun.

\[
\begin{align*}
(Fun \ x \ x) \ t &= I \ t \\
(Fun \ x \ t_1) \ t_2 &= K \ t_1 \ t_2 \\
(Fun \ x \ (t_1 \ t_2)) \ t_3 &= S (Fun \ x \ t_1) (Fun \ x \ t_2) \ t_3 \\
&\quad \quad \quad \quad \quad \rightarrow (IH) \\
&\quad \quad \quad \quad \quad t_1 \ t_3 / x \ t_2 [t_3 / x] = (t_1 \ t_2) \ t_3 / x.
\end{align*}
\]

For the interpretations defined above the following property holds:

**Lemma 3.5** ([11])

i) \( \langle Fun \ x \ t \rangle_\lambda \mapsto_\beta \lambda x. \langle t \rangle_\lambda \)

ii) \( \langle \llbracket M \rrbracket _{cl} \rangle_\lambda \mapsto_\beta M \).

iii) If \( t \rightarrow u \) in CL, then \( \langle t \rangle_\lambda \rightarrow_\beta \langle u \rangle_\lambda \).

**Proof:**

i) By induction on the definition of the function Fun.

a) \( \text{Fun } x x = I, \text{ and } \langle I \rangle_\lambda = \lambda x.x. \)

b) \( \text{Fun } x t = K \ t, \text{ and } \langle K \ t \rangle_\lambda = \langle K \rangle_\lambda \langle t \rangle_\lambda = \langle \lambda b. a \rangle \langle t \rangle_\lambda \rightarrow_\beta \lambda b. \langle t \rangle_\lambda, \text{ if } x \not\in t. \)

c) \( \text{Fun } x \ (t_1 \ t_2) = S (\text{Fun } x \ t_1) (\text{Fun } x \ t_2), \) and

\[
\begin{align*}
\langle S \rangle_\lambda \langle Fun \ x \ t_1 \rangle_\lambda \langle Fun \ x \ t_2 \rangle_\lambda &\quad \rightarrow_\beta (IH) \\
\langle S \rangle_\lambda \lambda x. \langle t_1 \rangle_\lambda \lambda x. \langle t_2 \rangle_\lambda &\quad \rightarrow_\beta (\lambda x. (\langle t_1 \rangle_\lambda \langle t_2 \rangle_\lambda)) \\
\lambda c. (\lambda x. (\langle t_1 \rangle_\lambda \langle t_2 \rangle_\lambda)) c (\lambda x. (\langle t_2 \rangle_\lambda c) &\quad \rightarrow_\beta (\lambda x. (\langle t_1 \rangle_\lambda \langle t_2 \rangle_\lambda)) (c / x) \\
\lambda x. (\langle t_1 \rangle_\lambda \langle t_2 \rangle_\lambda) &\quad \rightarrow_\beta (\lambda x. (\langle t_1 \rangle_\lambda \langle t_2 \rangle_\lambda) (c / x) \right)
\end{align*}
\]

ii) By induction on the structure of \( (\lambda x. \cdot) \)-terms.

a) \( M = x. \) Since \( \langle x \rangle_\lambda \lambda x. = x \), this is immediate.

b) \( M = \lambda x. N. \) Since \( \langle \llbracket \lambda x. N \rrbracket _{cl} \rangle_\lambda = \langle \text{Fun } x \ \llbracket N \rrbracket _{cl} \rangle_\lambda \rightarrow_\beta \lambda x. (\llbracket N \rrbracket _{cl} \rangle_\lambda \) by the previous part, and \( \lambda x. (\llbracket N \rrbracket _{cl} \rangle_\lambda \rightarrow_\beta \lambda x. N \) by induction.

c) \( M = PQ. \) Since \( \langle \llbracket PQ \rrbracket _{cl} \rangle_\lambda = \langle \llbracket P \rrbracket _{cl} \rangle_\lambda \langle \llbracket Q \rrbracket _{cl} \rangle_\lambda \rightarrow_\beta PQ \) by induction.

iii) We focus on the case that \( t = Ct_1 \cdots t_n \) for some name \( C \) with arity \( n. \) Let \( Cx_1 \cdots x_n \rightarrow t' \) be the definition for \( C, \) then \( u = t'[t / x_i] \). Notice that \( \langle t \rangle_\lambda = (\lambda x_1 \cdots x_n. (t')_\lambda) \langle t_1 \rangle_\lambda \cdots \langle t_n \rangle_\lambda \rightarrow_\beta \langle t' \rangle_\lambda \langle t_1 / x_i \rangle_\lambda = \langle t'[t / x_i] \rangle_\lambda = \langle u \rangle_\lambda. \)

**Example 3.6** \( \llbracket \lambda y. x \rrbracket _{cl} = \text{Fun } x \ \llbracket \lambda y. x \rrbracket _{cl} \)

\[
= \text{Fun } x \ (\text{Fun } y \ x) \\
= \text{Fun } x \ (K \ x) \\
= S (\text{Fun } x \ K) (\text{Fun } x \ x) \\
= S (KK) \ I
\]
and \( \langle \lambda xy.x \rangle_{\text{cl}} \lambda = \langle S (KK) \rangle_\lambda \)
\[ = \langle \lambda xyz.xz(yz) \rangle ((\lambda xy.x)\lambda xy.x) \lambda x.x \]
\[ \rightarrow_\beta \lambda xy.x. \]

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

Example 3.7 ([11])

i) \( SK \) is a normal form, but \( \langle SK \rangle_\lambda \rightarrow_\beta \lambda xy.y; \)

ii) \( t = S (K (SI)) (K (SI)) \) is a normal form, but \( \langle t \rangle_\lambda \rightarrow_\beta \lambda c.(\lambda x.xx) (\lambda x.xx) \), which does not have a \( \beta \)-normal form, and not even a head-normal form;

iii) \( t = SK (SI (SI)) \) has no normal form, while \( \langle t \rangle_\lambda \rightarrow_\beta \lambda x.x. \)

3.2 Extending cl

Bracket abstraction algorithms like \( \langle \rangle \lambda \) are used to translate \( \lambda \)-terms to combinator systems, and form, together with the technique of lambda lifting the basis of the Miranda [54] compiler. It is possible to define such a translation also for combinator systems that contain other combinators. With some accompanying optimization rules they provide an interesting example. If in the bracket abstraction we would use the following combinator set:

\[
\begin{align*}
I & \rightarrow x \\
K & x y \rightarrow x \\
B & x y z \rightarrow x(yz) \\
C & x y z \rightarrow xzy \\
S & x y z \rightarrow xz(yz)
\end{align*}
\]

then we could, as in [24], extend the notion of reduction by defining the following optimizations:

\[
\begin{align*}
S (K x) (K y) & \rightarrow K (x y) \\
S (K x) & I \rightarrow x \\
S (K x) & y \rightarrow B x y \\
S & x (K y) \rightarrow C x y
\end{align*}
\]

The correctness of these rules is easy to check.

Notice that, by adding this optimisation, we are stepping outside the realm of \( \lambda \)-calculus by adding pattern matching: the rule \( S (K x) (K y) \rightarrow K (x y) \) expresses that this reduction can only take place when the first and second argument of \( S \) are of the shape \( K t \). So, in particular, these arguments can not be any term as is the case with normal combinator rules, but must have a precise structure. In fact, adding these rules introduces pattern matching and would give a notion of rewriting that is known as term rewriting, which we will look at in Section 7; we will see there that we cannot express (arbitrary) pattern matching in the \( \lambda \)-calculus.

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

3.3 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 \( \Lambda^N \) we saw above.

Definition 3.8 (Type Assignment for cl) Type assignment on terms in \( \text{cl} \) is defined by the following natural deduction system.
Example 3.9 It is easy to check that the term SKII can be assigned the type \( \varphi \to \varphi \).

\[
\begin{align*}
\vdash S &: B \to C \to D \to \varphi \to \varphi \\
\vdash K &: B \\
\vdash SK &: C \to D \to \varphi \to \varphi \\
\vdash I &: C \\
\vdash SKI &: D \to \varphi \to \varphi \\
\vdash I &: D \\
\vdash SKII &: \varphi \to \varphi
\end{align*}
\]

where \( B = (\varphi \to \varphi) \to (\varphi \to \varphi) \to \varphi \to \varphi \)

\( C = (\varphi \to \varphi) \to \varphi \to \varphi \)

\( D = \varphi \to \varphi \)

Notice that the system of Definition 3.8 only specifies how to type terms; it does not specify how to type a program - we need to deal with definitions as well; we will do so below. Notice that each type that can be assigned to each \( n \in \{S,K,I\} \) using the system above is a substitution instance of the principal type of the body:

\[
\begin{align*}
pp_c(\lambda x y z. x y (y z)) &= (\varphi_1 \to \varphi_2 \to \varphi_3) \to (\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_3, \\
pp_c(\lambda x y . x) &= \varphi_4 \to \varphi_5 \to \varphi_4, \\
pp_c(\lambda x . x) &= \varphi_6 \to \varphi_6.
\end{align*}
\]

This feature is used when defining how to type a program in the next section.

The relation between type assignment in the Lambda Calculus and that in \( \text{cl} \) is very strong, as Exercise 3.10 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 \( \text{cl} \) was the first of this kind of property proven, by J.R. Hindley [32].

3.4 Exercises

* Exercise 3.10 Show, by induction on the definition of \( \text{Fun} \), \( \llbracket \rrbracket_{\text{cl}} \) and \( \langle \rangle_\lambda \), that

i) \( \Gamma, y: A \vdash_{\varepsilon_{\text{cl}}} t : B \) implies \( \Gamma, y: A \vdash_{\varepsilon_{\text{cl}}} \text{Fun} y t : A \to B \).

ii) \( \Gamma \vdash_{\varepsilon} M : A \) implies \( \Gamma \vdash_{\varepsilon_{\text{cl}}} \llbracket M \rrbracket_{\text{cl}} : A \).

iii) \( \Gamma \vdash_{\varepsilon_{\text{cl}}} t : A \) implies \( \Gamma \vdash_{\varepsilon} \langle t \rangle_\lambda : A \).

4 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:

\[ Ix = x \]

\[ II \]

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 \( \lambda \)-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 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 [44] (see also Section 6). 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 \( \Lambda^N \), a \( \lambda \)-calculus with names \( n \) and definitions like \( n = M \), which we will present below. Each occurrence of a name \( n \) 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 \( n \) to that term, and store the principal type of the right-hand side with \( n \) in an environment: the type of \( n \) is called generic, and the term defined by \( n \) is called a polymorphic function. In fact, we would like to model that each call to \( n \) can have a different type; we then call such a name polymorphic.

### 4.1 The language \( \Lambda^N \)

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

**Definition 4.1**

i) The syntax for programs in \( \Lambda^N \) is defined by:

\[
M ::= x | (M_1 \cdot M_2) | (\lambda x.M) \\
name ::= \text{'string of characters'} \\
Defs ::= name = M; Defs | \epsilon \quad (M \text{ is closed}) \\
Term ::= x | name | (Term_1 \cdot Term_2) | (\lambda x.\text{Term}) \\
Program ::= Defs : Term
\]

Like before, redundant brackets will be omitted.

ii) Reduction on terms in \( \Lambda^N \) is defined as normal \( \beta \)-reduction for the Lambda Calculus, extended by \( n \rightarrow M \), if \( n = M \) appears in the list of definitions.

We have put the restriction that in \( n = M, M \) is a closed \( \lambda \)-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 7). 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 \( \lambda \)-terms; we could even restrict bodies of definitions such that no redexes occur there, but that is not necessary for the present purpose.
4.2 Type assignment for $\Lambda^N$

In this section we will develop a notion of type assignment on programs in $\Lambda^N$. 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 $n$, we will force the type for $n$ here to be an instance of the principal type by copying the type in the environment, and allowing unification to change the copy, thereby instantiating it.

The notion of type assignment we will study below for this extended calculus is an extension of Curry’s system for the Lambda Calculus, and has been studied extensively in the past, e.g. in [32, 27, 9]. Basically, Curry (principal) types are assigned to named $\lambda$-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 $\lambda$-term, and the principal type for the term is derived as discussed above. That is what the system below specifies. Of course, when typing a term, we only need to check that the environment is sound, i.e. correct.

The notion of type assignment defined below uses the notation $\Gamma;\mathcal{E} \vdash t : A$; here $\Gamma$ is a context, $\mathcal{E}$ an environment, $t$ either a $\lambda$-term or a term in Term. We also use the notion $\mathcal{E} \vdash \text{Defs} : \emptyset$; here $\emptyset$ 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 4.2 (Type Assignment for $\Lambda^N$)**

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

ii) Type assignment (with respect to $\mathcal{E}$) is defined by the following natural deduction system.

Notice the use of a substitution in rule (name).

$\begin{align*}
(Ax) : & \Gamma, x : A; \mathcal{E} \vdash x : A \\
(\rightarrow I) : & \Gamma, x : A; \mathcal{E} \vdash t : B \\
(\epsilon) : & \emptyset \vdash \epsilon : \emptyset \\
(name) : & \Gamma; \mathcal{E}, \text{name} : A \vdash \text{name} : SA \\
(\rightarrow E) : & \Gamma; \mathcal{E} \vdash t_1 : A \rightarrow B, \Gamma; \mathcal{E} \vdash t_2 : A \\
(\text{Defs}) : & \emptyset \vdash M : A \\
(\text{Program}) : & \emptyset \vdash \text{Defs} : \emptyset, \Gamma; \mathcal{E} \vdash t : A \\
& \Gamma; \mathcal{E} \vdash \text{Defs} : t : A
\end{align*}$

Notice that, in rule (Defs), we use $\Gamma_c$ to type the pure $\lambda$-term $M$, and insist that it 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 just found for $M$. For this notion of type assignment, we can easily show the subject reduction result (see Exercise 4.6). 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.

Remark that for $\text{cl}$ the environment could specify

$\mathcal{E} (S) = (\varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3) \rightarrow (\varphi_1 \rightarrow \varphi_2) \rightarrow \varphi_1 \rightarrow \varphi_3$

$\mathcal{E} (K) = \varphi_4 \rightarrow \varphi_5 \rightarrow \varphi_4$

$\mathcal{E} (I) = \varphi_6 \rightarrow \varphi_6$

but that the system allows for any valid type for the $\lambda$-terms to be added to the environment.

Notice that the substitution in rule (name) is essential: without it, we would not be able to
type the term \( II \).

**Example 4.3** We can now derive:

\[
\begin{align*}
\frac{x : \phi \vdash x : \phi}{x : \phi \vdash x : \phi} & \quad (Ax) \\
\frac{\emptyset : I : \phi \vdash \phi \to I : (A \to A) \to A \to A}{\emptyset : I : \phi \vdash \phi \to I : (A \to A) \to A \to A} & \quad (\text{name}) \\
\frac{\emptyset : I : \phi \vdash \phi \to I : A \to A}{\emptyset : I : \phi \vdash \phi \to I : A \to A} & \quad (\to E)
\end{align*}
\]

\[
\frac{\emptyset \vdash \lambda x : \phi \to \phi}{\emptyset \vdash \epsilon : \emptyset} & \quad (\text{Defs}) \\
\frac{I : \phi \vdash \phi \to \phi \to I : \lambda x : \phi}{I : \phi \vdash \phi \to I : \lambda x : \phi} & \quad (\text{Program})
\]

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

As mentioned above, for every definition a pair – consisting of the principal type (found for the right-hand side) and the name of the defined function – is put in the *environment*. Every time \( n \) is encountered in a term, the algorithm looks in the environment to see what its (principal) type is. It takes a fresh instance of this type, and uses this instance 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 \( n \). 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 \( n \) do not affect the type already found for another. Moreover, this way the types actually used for \( n \) will always be substitution instances of the principal type that is associated to \( n \) in the environment.

The algorithm that calculates the types for a program in \( \Lambda^N \) is defined by:

**Definition 4.4** (*Principal environments and types for \( \Lambda^N \)*)

\[
\begin{align*}
pp_{\Lambda^N} \ x \ \emptyset & = \langle \{x : \phi\}, \phi \rangle \\
& \quad \text{where } \phi = \text{fresh} \\
pp_{\Lambda^N} \ n \ \emptyset & = \langle \emptyset, \text{FreshInstance} \ (\emptyset \ n) \rangle \\
pp_{\Lambda^N} \ (\lambda x. M) \ \emptyset & = \langle \Gamma, A \to B \rangle, \quad \text{if } pp_{\Lambda^N} \ M \ \emptyset = \langle \Gamma \cup \{x : A\}, B \rangle \\
& \quad \langle \Gamma, \phi \to B \rangle, \quad \text{if } pp_{\Lambda^N} \ M \ \emptyset = \langle \Gamma, B \rangle, x \notin \Gamma \\
& \quad \text{where } \phi = \text{fresh} \\
pp_{\Lambda^N} \ MN \ \emptyset & = S_2 \circ S_1 \langle \Gamma_1 \cup \Gamma_2, \phi \rangle \\
& \quad \text{where } \phi = \text{fresh} \\
& \quad \langle \Gamma_1, A \rangle = pp_{\Lambda^N} \ M \ \emptyset \\
& \quad \langle \Gamma_2, B \rangle = pp_{\Lambda^N} \ N \ \emptyset \\
& \quad S_1 = \text{unify} \ A \to B \to \phi \\
& \quad S_2 = \text{UnifyContexts} \ (S_1 \Gamma_1) \ (S_1 \Gamma_2) \\
pp_{\Lambda^N} \ (e : N) \ \emptyset & = pp_{\Lambda^N} \ N \ \emptyset \\
pp_{\Lambda^N} \ ((n = M) \ ; \text{Defs} : N) \ \emptyset & = pp_{\Lambda^N} \ (\text{Defs} : N) \ \emptyset, n : A \\
& \quad \text{where } \langle \emptyset, A \rangle = pp_{\Lambda^N} \ M \ \emptyset
\end{align*}
\]

Notice that the environment gets extended by the call to \( pp_{\Lambda^N} \) that types a program by travers-
ing 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 \( pp_{\lambda N} \), restricted to \( \lambda \)-terms, is exactly \( pp_{\lambda C} \), and that then the second argument
- the environment - becomes obsolete.

We leave the soundness of this algorithm as an exercise.

Exercises

Exercise 4.5 Prove that, if \( t \rightarrow t' \) in \( \lambda N \), then \( \langle t \rangle \beta \rightarrow \beta \langle t' \rangle \lambda \).
Exercise 4.6 If \( \Gamma; \mathcal{E} \vdash t : A \), and \( t \rightarrow t' \), then \( \Gamma; \mathcal{E} \vdash t' : A \).

* Exercise 4.7 If \( pp_{\lambda N} \ (\text{Defs} : N) \ \mathcal{E} = \langle \Gamma, A \rangle \), then \( \Gamma; \mathcal{E} \vdash N : A \).

5 Dealing with recursion

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

5.1 The language \( \lambda^{NR} \)

First, we define \( \lambda^{NR} \), that enables us to focus on polymorphism and recursion, by introducing
names for \( \lambda \)-terms and allowing names to occur in all terms, so also in definitions.

Definition 5.1 The syntax for programs in \( \lambda^{NR} \) is defined by:

\[
\begin{align*}
\text{name} & ::= \text{‘string of characters’} \\
M & ::= x \mid \text{name} \mid (M_1 \cdot M_2) \mid (\lambda x. M) \\
\text{Defs} & ::= (\text{name} = M) ; \text{Defs} \mid e \quad (M \text{ is closed}) \\
\text{Program} & ::= \text{Defs} : M
\end{align*}
\]

Reduction on \( \lambda^{NR} \)-terms is defined as before for \( \lambda N \).

Notice that, with respect to \( \lambda N \), 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 5.2 \( S = \lambda xyz. xz(yz) \) ;
\( K = \lambda xy.x \) ;
\( I = \text{SKK} \) ;
\( Y = \lambda m. m(Y m) \) :
\( Y I \)

5.2 Expressing recursion in the Lambda Calculus

Programs written in \( \lambda^{NR} \) can easily be translated to \( \lambda \)-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 5.3 As an illustration, take the well-known factorial function
Fac \( n = 1 \), \( n = 0 \)
Fac \( n = n \times (\text{Fac } n-1) \), otherwise

In order to create the \( \lambda \)-term that represents this function, we first observe that we could write it in \( \Lambda^{\text{NR}} \) as:

\[
\text{Fac} = \lambda n. (\text{cond } (n = 0) 1 (n \times (\text{Fac } n-1))),
\]

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

\[
YM =_\beta M(YM)
\]

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

\[
Y(\lambda x. N) =_\beta (\lambda x. N) (Y(\lambda x. N)) \rightarrow_\beta N[Y(\lambda x. N)/x]
\]

This gives us an equation like:

\[
F = N[F/x]
\]

with solution \( F = Y(\lambda x. N) \). 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

\[
F = C[F] = (\lambda f. C[f])F = Y(\lambda f. C[f])
\]

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

\[
\text{Fac} = Y(\lambda f. \lambda n. \text{cond } (n = 0) 1 (n \times (f (n - 1))))
\]

Notice that this approach is going to cause a problem when we want to type terms in \( \Lambda^{\text{NR}} \); we cannot just translate the term, type it in \( \Lambda \) and give that type to the original \( \Lambda^{\text{NR}} \) term, since, for recursive terms, this would involve typing \( Y \). This is impossible.

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

\[
\text{Fac} = \lambda n. \text{cond } (n = 0) 1 (n \times (\text{Fac } n-1))
\]

so \( \text{num} \rightarrow \text{num} \) should also be the type for

\[
\lambda n. \text{cond } (n = 0) 1 (n \times (\text{Fac } n-1))
\]

Notice that, by checking its context, the occurrence of \( \text{Fac} \) in this term has type \( \text{num} \rightarrow \text{num} \) as well. Therefore,

\[
\lambda gn. \text{cond } (n = 0) 1 (n \times (g (n - 1)))
\]

should have type \( (\text{num} \rightarrow \text{num}) \rightarrow \text{num} \rightarrow \text{num} \). 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

\[
((\text{num} \rightarrow \text{num}) \rightarrow \text{num} \rightarrow \text{num}) \rightarrow \text{num} \rightarrow \text{num}.
\]

Therefore, for \( \Lambda^{\text{NR}} \) 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: \( \Gamma; E; f: A \vdash C[f]: A \).

Example 5.4 We can generalise the observation of the previous example, and deduce that, in order to type both \( YM \) and \( M(YM) \) with the same type, we need to assume that \( Y \) has type \( (A \rightarrow A) \rightarrow A \), for all \( A \). So when enforcing typeable recursion for the \( \lambda \)-calculus, we would need to extend the language \( \Lambda \) by adding a constant term.
add the reduction rule

\[ \text{YM} \to M(\text{YM}) \]

and add the type assignment rule

\[
\Gamma \vdash Y : (A \to A) \to A
\]

This extension is enough to encode all typeable \( \Lambda^{NR} \) 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, \( \text{fix} \ g. E \), but the way to type this is essentially the same.

5.3 Type assignment and algorithms

Naturally, the notion of type assignment for \( \Lambda^{NR} \) is an extension of that of \( \Lambda^N \).

**Definition 5.5 (Type assignment for \( \Lambda^{NR} \))** Type assignment for programs in \( \Lambda^{NR} \) is defined through the following rules:

\[
\begin{align*}
(Ax) : & \quad \Gamma, x : A; \mathcal{E} \vdash x : A \\
(\to I) : & \quad \Gamma, x : A; \mathcal{E} \vdash M : B \quad \Gamma; \mathcal{E} \vdash \lambda x. M : A \rightarrow B \\
(\to E) : & \quad \Gamma; \mathcal{E} \vdash M : A \rightarrow B \quad \Gamma; \mathcal{E} \vdash N : A \\
(\text{Rec}) : & \quad \Gamma; \mathcal{E}, \text{name}: A \vdash \text{name} : A \\
(\text{Name}) : & \quad \Gamma; \mathcal{E}, \text{name}: A \vdash \text{name} : SA \\
(\epsilon) : & \quad \mathcal{E} \vdash \epsilon : \emptyset \\
\text{(Program)} : & \quad \mathcal{E} \vdash \text{Defs} : \emptyset \quad \Gamma; \mathcal{E} \vdash M : A \\
\end{align*}
\]

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

When looking to type a program 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 \( n = M \) 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 are simply stored in the environment.

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

**Definition 5.6 (Principal environments and types for \( \Lambda^{NR} \))**
\[ pp_{\text{NR}} \ x \ E = \langle \{ x: \varphi \}, \varphi, E \rangle \]
where \( \varphi = \text{fresh} \)

\[ pp_{\text{NR}} \ n \ E = \langle \varnothing, A, E \rangle, \text{if } n \text{ is recursive and depends on this occurrence} \]
\[ \langle \varnothing, \text{FreshInstance} A, E \rangle, \text{otherwise} \]
where \( n: A \in E \)

\[ pp_{\text{NR}} \ (\lambda x. M) \ E = \langle \Gamma, A \rightarrow B, E' \rangle, \text{if } pp_{\text{NR}} M E = \langle \Gamma \cup \{ x: A \}, B, E' \rangle \]
\[ \langle \Gamma, \varphi \rightarrow B, E' \rangle, \text{if } pp_{\text{NR}} M E = \langle \Gamma, B, E' \rangle, x \notin \Gamma \]
where \( \varphi = \text{fresh} \)

\[ pp_{\text{NR}} \ (MN) E = S_2 \circ S_1 (\langle \Gamma_1 \cup \Gamma_2, \varphi, E'' \rangle) \]
where \( \varphi = \text{fresh} \)
\[ \langle \Gamma_1, B, E' \rangle = pp_{\text{NR}} M E \]
\[ \langle \Gamma_2, A, E'' \rangle = pp_{\text{NR}} N E' \]
\[ S_1 = \text{unify } A \rightarrow \varphi \]
\[ S_2 = \text{UnifyContexts} (S_1 \Gamma_0) (S_1 \Gamma_1) \]

\[ pp_{\text{NR}} (n = M; \text{Defs}: N) E = pp_{\text{NR}} (\text{Defs}: N) E, n: S(A) \text{ otherwise} \]
where \( \varphi = \text{fresh} \)
\[ \langle \Gamma, A, E' \rangle = pp_{\text{NR}} M E, n: \varphi \]
\[ S = \text{unify } A (E' n) \]

\[ pp_{\text{NR}} (e : N) E = pp_{\text{NR}} N E \]

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

As before in Definition 2.17, we can give an alternative algorithm that does not need to unify contexts; since it changes also the environment, that is not returned as result.

**Definition 5.7 (Milner)**

\[
\text{Milner } \Gamma \ x \ E = \langle \text{Id}_S, \Gamma x \rangle
\]
\[
\text{Milner } \Gamma \ n \ E = \langle \text{Id}_S, E n \rangle, \text{if } n \text{ is recursive and depends on this occurrence} \]
\[
\langle \text{Id}_S, \text{FreshInstance} (E n) \rangle, \text{otherwise} \]
\[
\text{Milner } \Gamma \ (\lambda x. M) E = \langle S_1 S_2 S_3, \varphi \rightarrow C \rangle
\]
where \( \varphi = \text{fresh} \)
\[ \langle S_1, C \rangle = \text{Milner } (\Gamma, x: \varphi) M E \]
\[
\text{Milner } \Gamma \ (MN) E = \langle S_3 \circ S_2 \circ S_1, S_3 \varphi \rangle
\]
where \( \varphi = \text{fresh} \)
\[ \langle S_1, C \rangle = \text{Milner } \Gamma M E \]
\[ \langle S_2, A \rangle = \text{Milner } (S_1 \Gamma) (S_1 E) N \]
\[ S_3 = \text{unify } (S_2 C) A \rightarrow \varphi \]

\[
\text{FindType } \langle e : N \rangle E = \langle S \varnothing, A \rangle
\]
where \( \langle S, A \rangle = \text{Milner } \varnothing N E \)
\[ \text{FindType } (n = M; \text{Defs} : N) \mathcal{E} = \text{FindType } (\text{Defs}, N) \mathcal{E}, n : A, \quad \text{n not recursive} \]

where \( (S, A) = \text{Milner } \otimes M \mathcal{E} \)

= \text{FindType } (\text{Defs}, N) \mathcal{E}, n : s_1 A], \) otherwise

where \( \varphi = \text{fresh} \)

\( (S_2, A) = \text{Milner } \otimes M \mathcal{E}, n : \varphi \)

\( s_1 = \text{unify } (S_2 \varphi) A \)

Exercises

Exercise 5.8 Take \( P \) to be the program in Exercise 5.2. Find an environment \( \mathcal{E} \) such that \( \emptyset ; \mathcal{E} \vdash P : \phi \), and build the derivation that justifies this judgement.

6 Milner’s ML

In [44], a formal type discipline was presented for polymorphic procedures in a simple programming language called \( \mathcal{L}_\text{ml} \), designed to express that certain procedures work well on objects of a wide variety. This kind of procedure is called (shallow) polymorphic, and it is essential to obtain enough flexibility in programming.

\( \mathcal{L}_\text{ml} \) is based on the Lambda 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 [25] an algorithm \( W \) was defined that has become very famous and is implemented in a type checker that is embedded in the functional programming language \( \text{ml} \). \( W \) is shown to be semantically sound (based on a formal semantics for the language [44] – so typed programs cannot go wrong), and syntactically sound, so if \( W \) accepts a program, then it is well typed.

We will also present a variant as defined by A. Mycroft [47], 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 [54] 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.

6.1 The ML Type Assignment System

In this subsection, we present Milner’s Type Assignment System as was done in [26], and not as in [25, 44], because the former presentation is more detailed and clearer.

**Definition 6.1 (ML Expressions)** i) ML expressions are defined by the grammar:

\[
E ::= x \mid c \mid (\lambda x. E) \mid (E_1 \cdot E_2) \mid (\text{let } x = E_1 \text{ in } E_2) \mid (\text{fix } g. E)
\]

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

\[
(\text{let } x = E_1 \text{ in } E_2) \rightarrow_{\mathcal{L}_\text{ml}} E_2[E_1/x] \\
\text{fix } g. E \rightarrow_{\mathcal{L}_\text{ml}} E[(\text{fix } g. E)/g]
\]

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 \( (\text{let } x = E_2 \text{ in } E_1) \) 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 \((\text{let } x = E_2 \text{ in } E_1)\) 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 \((\text{let } x = E_2 \text{ in } E_1)\). As argued in [25], it is of course possible to set up type assignment in such a way that \(E_1[E_2/x]\) is typed every time the let-construct is encountered, but that would force us to type \(E_2\) perhaps many times; even though in normal implementations \(E_2\) is shared, the various references to it could require it to have different types. The elegance of the let-construct is that \(E_2\) is typed only once, and that its (generalised) principal type is used when typing \(E_1[E_2/x]\).

The language defined in [44] also contains a conditional-structure \((\text{if-then-else})\). It is not present in the definition of \(L_{\text{ml}}\) in [25], so we have omitted it here. The construct fix is introduced to model recursion; it is present in the definition of \(L_{\text{ml}}\) in [44], but not in [25]. 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 \(\lambda \ldots\)

The set of ML types is defined much in the spirit of Curry types (extended with type constants 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 \(\sigma, \tau\). 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 \(\tau\) is defined as the replacement of free type variables by renaming the generic type variables of \(\tau\) if necessary.

**Definition 6.2 ([44])** i) The set of ML types is defined in two layers.

\[
A, B ::= \varphi | C | (A \rightarrow B) \\
\sigma, \tau ::= A | (\forall \varphi. \tau)
\]

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

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

\[
\begin{align*}
(\varphi \mapsto C) \varphi &= C \\
(\varphi \mapsto C) C &= C \\
(\varphi \mapsto C) \varphi' &= \varphi' \quad \text{(if } \varphi' \neq \varphi) \\
(\varphi \mapsto C) A \rightarrow B &= ((\varphi \mapsto C) A) \rightarrow ((\varphi \mapsto C) B) \\
(\varphi \mapsto C) \forall \varphi'. \psi &= \forall \varphi'. ((\varphi \mapsto C) \psi)
\end{align*}
\]

iii) A type obtained from another by applying an ML-substitution is called an instance.

iv) We define the relation ‘\(\succeq\)’ as the least reflexive and transitive relation such that:

\[
\forall \varphi. \psi \succeq \forall \varphi'. \psi [B/\varphi].
\]

provided no \(\varphi'\) is free in \(\psi\). If \(\sigma \succeq \tau\), we call \(\tau\) a generic instance of \(\sigma\).

v) We define the relation ‘\(\preceq\)’ as the least pre-order such that:

\[
\begin{align*}
\tau &\preceq \forall \varphi. \tau, \\
\forall \varphi. \tau &\preceq \tau [B/\varphi].
\end{align*}
\]

Since \(\varphi'\) is bound in \(\forall \varphi'. \psi\), we can safely assume that, in \((\varphi \mapsto C) \forall \varphi'. \psi, \varphi \neq \varphi'\) and \(\varphi' \notin \text{fv}(C)\).
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 6.7.

**Definition 6.3** \( T \vdash A = \forall \varphi . A \) where \( \varphi \) are the type variables that occur free in \( A \) but not in \( \Gamma \).

In the following definition we will give the derivation rules for Milner’s system as presented in [26]; in the original paper [44] no derivation rules are given; instead, it contains a rather complicated definition of ‘well typed prefixed expressions’.

**Definition 6.4** ([26]) We assume the existence of a function \( \nu \) that maps each constant to ‘its’ type, which can be \( \text{Int}, \text{Char} \), or even a polymorphic type.

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

\[
\begin{align*}
(Ax) : & \quad \Gamma, x: \tau \vdash_{\text{ml}} x : \tau \\
(\rightarrow I) : & \quad \Gamma, x:A \vdash_{\text{ml}} E : B \quad \Gamma \vdash_{\text{ml}} \lambda x . E : A \rightarrow B \\
(let) : & \quad \Gamma \vdash_{\text{ml}} E_1 : \tau \quad \Gamma, x: \tau \vdash_{\text{ml}} E_2 : B \quad \Gamma \vdash_{\text{ml}} \text{let } x = E_1 \text{ in } E_2 : B \\
(\forall I) : & \quad \Gamma \vdash_{\text{ml}} E : \tau \quad \Gamma \vdash_{\text{ml}} \forall \varphi. \tau \quad x \notin \Gamma \\
(\forall E) : & \quad \Gamma \vdash_{\text{ml}} E : \forall \varphi. \tau \quad \Gamma \vdash_{\text{ml}} E[ \varphi / \varphi] \\
(\text{Const)} : & \quad \Gamma \vdash_{\text{ml}} c : \nu(c) \\
(\rightarrow E) : & \quad \Gamma, g:A \vdash_{\text{ml}} E : A \quad \Gamma \vdash_{\text{ml}} E_1 : A \\
(fix) : & \quad \Gamma, g:A \vdash_{\text{ml}} E : A \quad \Gamma \vdash_{\text{ml}} \text{fix } g . E : A \\
(V) : & \quad \Gamma, n\in N \vdash_{\text{ml}} \text{leq } n \vdash_{\text{ml}} \text{leq } n \\
(V E) : & \quad \Gamma \vdash_{\text{ml}} E : A[B/\varphi] \\
(V E) : & \quad \Gamma \vdash_{\text{ml}} E : A[B/\varphi] \\

\end{align*}
\]

The quantification of type variables is introduced in order to model the substitution operation on types that we have seen in previous sections; we can model the replacement of the type variable \( \varphi \) 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

\[
\begin{align*}
\emptyset \vdash_{\text{ml}} M : A \\
\emptyset \vdash_{\text{ml}} M : \forall \varphi. A \quad (\forall I) \\
\emptyset \vdash_{\text{ml}} M : A[B/\varphi] \quad (\forall E) \\
\end{align*}
\]

The \( \text{let}\)-construct corresponds in a way to the use of definitions in \( \Lambda^{\text{NR}} \); notice that we can represent \( n : N \vdash M \) by \( \text{let } n = \text{leq } n \text{ in } M \). But let is more general than that. First of all, a \( \text{let}\)-expression can occur at any point in the \( \text{ml}\)-term, not just on the outside, and, more significantly, \( N \) need not be a closed term. In \( \text{ml} \) it is possible to define a term that is partially polymorphic, i.e. has a type like \( \forall \varphi . A \), where \( A \) has also free type variables. Notice that, when applying rule \( (\forall I) \), we only need to check if the type variable we are trying to bind does not occur in the context; this can generate a derivation for \( \Gamma \vdash_{\text{ml}} M : \forall \varphi . A \), where the free type variables in \( A \) are those occurring in \( \Gamma \).

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

**Exercise 6.5** (Generation Lemma) i) If \( \Gamma \vdash_{\text{ml}} x : \sigma \), then there exists \( x: \tau \in \Gamma \) such that \( \tau \leq \sigma \).

ii) If \( \Gamma \vdash_{\text{ml}} E_1 E_2 : \sigma \), then there exist \( A, B \) such that \( \Gamma \vdash_{\text{ml}} E_1 : A \rightarrow B \), \( \Gamma \vdash_{\text{ml}} E_2 : A \), and \( \sigma = \forall \varphi_i. B \), with each \( \varphi_i \) not in \( \Gamma \).

iii) If \( \Gamma \vdash_{\text{ml}} \lambda x. E : \sigma \), then there exist \( A, B \) such that \( \Gamma, x:A \vdash_{\text{ml}} E : B \), and \( \sigma = \forall \varphi_i. A \rightarrow B \), with each \( \varphi_i \) not in \( \Gamma \).
iv) If $\Gamma \vdash_{\text{ML}} \text{fix } g.E : \sigma$, then there exists $A$ such that $\Gamma, g : A \vdash_{\text{ML}} E : A$, and $\sigma = \forall \varphi. A$, with each $\varphi_i$ not in $\Gamma$.

v) If $\Gamma \vdash_{\text{ML}} \text{let } x = E_1 \in E_2 : \sigma$, then there exists $A, \tau$ such that $\Gamma, x : \tau \vdash_{\text{ML}} E_1 : A$, and $\Gamma \vdash_{\text{ML}} E_2 : \tau$, and $\sigma = \forall \varphi. A$, with each $\varphi_i$ not in $\Gamma$.

Notice that we do not have that $\Gamma \vdash_{\text{ML}} E : \sigma$ and $\sigma \leq \tau$ imply $\Gamma \vdash_{\text{ML}} E : \tau$.

It is easy to show that the subject reduction property holds also for ML (see Exercise 6.13).

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 [30]. This system is obtained from Curry’s system by adding the type constructor ‘$\forall$’: if $\varphi$ 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 the generic type $\tau$ is used. Assume that $\tau = \forall \varphi. A$, and that in building the derivation for the statement $E_2 : B$, $\tau$ 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 $B$ such that $A_i = A[B/\varphi]$. 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:

\[
\frac{\Gamma \vdash_{\text{ML}} E_1 : A \quad \forall \varphi. A \vdash_{\text{ML}} x : \forall \varphi. A \quad (Ax)}{\Gamma, x : \forall \varphi. A \vdash_{\text{ML}} E_1 : A \quad (\forall I)}
\]

\[
\frac{\Gamma, x : \forall \varphi. A \vdash_{\text{ML}} E_1 : A \quad (Ax)}{\Gamma, x : \forall \varphi. A \vdash_{\text{ML}} E_1 : A \quad (\forall E)}
\]

\[
\frac{\Gamma, x : \forall \varphi. A \vdash_{\text{ML}} E_1 : A \quad (Ax)}{\Gamma, x : \forall \varphi. A \vdash_{\text{ML}} E_1 : A \quad (\forall E)}
\]

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 $\varphi$ does not occur in $\Gamma$), and we can type the contraction of the redex as follows (notice that then quantification is no longer used):

\[
\frac{\Gamma \vdash_{\text{ML}} E_1 : A[B/\varphi]}{\Gamma \vdash_{\text{ML}} E_2[E_1/x] : D}
\]

\[
\frac{\Gamma \vdash_{\text{ML}} E_1 : A[C/\varphi]}{\Gamma \vdash_{\text{ML}} E_2[E_1/x] : D}
\]

**Example 6.6** The program ‘$i = \lambda x.x : II$’ translates as ‘let $i = \lambda x.x$ in $ii$’ which we can type by:

\[
\frac{x : \varphi \vdash_{\text{ML}} x : \varphi}{(Ax)}
\]

\[
\frac{\emptyset \vdash_{\text{ML}} \lambda x.x : \varphi \rightarrow \varphi}{(\rightarrow I)}
\]

\[
\frac{\emptyset \vdash_{\text{ML}} \lambda x.x : \forall \varphi. \varphi \rightarrow \varphi \quad (\forall I)}{\emptyset \vdash_{\text{ML}} \lambda x.x : \forall \varphi. \varphi \rightarrow \varphi}
\]

\[
\frac{x : \varphi. \varphi \rightarrow \varphi \vdash_{\text{ML}} i : (A \rightarrow A) \rightarrow A \rightarrow A}{i : \forall \varphi. \varphi \rightarrow \varphi \vdash_{\text{ML}} i : (A \rightarrow A) \rightarrow A \rightarrow A \quad (Ax)}
\]

\[
\frac{i : \forall \varphi. \varphi \rightarrow \varphi \vdash_{\text{ML}} i : (A \rightarrow A) \rightarrow A \rightarrow A}{i : \forall \varphi. \varphi \rightarrow \varphi \vdash_{\text{ML}} i : A \rightarrow A \quad (\forall E)}
\]

\[
\frac{x : \varphi. \varphi \rightarrow \varphi \vdash_{\text{ML}} ii : A \rightarrow A}{i : \forall \varphi. \varphi \rightarrow \varphi \vdash_{\text{ML}} ii : A \rightarrow A \quad (Ax)}
\]

\[
\frac{\emptyset \vdash_{\text{ML}} \text{let } i = \lambda x.x \in ii : A \rightarrow A}{\emptyset \vdash_{\text{ML}} \text{let } i = \lambda x.x \in ii : A \rightarrow A \quad (\forall E)}
\]

As for rule (fix), remember that, to express recursion, we look for a solution to an equation like $F = N[F/x]$ which has as solution $F = Y(\lambda x.N)$. One way of dealing with this, and the
approach of [25], is to add the constant \( Y \) to the calculus as discussed in Example 5.4.

Instead, here we follow the approach of [26] 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 \( \text{fix } g \ x.E \) as general fixed-point constructor, which would correspond to our \( \text{fix } g.\lambda x.E \); the rule then is formulated as follows:

\[
(fix) : \quad \frac{\Gamma, g : A \rightarrow B, x : A \vdash E : B}{\Gamma \vdash_{\text{fix}} \text{fix } g \ x. E : A \rightarrow B}
\]

This is, for example, the approach of [48].

Another approach is the use of \( \text{letrec} \), a combination of \( \text{let} \) and \( \text{fix} \), of the form

\[
\text{letrec } g = \lambda x. E_1 \text{ in } E_2
\]

with derivation rule

\[
(\text{letrec}) : \quad \frac{\Gamma, g : B \rightarrow C, x : B \vdash E_1 : C \quad \Gamma, g : \tau \vdash_{\text{fix}} E_2 : A \quad (\tau = \Gamma(B \rightarrow C))}{\Gamma \vdash_{\text{letrec}} \text{letrec } g = \lambda x. E_1 \text{ in } E_2 : A}
\]

This construct \( \text{letrec } g = \lambda x. E_1 \text{ in } E_2 \) can be viewed as syntactic sugar for

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

### 6.2 Milner’s \( W \)

We will now define Milner’s algorithm \( W \). Notice that, different from the algorithms we considered above, \( W \) 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 \( \text{let}-\text{bound} \) variables: these might need to have a quantified type, which does not get constructed by \( W \); so, for the same reason, \( W \) passes the context as a parameter, which should have the correct type for variables to make the term typeable.

**Definition 6.7 (Milner’s Algorithm \( W \) [44])**

\[
W \Gamma x = \langle \text{id}, B \rangle
\]

where \( x : \forall \varphi. A \in \Gamma \)

\[
B = A[\varphi' / \varphi]
\]

all \( \varphi' \) = fresh

\[
W \Gamma (\lambda x. E) = \langle S_1, S_1(\varphi \rightarrow A) \rangle
\]

where \( \langle S_1, A \rangle = W (\Gamma, x : \varphi) E \)

\( \varphi = \text{fresh} \)

\[
W \Gamma (E_1 E_2) = \langle S_3 \circ S_2 \circ S_1, S_3 \varphi \rangle
\]

where \( \varphi = \text{fresh} \)

\( \langle S_1, A \rangle = W \Gamma E_1 \)

\( \langle S_2, B \rangle = W (S_1 \Gamma) E_2 \)

\( S_3 = \text{unify} (S_2 A) (B \rightarrow \varphi) \)

\[
W \Gamma (\text{let } x = E_1 \text{ in } E_2) = \langle S_2 \circ S_1, B \rangle
\]

where \( \langle S_1, A \rangle = W \Gamma E_1 \)

\( \langle S_2, B \rangle = W (S_1 \Gamma, x : S_1 \Gamma A) E_2 \)
\[ W \Gamma (\text{fix } g.E) = (S_2 \circ S_1, S_2 A) \]
where \( (S_1, A) = W (\Gamma, g; \varphi) E \)
\[
S_2 = \text{unify} (S_1 \varphi) A \\
\varphi = \text{fresh}
\]

Notice the use of \( S_1 \Gamma A \) in the case for let, where we add a quantified type for \( x \) to the context.

This system has several important properties:

- The system has a principal type property, in that, given \( \Gamma \) and \( E \), there exists a principal type, calculated by \( W \). 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, \( W \) satisfies:

**Theorem 6.8** 
Complete property of \( W \). If for a term \( E \) there are contexts \( \Gamma \) and \( \Gamma' \) and type \( A \), such that \( \Gamma' \) is an instance of \( \Gamma \) and \( \Gamma' \vdash_{\text{ml}} E : A \), then \( W \Gamma E = (S, B) \), and there is a substitution \( S' \)

such that \( \Gamma' = S'(S \Gamma) \) and \( S'(S B) \geq A \).

- Soundness of \( W \). For every term \( E \): if \( W \Gamma E = (S, A) \), then \( S \Gamma \vdash_{\text{ml}} E : A \).

**Example 6.9** To express addition in \( \text{ML} \), we can proceed as follows. We can define addition by:

\[
\text{Add} = \lambda xy. \text{cond} (\text{IsZero } x) y (\text{Succ} (\text{Add} (\text{Pred} x) y))
\]

We have seen in the first section that we can express \( \text{Succ}, \text{Pred}, \) and \( \text{IsZero} \) in the \( \lambda \)-calculus, and now know that we can express recursive definitions in \( \text{ML} \): so we can write

\[
\text{Add} = \text{fix } a. \lambda xy. \text{cond} (\text{IsZero } x) y (\text{Succ} (a (\text{Pred} x) y))
\]

Assuming we can type the added constructs as follows:

- \( \text{Succ} : \text{Num} \rightarrow \text{Num} \)
- \( \text{Pred} : \text{Num} \rightarrow \text{Num} \)
- \( \text{IsZero} : \text{Num} \rightarrow \text{Bool} \)
- \( \text{cond} : \forall \varphi. \text{Bool} \rightarrow \varphi \rightarrow \varphi \rightarrow \varphi \)

we can type the definition of addition as follows (where we write \( N \) for \( \text{Num} \), \( B \) for \( \text{Bool} \), and \( \Gamma \) for \( a : N \rightarrow N \rightarrow N, x : N, y : N \)):

Let

\[
D_1 = \\
\Gamma \vdash \text{cond} : \forall \varphi. B \rightarrow \varphi \rightarrow \varphi \rightarrow \varphi \quad (\text{Ass}) \\
\Gamma \vdash \text{IsZero} : N \rightarrow B \quad (\text{Ass}) \\
\Gamma \vdash x : N \quad (\text{Ax}) \\
\Gamma \vdash y : N \quad (\text{Ax})
\]

\[
\Gamma \vdash \text{cond} (\text{IsZero } x) : N \rightarrow N \rightarrow N \\
\Gamma \vdash \text{IsZero} x : B \quad (\rightarrow E) \\
\Gamma \vdash y : N \quad (\rightarrow E)
\]

\[
\Gamma \vdash \text{cond} (\text{IsZero } x) y : N \rightarrow N
\]

\[
D_2 = \\
\Gamma \vdash_{\text{ml}} a : N \rightarrow N \rightarrow N \quad (\text{Ax}) \\
\Gamma \vdash_{\text{ml}} \text{Pred} : N \rightarrow N \quad (\text{Ass}) \\
\Gamma \vdash_{\text{ml}} x : N \quad (\text{Ax}) \\
\Gamma \vdash_{\text{ml}} y : N \quad (\text{Ax})
\]

\[
\Gamma \vdash_{\text{ml}} a (\text{Pred } x) N \quad (\rightarrow E) \\
\Gamma \vdash_{\text{ml}} a (\text{Pred } x) y : N \quad (\rightarrow E)
\]

\[
\Gamma \vdash_{\text{ml}} \text{Succ} (a (\text{Pred } x) y) : N
\]

36
then we can construct:

\[
\begin{array}{c}
D_1 \\
\Gamma \vdash \text{cond} (\text{IsZero} \ x) \ y : N \rightarrow N \\
\hline
\Gamma \vdash (\text{IsZero} \ x) \ y \ (\text{Succ} \ (a \ (\text{Pred} \ y) \ y)) : N \hspace{1cm} (\rightarrow E)
\end{array}
\]

\[
\begin{array}{c}
D_2 \\
\Gamma \vdash \text{cond} (\text{IsZero} \ x) \ y : N \rightarrow N \\
\hline
\Gamma \vdash \text{cond} (\text{IsZero} \ x) \ y \ (\text{Succ} \ (a \ (\text{Pred} \ x) \ y)) : N \hspace{1cm} (\rightarrow I)
\end{array}
\]

6.3 Polymorphic recursion

In \cite{47} A. Mycroft defined a generalisation of Milner’s system (independently, a similar system was defined in \cite{36}). This generalisation is made to obtain more permissive types for recursively defined objects.

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

\[
\begin{align*}
\text{map} & = \lambda m. \text{cond} \ (\text{nil} \ l) \ (\text{nil} \ (m \ (\text{hd} \ l))) \ (\text{map} \ m \ (\text{tl} \ l)) \ ; \\
\text{squarelist} & = \lambda l. \text{map} \ (\lambda x. \text{mul} \ x \ x) \ l \\
& \quad : \text{squarelist} \ (\text{cons} \ 2 \ \text{nil})
\end{align*}
\]

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

\[
\begin{align*}
\text{let} \langle \text{map}, \text{squarelist} \rangle & = \text{fix} \ (m, s). (\lambda g l. \text{cond} \ (\text{nil} \ l) \ (\text{nil} \ (g \ (\text{hd} \ l))) \ (m \ (\text{tl} \ l))) , \\
& \quad \lambda l. (m (\lambda x. \text{mul} \ x \ x) \ l) \\
& \quad \text{in} \ \text{squarelist} \ (\text{cons} \ 2 \ \text{nil})
\end{align*}
\]

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

\[
\begin{align*}
\text{map} & :: (\text{num} \rightarrow \text{num}) \rightarrow [\text{num}] \rightarrow [\text{num}] \\
\text{squarelist} & :: [\text{num}] \rightarrow [\text{num}]
\end{align*}
\]

while the definition of map alone would yield the type:

\[
\begin{align*}
\text{map} & :: \forall \varphi_1 \varphi_2. (\varphi_1 \rightarrow \varphi_2) \rightarrow [\varphi_1] \rightarrow [\varphi_2].
\end{align*}
\]

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

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

\[
\begin{align*}
\text{let} \ \text{map} & = \text{fix} m g l. \text{cond} \ (\text{nil} \ l) \ (\text{nil} \ (g \ (\text{hd} \ l))) \ (m \ (\text{tl} \ l))) , \\
& \quad \text{in} \ (\text{let} \ \text{squarelist} = (\lambda l. (\text{map} \ (\lambda x. \text{mul} \ x \ x) \ l)) \\
& \quad \text{in} \ (\text{squarelist} \ (\text{cons} \ 2 \ \text{nil})))
\end{align*}
\]
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 6.10** ([47]) *Mycroft type assignment* is defined by replacing rule (fix) by:

\[
\text{fix} : \frac{\Gamma, g : \tau \vdash_{\text{Myc}} E : \tau}{\Gamma \vdash_{\text{Myc}} \text{fix } g.E : \tau}
\]

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

\[
\text{name} : \frac{\Gamma ; E, name : A 
\vdash \text{name} : SA}{\Gamma \vdash_{\text{Myc}} \text{name} : SA}
\]

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

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

Mycroft’s system has the following properties:

- Like in Milner’s system, in this system polymorphism can be modelled.
- Type assignment in this system is undecidable, as shown by A.J. Kfoury, J. Tiuryn and P. Urzyczyn in [37].

For $\Lambda^{NR}$, Mycroft’s approach results in the following implementation:

**Definition 6.11** *(Mycroft)*

\[
\begin{align*}
\text{Mycroft } \Gamma & \vdash x \ E \quad = \langle \text{Id}_{S}, \Gamma \ x \rangle \\
\text{Mycroft } \Gamma & \vdash n \ E \quad = \langle \text{Id}_{S}, \text{FreshInstance } (E \ n) \rangle \\
\text{Mycroft } \Gamma & \vdash (E_{1}E_{2}) \ E \quad = \langle S_{1} \circ S_{2} \circ S_{3}, S_{1} \phi \rangle \\
& \quad \text{where } \phi \quad = \text{fresh} \\
& \quad \langle S_{3}, C \rangle \quad = \text{Mycroft } \Gamma \ E_{1} \ E \\
& \quad \langle S_{2}, A \rangle \quad = \text{Mycroft } (S_{3} \Gamma) \ E_{2} \ E \\
& \quad S_{1} \quad = \text{unify } (S_{2} C) \ A \rightarrow \phi \\
\text{Mycroft } \Gamma & \vdash (\lambda x. \ E) \ E \quad = \langle S_{1}, S_{1} (\phi \rightarrow C) \rangle \\
& \quad \text{where } \phi \quad = \text{fresh} \\
& \quad \langle S_{1}, C \rangle \quad = \text{Mycroft } (\Gamma, x : \phi) \ E \ E \\
\text{CheckType } \langle e : E \rangle \ E \quad = \ A, \\
& \quad \text{where } \langle S, A \rangle = \text{Mycroft } \odot E \ E \\
\text{CheckType } \langle n = E_{1} ; \text{Defs} : E_{2} \rangle \ E \quad = \text{CheckType } \langle \text{Defs} : E_{2} \rangle \ E, \\
& \quad \text{if } A = B \\
& \quad \text{where } A \quad = \ E \ n \\
& \quad \langle S, B \rangle \quad = \text{Mycroft } \odot E_{1} \ E
\end{align*}
\]

Notice that, in this approach, 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.

### 6.4 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
more general types to terms that are typeable in Milner’s system. For example, the statement
terms that can be typed in Mycroft’s system larger than in Milner’s, it is also possible to assign
Mycroft suggests, type assignment in this system is undecidable. And not only is the set of
is a derivable statement (let \( \Gamma = g: \forall \varphi_1 \varphi_2.(\varphi_1 \rightarrow \varphi_2) \)):

\[
\Gamma,a:q_3 \rightarrow q_4, b:B \vdash a:q_3 \rightarrow q_4 \quad (Ax) \\
\Gamma, b:B \vdash \lambda b.a:(q_3 \rightarrow q_4) \rightarrow B \rightarrow q_3 \rightarrow q_4 \quad (\rightarrow I) \\
\Gamma \vdash g:(C \rightarrow C) \rightarrow q_3 \rightarrow q_4 \quad (Ax) \\
\Gamma \vdash g \lambda c.c:q_3 \rightarrow q_4 \quad (\rightarrow E) \\
\Gamma \vdash (\lambda b.a)(g \lambda c.c)(g \lambda d.e.d):q_3 \rightarrow q_4 \quad (\rightarrow E) \\
\Gamma \vdash ((\lambda b.a)(g \lambda c.c)(g \lambda d.e.d)):\forall \varphi_2.(\varphi_1 \rightarrow \varphi_2) \quad (\forall I) \\
\Gamma \vdash ((\lambda b.a)(g \lambda c.c)(g \lambda d.e.d)) : \forall \varphi_1 \varphi_2. (\varphi_1 \rightarrow \varphi_2) \quad (\forall I) \\
\Box \vdash \text{fix } g. ((\lambda b.a)(g \lambda c.c)(g \lambda d.e.d)) : \forall \varphi_1 \varphi_2. (\varphi_1 \rightarrow \varphi_2) \quad (\text{fix})
\]

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

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

\[
\Box \vdash \text{Myc } \text{r. } (\lambda x y. (r (r y (\lambda y a)) x)) : \forall \varphi_1 \varphi_2 \varphi_3. (\varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3)
\]

is derivable in Mycroft’s system (where \( \Gamma = r : \forall \varphi_1 \varphi_2 \varphi_3. (\varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3), x : \varphi_4, y : \varphi_5 \)),

\[
\Gamma \vdash r:q_5 \rightarrow (A \rightarrow B \rightarrow A) \rightarrow q_7 \quad (Ax) \\
\Gamma \vdash y:q_5 \quad (Ax) \\
\Gamma \vdash r:q_2 \rightarrow q_4 \rightarrow q_6 \quad (Ax) \\
\Gamma \vdash r r y (\lambda a b.a) : q_7 \quad (\rightarrow E) \\
\Gamma \vdash y:q_4 \quad (Ax) \\
\Gamma \vdash r (r y (\lambda a b.a)) : q_4 \rightarrow q_6 \quad (Ax) \\
\Gamma \vdash r (r y (\lambda a b.a)) x : q_6 \quad (\rightarrow E) \\
\Gamma \vdash r \forall \varphi_1 \varphi_2 \varphi_3. (\varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3) x : q_4 \rightarrow q_5 \rightarrow q_6 \quad (\forall I) \\
\Gamma \vdash r \forall \varphi_1 \varphi_2 \varphi_3. (\varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3) \lambda y. (r y (\lambda a b.a)) x : q_4 \rightarrow q_5 \rightarrow q_6 \quad (\forall I) \\
\Gamma \vdash r \forall \varphi_1 \varphi_2 \varphi_3. (\varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3) \lambda y. (r y (\lambda a b.a)) x : q_6 (q_4 \rightarrow q_5 \rightarrow q_6) \quad (\forall I) \\
\Box \vdash \text{fix } r. (\lambda y. (r y (\lambda a b.a)) x) : \forall \varphi_4 \varphi_5 \varphi_6 (q_4 \rightarrow q_5 \rightarrow q_6) \quad (\text{fix})
\]

Notice that \( \forall \varphi_4 \varphi_5 \varphi_6. (q_4 \rightarrow q_5 \rightarrow q_6) = \forall \varphi_1 \varphi_2 \varphi_3. (\varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3) \), so we can apply rule (fix);
moreover \( A \) and \( B \) are irrelevant for this construction.

\( R \) is also typeable in Milner’s system, as shown in Exercise 6.15.

**Exercises**

**Exercise 6.12** If \( A \leq B \), then there exists a substitution \( S \) such that \( SA = B \).
Exercise 6.13 If $\Gamma \vdash_{\text{ML}} M : A$, and $M \rightarrow_{\text{ML}} N$, then $\Gamma \vdash_{\text{ML}} N : A$.

Exercise 6.14 Find an ML-term for multiplication, and type it; you can abbreviate the derivation from Exercise 6.9.

Exercise 6.15 Show $\vdash_{\text{ML}} R : \forall \phi_4 \phi_5. ((\phi_4 \rightarrow \phi_5 \rightarrow \phi_4) \rightarrow (\phi_4 \rightarrow \phi_5 \rightarrow \phi_4) \rightarrow \phi_4 \rightarrow \phi_4 \rightarrow \phi_4)$.

7 Pattern matching: term rewriting

The notion of reduction we will study in this section is that of term rewriting [39, 40], 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.

7.1 Term Rewriting Systems

Term rewriting systems can be seen as an extension of the Lambda 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 7.1 (Syntax) i) An alphabet or signature $\Sigma$ consists of a countable, infinite set $X$ of variables $x_1, x_2, x_3, \ldots$ (or $x, y, z, x', y', \ldots$), a non-empty set $F$ of function symbols $F, G, \ldots$, each with a fixed arity.

ii) The set $T(F, X)$ 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 $\lambda$-calculus. We write $t^R$ for the result of applying the replacement $R$ to $t$.

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

Definition 7.2 (Reduction) i) A rewrite rule is a pair $(l, r)$ of terms. Often, a rewrite rule will get a name, e.g. $r$, and we write

$$r : l \rightarrow r$$

Two conditions are imposed:

a) $l = F t_1 \cdots t_n$, for some $F \in F$ with arity $n$ and $t_1, \ldots, t_n \in T(F, X)$, and

b) $\text{fv}(r) \subseteq \text{fv}(l)$.

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 $x \in \text{fv}(t_j)$.

iii) A rewrite rule $l \rightarrow r$ determines a set of rewrites $l^R \rightarrow r^R$ 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 $C[\ ]$; this gives rise to rewrite steps $C[t] \rightarrow C[t']$. Concatenating rewrite steps we have rewrite sequences $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots$. If $t_0 \rightarrow \cdots \rightarrow t_n (n \geq 0)$ we also write $t_0 \rightarrow\rightarrow t_n$. 

40
Notice that, if \( l \rightarrow r \) 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 soon above, Combinatory Logic is a special TRS.

**Definition 7.3** A **Term Rewriting System** (TRS) is defined by a pair \((\Sigma, R)\) of an alphabet \(\Sigma\) and a set \(R\) of rewrite rules.

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

**Definition 7.4** In a rewrite rule \( r : F t_1 \cdots t_n \rightarrow r \), \( F \) 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 \( F \), and \( Q \in F \) is called a constructor if \( Q \) is not a defined symbol.

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

**Example 7.5** 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.

```plaintext
append nil l \rightarrow l
append (cons x l) l' \rightarrow cons x (append l l')
append (append l l') l'' \rightarrow append l (append l' l'')
map f nil \rightarrow nil
map f (cons y l) \rightarrow cons (f y) (map f l)
```

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 \( F t_1 \cdots t_n \rightarrow r \), 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 \( F \), as long as those are not function symbols, i.e. are constructors.

This division is not used in TRS: the symbol ‘`append`’ appears in the third rule in both function and constructor position, so, in TRS, the distinction between the notions of function symbol and constructor is lost.

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

```plaintext
In-left (Pair x y) \rightarrow x
In-right (Pair x y) \rightarrow y
Pair (In-left x) (In-right x) \rightarrow x
```

A difficulty with this TRS is that it forms Klop’s famous ‘Surjective Pairing’ example [38]; this function cannot be expressed in the Lambda Calculus because when added to the Lambda Calculus, the Church-Rosser property no longer holds. This implies that, although both the Lambda 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 Lambda Calculus, so a full-purpose translation (interpretation) of TRS in the Lambda Calculus is not feasible.
7.2 Type assignment for TRS

We will now set up a notion of type assignment for TRS, as defined and studied in [10, 7, 8]. For the same reasons as before we use an environment providing a type for each function symbol. From it we can derive many types to be used for different occurrences of the symbol in a term, all of them ‘consistent’ with the type provided by the environment; an environment functions as in the Milner and Mycroft algorithms.

Definition 7.6 (Environment) An environment is a mapping \( E : \mathcal{F} \to \mathcal{T}_c \).

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

Definition 7.7 A TRS-context is a set of statements with variables (not necessarily distinct) as subjects.

Notice that this generalisation would allow for \( \mathbf{xx} \) to be typeable.

Definition 7.8 (Type Assignment on Terms) Type assignment (with respect to an environment \( E \)) is defined by the following natural deduction system. Note the use of a substitution in rule \((F)\).

\[
\begin{align*}
(Ax) & : \Gamma ; x : A; E \vdash x : A \\
(F) & : \Gamma ; E; F : A \vdash F : SA \\
(\to E) & : \Gamma ; E \vdash t_1 : A \to B \quad \Gamma ; E \vdash t_2 : A \\
& \quad \Gamma ; E \vdash t_1 t_2 : B
\end{align*}
\]

As before, the use of an environment in rule \((F)\) 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 7.3.
- Subject reduction. This will be proven in Section 7.4.
- It is, in general, not possible to prove a strong normalisation result: take \( t \) that is typeable, and the rule \( t \to t \), 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

\[
F \overline{C[\mathbf{x}]} \to C'(F \overline{C_1[\mathbf{x}]}) \ldots (F \overline{C_m[\mathbf{x}]},
\]

where, for every \( j \in m, \overline{C_j[\mathbf{x}]} \) is a strict subterm of \( \overline{C[\mathbf{x}]} \), so \( F \) is only called recursively on terms that are substructures of its initial arguments (see [8] for details); this scheme generalizes primitive recursion.

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

7.3 The principal pair for a term

In this subsection, the principal pair for a term \( t \) with respect to the environment \( E \) is defined, consisting of context \( \Pi \) and type \( P \), using Robinson’s unification algorithm \text{unify}. In the following, we will show that for every typeable term, this is a legal pair and is indeed the most general one.

Definition 7.9 We define the notion \( \text{pp} \ E t = \langle \Pi, P \rangle \) inductively by:
Notice that, since we allow a context to contain more than one statement for each variable, we do not require \( \Pi_1 \) and \( \Pi_2 \) to agree via the unification of contexts.

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

**Lemma 7.10 (Soundness of Substitution)** If \( \Gamma; E \vdash t : A \), then \( S \Gamma; E \vdash t : S A \), for every \( S \).

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

**Theorem 7.11 (Completeness of Substitution)** If \( \Gamma; E \vdash t : A \), then there are \( \Pi, P \), and a substitution \( S \) such that: \( pp \ E t = (\Pi, P) \), and \( S \Pi \subseteq \Gamma, S P = A \).

**Proof:** By induction on the structure of \( t \).

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

ii) \( t \equiv t_1 t_2 \), and there is a \( B \in T_c \) such that \( \Gamma; E \vdash t_1 : B \rightarrow A \), and \( \Gamma; E \vdash t_2 : B \). By induction, for \( i = 1, 2 \), there are \( \Pi_i, P_i \), and a substitution \( S_i \) such that

\[
pp \ E t_i = \langle \Pi_i, P_i \rangle, \quad S_1 \Pi_1 \subseteq \Gamma, \quad S_2 \Pi_2 \subseteq \Gamma, \quad S_1 P_1 = B \rightarrow A, \quad \text{and} \quad S_2 P_2 = B.
\]

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

### 7.4 Subject Reduction

By Definition 7.2, if a term \( t \) is rewritten to the term \( t' \) using the rewrite rule \( l \rightarrow r \), there is a subterm \( t_0 \) of \( t \), and a replacement \( R \), such that \( t^R = t_0 \), and \( t' \) is obtained by replacing \( t_0 \) by \( r^R \).

To guarantee the subject reduction property, we should accept only those rewrite rules \( l \rightarrow r \), that satisfy:

\[
\text{For all replacements } R, \text{ contexts } \Gamma \text{ and types } A: \text{ if } \Gamma; E \vdash l^R : A, \text{ then } \Gamma; E \vdash r^R : A.
\]

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 7.12** i) We say that \( l \rightarrow r \in R \) with defined symbol \( F \) is typeable with respect to \( E \), if there are \( \Pi, P \) such that \( (\Pi, P) \) is the principal pair for \( l, \Pi; E \vdash r : P \), and such that the leftmost occurrence of \( F \) in \( \Pi; E \vdash l : P \) is typed with \( E(F) \).

ii) We say that a TRS is typeable with respect to \( E \), if all \( r \in R \) are.

Notice that the notion \( pp \ E t \) is defined independently from the definition of typeable rewrite rules. Also, since only ‘the leftmost occurrence of \( F \) in \( \Pi; E \vdash l : P \) is typed with \( E(F) \)’, this notion of type assignment uses Mycroft’s solution for recursion; using Milner’s, the definition would have defined ‘all occurrences of \( F \) in \( \Pi; E \vdash l : P \) and \( \Pi; E \vdash r : P \) are typed with \( E(F) \)’.
In the following lemma we show that if \( F \) is the defined symbol of a rewrite rule, then the type \( \mathcal{E} F \) dictates not only the type for the left and right-hand side of that rule, but also the principal type for the left-hand side.

**Lemma 7.13** If \( F \) is the defined symbol of the typeable rewrite rule \( F t_1 \cdots t_n \to r \), then there are contexts \( \Pi, \Gamma \), and types \( A_i \ (i \in \mathbb{N}) \) and \( A \) such that
\[
\mathcal{E} F = A_1 \to \cdots \to A_n \to A,
\]
\[
pp \mathcal{E} I = (\Pi, A),
\]
\[
\Pi; \mathcal{E} \vdash t_i : A_i
\]
\[
\Gamma; \mathcal{E} \vdash I : A, \text{ and}
\]
\[
\Gamma; \mathcal{E} \vdash r : A.
\]

**Proof:** Easy, using Theorem 7.11 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 7.14** 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 \( \mathcal{E} S = (\varphi_1 \to \varphi_2 \to \varphi_3) \to (\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_3 \)
\[
\mathcal{E} K = \varphi_4 \to \varphi_5 \to \varphi_4
\]
\[
\mathcal{E} I = \varphi_6 \to \varphi_6
\]
\[
\mathcal{E} M = ((\varphi_7 \to \varphi_8) \to \varphi_9) \to (\varphi_7 \to \varphi_8) \to \varphi_8.
\]

To obtain \( pp \mathcal{E}(M(S\ x\ y)) \), we assign types to nodes in the tree in the following way. Let
\[
A = ((\varphi_1 \to \varphi_2) \to \varphi_4 \to \varphi_3) \to ((\varphi_1 \to \varphi_2) \to \varphi_4) \to (\varphi_1 \to \varphi_2) \to \varphi_3, \text{ and}
\]
\[
\Gamma = x:(\varphi_1 \to \varphi_2) \to \varphi_4 \to \varphi_3, \ y:(\varphi_1 \to \varphi_2) \to \varphi_4
\]
\[
\begin{array}{c}
\Gamma; \mathcal{E} \vdash S : A \\
\Gamma; \mathcal{E} \vdash x : (\varphi_1 \to \varphi_2) \to \varphi_4 \to \varphi_3 \\
\Gamma; \mathcal{E} \vdash S x : ((\varphi_1 \to \varphi_2) \to \varphi_4) \to (\varphi_1 \to \varphi_2) \to \varphi_3 \\
\Gamma; \mathcal{E} \vdash y : (\varphi_1 \to \varphi_2) \to \varphi_4
\end{array}
\]
\[
\begin{array}{c}
\Gamma; \mathcal{E} \vdash S x y : (\varphi_1 \to \varphi_2) \to \varphi_3 \\
\Gamma; \mathcal{E} \vdash M : ((\varphi_1 \to \varphi_2) \to \varphi_3) \to (\varphi_1 \to \varphi_2) \to \varphi_2
\end{array}
\]

If the right-hand side term of the rewrite rule should be typed with \( (\varphi_1 \to \varphi_2) \to \varphi_1 \), where
\[
B = ((\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_2) \to ((\varphi_1 \to \varphi_2) \to \varphi_1) \to (\varphi_1 \to \varphi_2) \to \varphi_2, \text{ and}
\]
\[
\Gamma' = y : (\varphi_1 \to \varphi_2) \to \varphi_1
\]
\[
\Gamma'; \mathcal{E} \vdash S : B \\
\Gamma'; \mathcal{E} \vdash I : (\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_2
\]
\[
\Gamma'; \mathcal{E} \vdash S I : ((\varphi_1 \to \varphi_2) \to \varphi_1) \to (\varphi_1 \to \varphi_2) \to \varphi_2 \\
\Gamma'; \mathcal{E} \vdash y : (\varphi_1 \to \varphi_2) \to \varphi_1
\]
\[
\Gamma'; \mathcal{E} \vdash S I y : (\varphi_1 \to \varphi_2) \to \varphi_2
\]

Take the term \( M(S\ K\ I) \), which rewrites to \( S\ I\ I \). Although the first term is typeable, with
Let for every replacement $R$ and type $A$

By induction on the structure of $(S \rightarrow M) : (C \rightarrow C) \rightarrow C \rightarrow C$

If $Γ ; E \vdash t : A$, and $R$ is a replacement and $Γ'$ a context such that for every statement $x : C ∈ Γ$: $Γ' ; E \vdash x^R : C$, then $Γ' ; E \vdash t^R : A$.

Proof: By induction on the structure of $t$.

i) $t ≡ x$, then $II = x : φ$, and $P = φ$. Take $S = (φ → A)$. By assumption, $Γ ; E \vdash x^R : A$, so for every $x : C ∈ Γ$ we have $Γ ; E \vdash x^R : SC$.

b) $t ≡ F$, then $II = ∅$, and $P = E F$. By rule $(F)$ there exists a substitution $S₀$ such that $A = S₀ (E F)$, and we have $Γ ; E \vdash t^R : A$ for every context $Γ$.

c) $t ≡ t₁ t₂$. Let $φ$ be a type-variable not occurring in any other type. If $pp E t₁ = (Π₂ P₂)$, then for $i = 1, 2$, there are $⟨Πᵢ Pᵢ⟩$ (disjoint), such that $pp E tᵢ = (Πᵢ Pᵢ)$. By induction, for $i = 1, 2$, there is a substitution $Sᵢ$ such that $Sᵢ Pᵢ = Aᵢ$, and, for every $x : Aᵢ ∈ Πᵢ$, $Γ ; E \vdash x^R : Sᵢ Aᵢ$. Notice that $S₁$ and $S₂$ do not interfere in that they are defined on separate sets of type variables. Take $S' = S₂ S₁ (φ → A)$, then, for every $x : A' ∈ Π₁ ∪ Π₂$, $Γ ; E \vdash x^R : S' A'$, and $S' φ = A$. By property 2.13 there are substitutions $S$ and $S₈$ such that

Then, for every $x : B' ∈ S₈ (Π₁ ∪ Π₂)$, $Γ ; E \vdash x^R : S B'$, and $S (S₈ φ) = A$.

ii) $t ≡ x F$. Trivial.

b) $t ≡ t₁ t₂$. Then there exists $B$, such that $Γ ; E \vdash t₁ : B → A$ and $Γ ; E \vdash t₂ : B$. By induction, $Γ' ; E \vdash t₁^R : B → A$ and $Γ' ; E \vdash t₂^R : B$. So, by $(→ E)$, we obtain $Γ' ; E \vdash (t₁ t₂)^R : A$.

Theorem 7.16 (Subject Reduction Theorem) If $Γ ; E \vdash t : A$ and $t → t'$, then $Γ ; E \vdash t' : A$.

Proof: Let $l → r$ be the typeable rewrite rule applied in the rewrite step $t → t'$. We will prove that for every replacement $R$ and type $A$, if $Γ ; E \vdash t^R : A$, then $Γ ; E \vdash r^R : A$, which proves the
theorem.

Since \( r \) is typeable, there are \( \Pi, P \) 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 \( \Gamma; \mathcal{E} \vdash l^R : \mathcal{A} \). By Lemma 7.15(i) there is a \( \Gamma' \) such that for every \( x : C \in \Gamma' \), \( \Gamma' ; \mathcal{E} \vdash x^R : C \), and \( \Gamma' ; \mathcal{E} \vdash l : \mathcal{A} \). Since \( \langle \Pi, P \rangle \) is a principal pair for \( l \) with respect to \( \mathcal{E} \), by Definition 7.9 there is a substitution \( S \) such that \( S \langle \Pi, P \rangle = \langle \Gamma', \mathcal{A} \rangle \).

Since \( \Pi; \mathcal{E} \vdash r : P \), by Theorem 7.10 also \( \Gamma'; \mathcal{E} \vdash r : \mathcal{A} \). Then by Lemma 7.15(ii) \( \Gamma'; \mathcal{E} \vdash r^R : \mathcal{A} \). □

7.5 A type check algorithm for TRSs

In this section we present a type check algorithm, as first presented in [10], that, when applied to a TRS and an environment determines whether this TRS is typeable with respect to the environment.

The goal of the type check algorithm presented below is to determine whether a type assignment can be constructed such that all the conditions of Definitions 7.8 and 7.12 are satisfied. The main function of the algorithm, called TypeTRS, expects a TRS as well as an environment as parameters. It returns a boolean that indicates whether the construction of the type assignment was successful.

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

**Theorem 7.17**

i) If \( t \) is typeable with respect to \( \mathcal{E} \), then TypeTerm \( t \mathcal{E} \) returns \( \langle \Gamma, A \rangle \).

ii) If TypeTerm \( t \mathcal{E} \) returns the pair \( \langle \Gamma, A \rangle \), then \( \langle \Gamma, A \rangle = \langle \Gamma', A \rangle \).

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

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

The algorithm does not perform any consistency check on its input so it assumes the input to be correct according to Definitions 7.1 and 7.2. Moreover, all possible error messages and error handling cases are omitted, and the algorithm TypeRules returns only true for rewrite systems that are typeable. It could easily be extended to an algorithm that rejects untypeable rewrite rules. Notice that, below, a TRS is a pair of rules and term; as in \( \Lambda \mathcal{N} \) and \( \Lambda \mathcal{NR} \), 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.

\[
\text{TypeTerm } x \quad \mathcal{E} \rightarrow \langle x : \varphi, \varphi \rangle
\]

where \( \varphi = \text{fresh} \)

\[
\text{TypeTerm } t_1 t_2 \quad \mathcal{E} \rightarrow S (\Gamma_1 \cup \Gamma_2, \varphi)
\]

where \( \varphi = \text{fresh} \)

\[
\langle \Gamma_1, B \rangle = \text{TypeTerm } t_1 \mathcal{E} \\
\langle \Gamma_2, A \rangle = \text{TypeTerm } t_2 \mathcal{E} \\
S = \text{unify } B A \rightarrow \varphi
\]

\[
\text{TypeTerm } F \quad \mathcal{E} \rightarrow \text{Freeze}(\mathcal{E} F), \quad \text{if this is defining occurrence of } F
\]

\[
\text{FreezInstance}(\mathcal{E} F), \quad \text{otherwise}
\]

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

TypeRule takes care of checking the safeness constraint as given in Definition 7.12, 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.

\[
\text{TypeRule} \ (l \rightarrow r) \ \mathcal{E} \rightarrow (S_2(S_1 \Gamma_l)) = \Gamma_l
\]

where
\[
S_2 = \text{UnifyContexts} (S_1 \Gamma_l) (S_1 \Gamma_r),
\]
\[
S_1 = \text{unify} \ A \ B,
\]
\[
\langle \Gamma_l, A \rangle = \text{TypeTerm} \ l \ \mathcal{E},
\]
\[
\langle \Gamma_r, B \rangle = \text{TypeTerm} \ r \ \mathcal{E}
\]

and the procedure that type checks the program:

\[
\text{TypeTRS} \ \langle R : t \rangle \ \mathcal{E} \rightarrow \text{TypeTerm} \ t \ \mathcal{E}, \text{if TypeRules} \ R \ \mathcal{E}
\]

Exercises

Exercise 7.18 Define

\[
\begin{align*}
1 \ x \ &\rightarrow x \\
K \ x \ y \ &\rightarrow x \\
B \ x \ y \ z \ &\rightarrow x(yz) \\
C \ x \ y \ z \ &\rightarrow xzy \\
S \ x \ y \ z \ &\rightarrow xz(yz)
\end{align*}
\]

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

Exercise 7.19 Check that these rules given in Section 3.2 are admissible, and check if they introduce any conflict with respect to types.

* Exercise 7.20 (Soundness of Substitution) If \( \Gamma; \mathcal{E} \vdash t : A \), then \( S \Gamma; \mathcal{E} \vdash t : S \ A \), for every \( S \).

* Exercise 7.21 (Soundness of \( pp \) \( \mathcal{E} \)) Verify that \( pp \ \mathcal{E} t = \langle \Pi, P \rangle \) implies \( \Pi; \mathcal{E} \vdash t : P \).

8 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.\(^2\)

8.1 Data structures

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

**Definition 8.1** The grammar of types is extended as follows:

\(^2\) This section is in part based on [49]
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 8.2 (Pairing)** We extend the calculus with the following constructors

\[
E ::= \ldots \mid \langle E_1, E_2 \rangle \mid \text{left}(E) \mid \text{right}(E)
\]

with their type assignment rules:

\[
\frac{\Gamma \vdash E_1 : A \quad \Gamma \vdash E_2 : B}{\Gamma \vdash \langle E_1, E_2 \rangle : A \times B}
\]

\[
\frac{\Gamma \vdash E : A \times B}{\Gamma \vdash \text{left}(E) : A}
\]

\[
\frac{\Gamma \vdash E : A \times B}{\Gamma \vdash \text{right}(E) : B}
\]

The reduction rules that come with these constructs are:

\[
\text{left}(\langle E_1, E_2 \rangle) \rightarrow E_1
\]

\[
\text{right}(\langle E_1, E_2 \rangle) \rightarrow E_2
\]

Notice that these rules are expressed through pattern matching.

We could be tempted to add the rule

\[
\langle \text{left}(E), \text{right}(E) \rangle \rightarrow E
\]

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

**Definition 8.3 (Disjoint Union)** We extend the calculus with the following constants

\[
E ::= \ldots \mid \text{case}(E_1, E_2, E_3) \mid \text{inj-}l(E) \mid \text{inj-}r(E)
\]

with their type assignment rules:

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

\[
\frac{\Gamma \vdash E : A}{\Gamma \vdash \text{inj-}l(E) : A + B}
\]

\[
\frac{\Gamma \vdash E : B}{\Gamma \vdash \text{inj-}r(E) : A + B}
\]

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 constants are:

\[
\text{case}(\text{inj-}l(E_1), E_2, E_3) \rightarrow E_2 \ E_1
\]

\[
\text{case}(\text{inj-}r(E_1), E_2, E_3) \rightarrow E_3 \ E_1
\]

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

### 8.2 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 $E : \text{unit} \rightarrow A$. In the functional programming languages Haskell [33], and Clean [14], the unit type is called $(\cdot)$ and its only value is also $(\cdot)$, reflecting the 0-tuple interpretation. In SML (Standard ML [31, 46]), the type is called unit but the value is written as $(\cdot)$. Using this approach here, we extend the syntax with $(\cdot)$, the type language with ‘unit’ and add the rule

\[
(unit) : \Gamma \vdash (\cdot) : \text{unit}
\]

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

\[
A = \text{unit} + (B \times A);
\]

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 $A = \text{unit} + (B \times A)$; remark that we cannot solve this without such an extension.

**Definition 8.4 (Recursive types)** The grammar of types is extended with:

\[
A, B = \cdots \mid X \mid \mu X.A
\]

We consider types $\mu$-equal when we can transform one into the other via a number of steps like

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

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

\[
\mu X.\text{unit} + (B \times X)
\]

because $A = \mu X.\text{unit} + (B \times X)$

\[
= \mu (\text{unit} + (B \times X)) [\mu X.\text{unit} + (B \times X)/X]
\]

\[
= \text{unit} + (B \times (\mu X.\text{unit} + (B \times X))) = \text{unit} + (B \times A)
\]

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 unfoldings 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.
8.3 The equi-recursive approach

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

Definition 8.5 (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:

\[
\frac{\Gamma \vdash E : A}{\Gamma \vdash E : B} \quad (A =_\mu B)
\]

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 8.6 A term now has a \([B]\) type if either it is of the shape \(\text{inj-1}()\) or \(\text{inj-}\langle a,b \rangle\):

\[
\frac{\Gamma \vdash () : \text{unit} \quad (\text{unit})}{\Gamma \vdash \text{inj-1}() : \text{unit} + (B \times [B]) \quad (\text{inj-1})}
\]

\[
\frac{\Gamma \vdash a : B \quad \Gamma \vdash b : [B] \quad (\text{Pair})}{\Gamma \vdash \langle a,b \rangle : B \times [B] \quad (\text{inj-r})}
\]

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

\[
\text{LL} = \text{fix } \lambda l \cdot \text{case } (\text{list}, \lambda x.0, \lambda x. + 1 \ (\text{ll}(\text{right} \ x)))
\]

Notice that now

\[
\langle \text{fix } \lambda l \cdot \text{case } (\text{list}, \lambda x.0, \lambda x. + 1 \ (\text{ll}(\text{right} \ x))) \rangle \ (\text{inj-r}(a,b)) \rightarrow
\]

\[
\langle \lambda x. + 1 \ (\text{ll}(\text{right} \ x)) \rangle \ (\text{inj-r}(a,b)) \rightarrow
\]

\[
+ 1 \ (\text{ll} \ b)
\]

Using \( I \) for the type for numbers, we can construct the following derivation (hiding obsolete statements in contexts) for the term above:

\[
\frac{\vdash + : I + I \rightarrow I}{\vdash 1 : I \rightarrow I \quad \Gamma : [\varphi] \rightarrow \Gamma : [\varphi] \rightarrow I \quad \frac{\vdash \Gamma : [\varphi] \rightarrow \Gamma : [\varphi] \rightarrow I \quad \vdash x : \varphi \times [\varphi] \rightarrow x \times \varphi \rightarrow \varphi}{\vdash x : \varphi \times [\varphi] \rightarrow \varphi \rightarrow x \rightarrow \varphi}}{\vdash 1 : I \rightarrow I \quad \vdash + : I + I \rightarrow I \quad \vdash \text{list} : [\varphi] \rightarrow \text{list} : [\varphi] \quad \vdash \text{x : unit} \rightarrow 0 : I \quad \vdash \lambda x.0 : \text{unit} \rightarrow I \quad \frac{\vdash a : \varphi \rightarrow b : \varphi}{\vdash \langle a,b \rangle : \varphi \times [\varphi] \rightarrow \varphi \times \varphi \rightarrow \varphi \rightarrow \langle \varphi, \varphi \rangle \rightarrow I \quad \frac{\vdash \text{fix } \lambda l \cdot \text{case } (\text{list}, \lambda x.0, \lambda x. + 1 \ (\text{ll}(\text{right} \ x))) \rightarrow I}{\vdash \text{fix } \lambda l \cdot \text{case } (\text{list}, \lambda x.0, \lambda x. + 1 \ (\text{ll}(\text{right} \ x))) \rightarrow I \quad \vdash \text{inj-r}(a,b) : \text{unit} + (\varphi \times [\varphi]) \rightarrow I}}{\vdash \text{inj-r}(a,b) \rightarrow I}
\]

This approach to recursive types is known as the equi-recursive approach [2, 29], because equal-
ity 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 [50]. Its main disadvantage is that, in the presence of equi-recursive types, apparently meaningless programs have types.

Example 8.7 Self-application \( \lambda x.xx \) has the type \( \mu X.X \rightarrow \varphi \):

\[
\begin{align*}
\Gamma \vdash E : A &\quad \text{(fold)} : \quad \Gamma \vdash \text{fold}(E) : \mu X.A \\
\Gamma \vdash \text{fold}(E) : \mu X.A &\quad \text{unfold} : \quad \Gamma \vdash \text{unfold}(E) : A[\mu X.A/\varphi]
\end{align*}
\]

Notice that it is possible to apply \( \text{unfold} \) directly after \( \text{fold} \), 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 \( \text{unfold}(\text{fold}(E)) \rightarrow E \).

Example 8.9 A term now has a \([B]\) type if either it is of the shape \( \text{fold}(\text{inj-l}(\)) \) or \( \text{fold}(\text{inj-r}(a,b)) \):

\[
\begin{align*}
\Gamma \vdash () &\quad \text{(unit)} : \quad \Gamma \vdash a : B \\
\Gamma \vdash \text{inj-l}(\) &\quad \text{(inj-l)} : \quad \Gamma \vdash (a,b) : B \times [B] \\
\Gamma \vdash \text{fold}(\text{inj-l}(\)) &\quad \text{(fold)} : \quad \Gamma \vdash \text{fold}(\text{inj-l}(a,b)) : [B]
\end{align*}
\]
\[ x : \mu X.X \rightarrow \varphi \vdash x : \mu X.X \rightarrow \varphi \]
\[ x : \mu X.X \rightarrow \varphi \vdash \text{unfold}(x) : (\mu X.X \rightarrow \varphi) \rightarrow \varphi \]
\[ x : \mu X.X \rightarrow \varphi \vdash \text{fold}(\text{unfold}(x)) : \mu X.X \rightarrow \varphi \]

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.

Remark that the two added rules 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:

\[
\Gamma \vdash E : A[\mu X.A/X] \\
\Gamma \vdash \text{fold}_{\mu X.A}(E) : \mu X.A \\
\Gamma \vdash E : \mu X.A
\]

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 \([\cdot] \) type constructor

\[ [\varphi] = \text{unit} + (\varphi \times [\varphi]) \]

as a solution to the type equation

\[ A = \text{unit} + (B \times A); \]

we have the type assignment rules

\[
\text{(fold}_{\text{List}}) : \quad \Gamma \vdash E : \text{unit} + (\varphi \times [\varphi]) \\
\Gamma \vdash \text{fold}_{\text{List}}(E) : [\varphi] \\
\text{(unfold}_{\text{List}}) : \quad \Gamma \vdash E : [\varphi] \\
\Gamma \vdash \text{unfold}_{\text{List}}(E) : \text{unit} + (\varphi \times [\varphi])
\]

The \text{(fold}_{\text{List}}) rule now expresses: if we have derived that a term \( E \) has type \( \text{unit} + (B \times [B]) \) (typically by deriving either unit and using \((\text{inj}-l)\) or deriving \( B \times [B] \) and using \((\text{inj}-r)\)), 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 \text{(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 \( \text{unit} + (B \times [B]) \) (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 is written

\[ \text{fold}(\text{inj}-l( )) \]

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

\[ \text{case}(\text{unfold } l, \lambda n \ldots, \lambda c. \text{let } \text{hd} = \text{left } c \text{ in let } \text{tl} = \text{right } c \text{ in } \ldots) \]

### 8.5 Recursive data types

More generally, recursive (data) types can be defined via:
\[ C \varphi = A_C[\varphi] \]

where \( C \) is the user-defined type constructor, defined over a number of type variables \( \varphi \), and \( A_C[\varphi] \) 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 \( C \varphi \) and \( A_C[\varphi] \) are distinct types, but it is possible to convert one into the other via folding and unfolding.

**Definition 8.10** The syntax is extended by
\[
E ::= \cdots | \text{fold}_C(E) | \text{unfold}_C(E)
\]
and we add the reduction rule
\[
\text{unfold}_C(\text{fold}_C(E)) \to E
\]
and the type assignment rules
\[
\text{(fold}_C) : \frac{\Gamma \vdash E : A_C[\varphi]}{\Gamma \vdash \text{fold}_C(E) : C \varphi} \quad \text{(unfold}_C) : \frac{\Gamma \vdash E : C \varphi}{\Gamma \vdash \text{unfold}_C(E) : A_C[\varphi]}
\]
for every type definition \( C \varphi = A_C[\varphi] \).

We will often omit the type-subscript \( C \).

Converting \( C \varphi \) to its unfolding \( A_C[\varphi] \) – or folding \( A_C[\varphi] \) to \( C \varphi \) – requires an explicit use of \( \text{fold}_C \) or \( \text{unfold}_C \), that is, an explicit syntax in the calculus, making a recursive type-conversion only possible on call, i.e. if a \( \text{fold}_C \) or \( \text{unfold}_C \) 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)foldering gives a complicated syntax.

**Example 8.11** In this setting, the (silent) \( \mu \)-conversion in the definition of \( LL \) in Example 8.9 are now made explicit, and \( LL \) becomes
\[
\text{LL} = \text{fix} \ ll. \lambda \text{list. } \text{case} \ (\text{unfold} \ (\text{list}), \lambda x.0, \lambda x.+1 \ (\text{ll}(\text{right} \ x)))
\]

Notice that now
\[
\begin{align*}
(\text{fix} \ ll. \lambda \text{list. } \text{case} \ (\text{unfold} \ (\text{list}), \lambda x.0, \lambda x.+1 \ (\text{ll}(\text{right} \ x)))) & \ (\text{fold} \ (\text{inj} \ r \ (a,b))) \to \\
(\lambda \text{list. } \text{case} \ (\text{unfold} \ (\text{list}), \lambda x.0, \lambda x.+1 \ (\text{ll}(\text{right} \ x)))) & \ (\text{fold} \ (\text{inj} \ r \ (a,b))) \to \\
\text{case} \ (\text{unfold} \ (\text{fold} \ (\text{inj} \ r \ (a,b))), \lambda x.0, \lambda x.+1 \ (\text{ll}(\text{right} \ x))) & \to \\
\text{case} \ (\text{inj} \ r \ (\langle a,b \rangle), \lambda x.0, \lambda x.+1 \ (\text{ll}(\text{right} \ x))) & \to \\
(\lambda x.+1 \ (\text{ll}(\text{right} \ x))) & \langle a,b \rangle \\
+1 \ (\text{ll}(\text{right} \ a,b)) & \to \\
+1 \ (\text{ll} \ (a,b)) & \\
\end{align*}
\]

### 8.6 Algebraic datatypes

In ML and Haskell, structural products and sums are defined via iso-recursive types, yielding so-called algebraic data types [15]. The idea is to avoid requiring both a (type) name and a (field or tag) number, as in \( \text{fold} \ (\text{inj} \ 1(\cdot)) \). Instead, it would be desirable to mention a single name, as in \( \llbracket \cdot \rrbracket \) for the empty list. This is permitted by algebraic data type declarations.

**Definition 8.12** An algebraic data type constructor \( C \) is introduced via a record type definition:
\[
C \varphi = \Pi_{i=1}^k \ell_i : A_i[\varphi] \quad (\text{short for } \ell_1 : A_1[\varphi] \times \cdots \times \ell_k : A_k[\varphi])
\]
or the or **variant** type definition:

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

The record labels \( \ell_i \) 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 \( i \).

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

\[
\text{Bool} = \text{True} : \text{unit} + \text{False} : \text{unit}
\]

which we normally write as

\[
\text{Bool} = \text{True + False}
\]

**Definition 8.13** The **record** type definition

\[ C \varphi = \Pi_{i=1}^k \ell_i : A_i[\varphi] \]

introduces the constructors \( \ell_i \) for \( 1 \leq i \leq k \) and build\(_C\), with the following rules:

\[
(\ell_i) : \quad \Gamma \vdash E : C \varphi \quad (1 \leq i \leq k) \quad \text{(build\(_C\))} : \quad \Gamma \vdash E_1 : A_1[\varphi] \cdots \Gamma \vdash E_k : A_k[\varphi] \quad \Gamma \vdash \text{build\(_C\)} E_1 \ldots E_k : C \varphi
\]

so the labels act as projection functions into the product type.

**Example 8.14** In this setting, pairing can be expressed via the product type

\[
\langle \rangle \varphi_1 \varphi_2 = \text{left} : \varphi_1 \times \text{right} : \varphi_2
\]

and the rules

\[
\Gamma \vdash \langle \rangle : \varphi_1 \varphi_2 \quad \gamma \vdash \langle \rangle : \varphi_1 \varphi_2 \quad \gamma \vdash \text{left} (E) : \varphi_1 \quad \Gamma \vdash \text{right} (E) : \varphi_2 \quad \gamma \vdash \text{build\(_\{\}\)} E_1 E_2 : \langle \rangle : \varphi_1 \varphi_2
\]

Of course an in-fix notation would give better readability: \( \Gamma \vdash \langle E_1, E_2 \rangle : \langle \varphi_1, \varphi_2 \rangle \). **Definition 8.15** The **variant** type definition

\[ C \varphi = \Sigma_{i=1}^k \ell_i : A_i[\varphi] \]

introduces the constructors \( \ell_i \) (with \( 1 \leq i \leq k \)) and case\(_C\), typeable via the rules:

\[
(\ell_i) : \quad \Gamma \vdash E : A_i[\varphi] \quad (1 \leq i \leq k) \quad \text{(case\(_C\))} : \quad \Gamma \vdash E : C \varphi \quad \Gamma \vdash E_1 : A_1[\varphi] \rightarrow C \cdots \Gamma \vdash E_k : A_k[\varphi] \rightarrow C \quad \Gamma \vdash \text{case\(_C\)} (E, E_1, \ldots, E_k) : C
\]

(Notice that the latter is a generalised case of the rule presented above.)

For readability, we write case \( E [\ell_1 : E_1 \cdots \ell_k : E_k] \) for case\(_C\) (\( E, E_1, \ldots, E_k \)) when \( k > 0 \), and \( C \varphi = \Sigma_{i=1}^k \ell_i : A_i[\varphi] \), thus avoiding to label case.

We can now give the type declaration for lists as

\[
[\varphi] = [ ] : \text{unit} + \text{Cons} : \varphi \times [\varphi]
\]

This gives rise to the rules
\[ \begin{array}{c}
(\[\]) : \Gamma \vdash [\cdot] : \varphi \\
(\text{Cons}) : \Gamma \vdash E : \varphi \times [\varphi] \\
\Gamma \vdash \text{Cons} E : [\varphi] \\
(\text{case}[\varphi]) : \\
\Gamma \vdash E_1 : [\varphi] \\
\Gamma \vdash E_2 : [\cdot] \rightarrow \varphi' \\
\Gamma \vdash E_3 : (\varphi \times [\varphi]) \rightarrow \varphi' \\
\Gamma \vdash \text{case}[\varphi] (E_1, E_2, E_3) : \varphi'
\end{array} \]

Notice that here \text{Cons} and [\cdot] 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:

\[
\begin{array}{l}
(\text{fix } \ll . \lambda \text{list}. \text{case}[\varphi] (\text{list}, \lambda x.0, \lambda x. +1 (\text{ll}(\text{right } x)) \,)) (\text{Cons}(a,b))
\end{array}
\]

or

\[
(\text{fix } \ll . \lambda \text{list} \cdot \text{case} (\text{list}, \text{Nil} : \lambda x.0, \text{Cons} : \lambda x. +1 (\text{ll}(\text{right } x)) \,)) (\text{Cons}(a,b))
\]

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 8.16** Using Example 8.7, find a type for \( (\lambda x. xx) (\lambda x. xx) \).

**Exercise 8.17** Similar to the previous exercise, find a type for \( \lambda f. (\lambda x.f(xx)) (\lambda x.f(xx)) \).

**Exercise 8.18** Give the derivation for

\[
\Gamma \vdash (\text{fix } \ll . \lambda \text{list}. \text{case} (\text{unfold}(\text{list}), \lambda x.0, \lambda x. +1 (\text{ll}(\text{right } x))) \,)) (\text{fold}(\text{inj}\cdot\text{r}(a,b))) : I
\]

**Exercise 8.19** Give the derivation for

\[
\text{case} (\text{list}, [\cdot] : \lambda x.0, \text{Cons} : \lambda x. +1 (\text{ll}(\text{right } x)) \,)(\text{Cons}(a,b)) : I
\]

### 9 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 [17, 19, 20, 12, 18, 21, 4, 6], all more or less equivalent; we will use the system of [4] here, because it is the most intuitive.

Intersection types are an extension of Curry types by adding an extra type constructor ‘∩’, that enriches the notion of type assignment in a dramatic way. In fact, type assignment is now closed for =_β, 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 [7, 35], the trivial being to do without intersection types at all, and fall back to Curry types.

#### 9.1 Intersection types

Intersection types are defined by extending Curry types with the type constructor ‘∩’; we limit the occurrence of intersection types in arrow types to the left-hand side, so have to use a two-level grammar.
**Definition 9.1** (Strict types) 

i) The set of is defined by the grammar:

\[
A ::= \varphi \mid (\sigma \to A) \quad \text{(strict types)}
\]

\[
\sigma, \tau ::= (A_1 \cap \cdots \cap A_n) \quad (n \geq 0) \quad \text{(intersection types)}
\]

ii) On \(\mathcal{T}\), the relation \(\leq\) is defined as the smallest relation satisfying:

\[
\forall 1 \leq i \leq n \ [A_1 \cap \cdots \cap A_n \leq A_i] \quad (n \geq 1)
\]

\[
\forall 1 \leq i \leq n \ [A \leq A_i] \Rightarrow A \leq A_1 \cap \cdots \cap A_n \quad (n \geq 0)
\]

\[
\sigma \leq \tau \leq \rho \Rightarrow \sigma \leq \rho
\]

iii) We define the relation \(\sim\) by:

\[
\sigma \leq \tau \leq \rho \Rightarrow \sigma \sim \tau
\]

\[
\sigma \sim \tau \& \ A \sim B \Rightarrow \sigma \to A \sim \tau \to B
\]

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, ‘∩’ binds stronger than ‘→’, so \(C \cap D \to C \to D\) stands for \(((C \cap D) \to (C \to D))\).

We will write \(\cap_n A_i\) for \(A_1 \cap \cdots \cap A_n\), and use \(\top\) to represent an intersection over zero elements: if \(n = 0\), then \(\cap_n A_i = \top\), so, in particular, \(\top\) does not occur in an intersection subtype.

Moreover, intersection type schemes (so also \(\top\)) occur in strict types only as subtypes at the left-hand side of an arrow type.

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 9.2** (Contexts) 

i) A statement is an expression of the form \(M : \sigma\), where \(M\) is the subject and \(\sigma\) is the predicate of \(M\).

ii) A context \(\Gamma\) is a set of statements with (distinct) variables as subjects.

iii) The relations \(\leq\) and \(\sim\) are extended to contexts by:

\[
\Gamma \leq \Gamma' \iff \forall x : \tau \in \Gamma' \exists x : \sigma \in \Gamma \ [\sigma \leq \tau]
\]

\[
\Gamma \sim \Gamma' \iff \Gamma \leq \Gamma' \leq \Gamma
\]

iv) Given two bases \(\Gamma_1\) and \(\Gamma_2\), we define the basis \(\Gamma_1 \cap \Gamma_2\) as follows:

\[
\Gamma_1 \cap \Gamma_2 \triangleq \{ x : \sigma \cap \tau \mid x : \sigma \in \Gamma_1 \& x : \tau \in \Gamma_2 \} \cup
\]  

\[
\{ x : \sigma \mid x : \sigma \in \Gamma_1 \& x \notin \Gamma_2 \} \cup \{ x : \tau \mid x : \tau \in \Gamma_2 \& x \notin \Gamma_1 \}
\]

and write \(\cap_n \Gamma_i\) 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.

### 9.2 Intersection type assignment

This type assignment system will derive judgements of the form \(\Gamma \vdash M : \sigma\), where \(\Gamma\) is a context and \(\sigma\) a type.

**Definition 9.3** 

i) Strict type assignment and strict derivations are defined by the following natural deduction system (where all types displayed are strict, except \(\sigma\) in the derivation rules \((\to I)\) and \((\to E)\)): 

```
```
Lemma 9.4 (Generation Lemma) i) \( \Gamma \vdash MN : A \iff \exists B [ \Gamma \vdash M : \sigma \rightarrow A \land \Gamma \vdash N : \sigma ] \).

ii) \( \Gamma \vdash \lambda x : A. D : (\Lambda x : A. D) \iff \exists \sigma, D [ A = \sigma \rightarrow D \land \Gamma \setminus x \vdash x : \sigma \vdash \lambda x : A. D ] \).

iii) \( \Gamma \vdash M : \sigma \iff \{ x : \sigma \in \Gamma \mid x \in \text{fv}(M) \} \vdash M : \sigma \).

Example 9.5 In this system, we can derive both \( \emptyset \vdash (\lambda xy.z(xyz))(\lambda ab.a) : T \rightarrow A \rightarrow A \) and \( \emptyset \vdash \lambda yz.z : T \rightarrow A \rightarrow A \):

\[
\begin{align*}
\Gamma \vdash x : A & \rightarrow T \rightarrow A \quad \Gamma \vdash z : A \\
\Gamma \vdash xz : T \rightarrow A & \quad \Gamma \vdash yz : T \\
xz & \vdash x : A \rightarrow T \rightarrow A, yz \vdash xz(yz) : A \\
xz & \vdash y : T, z : A \vdash xz(yz) : A \\
\end{align*}
\]

(where \( \Gamma = x : A \rightarrow T \rightarrow A, y : T, z : A \)) and

\[
\begin{align*}
z & \vdash A, y : T, z : A \\
y & \vdash A, z : T \rightarrow A \\
\end{align*}
\]

Notice that, by using \( \Gamma = x : A \rightarrow T \rightarrow A, y : T, z : A \) in the first derivation above, we could as well have derived \( \emptyset \vdash (\lambda xy.z(xyz))(\lambda ab.a) : B \rightarrow A \rightarrow A \), for any Curry types \( A \) and \( B \); as we have seen in Example 2.9, this is not possible in Curry’s system.

9.3 Subject reduction and normalisation

That subject reduction holds in this system is not difficult to see. The proof follows very much the same lines as the one given in Theorem 2.4, and will follow below; first we give an intuitive argument.

Suppose there exists a type assignment for the redex \((\lambda x.M)N\), so there are a context \( \Gamma \) and a type \( A \) such that there is a derivation for \( \Gamma \vdash (\lambda x.M)N : A \). Then by \((\rightarrow E)\) there is a type \( \cap_i B_i \) such that there are derivations \( \Gamma \vdash \lambda x.M : \cap_i B_i \rightarrow A \) and \( \Gamma \vdash N : \cap_i B_i \). Since \((\rightarrow I)\) should be the last step performed in the derivation for \( \Gamma \vdash (\lambda x.M)N : \cap_i B_i \rightarrow A \) (the type is not an intersection), there is also a derivation for \( \Gamma, x : \cap_i B_i \vdash M : A \). Since \((\cap I)\) must have been the last step performed in the derivation for \( \Gamma \vdash N : \cap_i B_i \), for every \( 1 \leq i \leq n \), there exists a derivation for \( \Gamma \vdash N : B_i \). In other words, we have the derivation:
the derivation for \( \Gamma \) of different types \( A \)

suppose we have derived and the term-variable can have different types within a derivation, combined in an intersection, \( x \) would contain more than one type for

Then in Curry’s system \( M \)

yielding

by the derivation for

yielding

The problem to solve in a proof for closure under \( \beta \)-equality is then that of \( \beta \)-expansion: suppose we have derived \( \Gamma \vdash M[N/x] : A \) and also want to derive \( \Gamma \vdash (\lambda x. M) N : A \).

We distinguish two cases. If the term-variable \( x \) occurs in \( M \), then the term \( N \) is a subterm of \( M[N/x] \), so \( N \) is typed in the derivation for \( \Gamma \vdash M[N/x] : A \); assume it is typed with the different types \( A_1, \ldots, A_n \), so, for \( 1 \leq i \leq n \), \( \Gamma \vdash N : A_i \).

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 \( \Gamma, x : \cap_i A_i \vdash M : A \), and from this we get, by rule \( (\to I) \), \( \Gamma \vdash \lambda x. M : \cap_i A_i \to A \). Since, for every \( 1 \leq i \leq n \), \( \Gamma \vdash N : A_i \), by rule \( (\cap I) \) we also have \( \Gamma \vdash N : \cap_i A_i \). Then, using \((\to E)\), the redex can be typed.

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 \( \Gamma \vdash M[N/x] : A \), and in fact we have
Lemma 9.8 holds in both directions.

By induction on Lemma 9.7

\[ \vdash \lambda x. M : \top \rightarrow A \]

By weakening, the term \( M \) can then be typed by \( \Gamma, x : \top \vdash M : A \), and from this we get, by rule \((\rightarrow I)\), \( \Gamma \vdash \lambda x. M : \top \rightarrow A \). Since also \( \Gamma \vdash N : \top \) by rule \((\cap I)\), using \((\rightarrow E)\), the redex can be typed.

\[
\begin{align*}
\frac{\map{D_1}}{\Gamma \vdash \lambda x. M : \top \rightarrow A} & \quad (\rightarrow I) \\
\frac{\map{D_1}}{\Gamma \vdash M : A} & \quad (\cap I) \\
\frac{\map{D_1}}{\Gamma \vdash (\lambda x. M) N : A} & \quad (\rightarrow E)
\end{align*}
\]

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 9.6** If \( \Gamma \vdash M : \sigma \) and \( \sigma \leq \tau \), and \( \Gamma' \leq \Gamma \), then \( \Gamma' \vdash M : \tau \).

**Lemma 9.7**

i) If \( \Gamma \vdash M : \sigma \), and \( \Gamma' \supseteq \Gamma \), then \( \Gamma' \vdash M : \sigma \).

ii) If \( \Gamma \vdash M : \sigma \), then \( \{ x : \tau \mid x : \tau \in \Gamma \land x \in \text{fv}(M) \} \vdash M : \sigma \).

Also, a substitution lemma is needed. Notice that, unlike for Curry’s system, the implication holds in both directions.

**Lemma 9.8** \( \exists \sigma \left[ \Gamma, x : \sigma \vdash M : A \land \Gamma \vdash N : \sigma \right] \leftrightarrow \Gamma \vdash M[N/x] : A \).

**Proof:** By induction on \( M \). Only the case \( A \in T_0 \) is considered.

\[
\begin{align*}
(M \equiv x) & : \quad (\Rightarrow) : \exists \sigma \left[ \Gamma, x : \sigma \vdash x : A \land \Gamma \vdash N : \sigma \right] \quad (Ax) \\
& \quad \exists A_i (i \in \set{\mathbf{y}}, j \in \set{\mathbf{n}} \mid A = A_j \land \Gamma \vdash N : \cap _n A_i) \quad (9.6) \\
& \quad \Gamma \vdash x[N/x] : A_j, \\
\left(\Leftarrow\right) & : \quad \Gamma \vdash x[N/x] : A \Rightarrow \Gamma, x : A \vdash x : A \land \Gamma \vdash N : A. \\
(M \equiv y \neq x) & : \quad (\Rightarrow) : \exists \sigma \left[ \Gamma, x : \sigma \vdash y : A \land \Gamma \vdash N : \sigma \right] \Rightarrow (9.7) \Gamma \vdash y[N/x] : A. \\
& \quad \left(\Leftarrow\right) : \quad \Gamma \vdash y[N/x] : A \Rightarrow \Gamma \vdash y : A \land \Gamma \vdash N : T. \\
(M \equiv \lambda y. M') & : \quad (\Leftarrow) : \exists \sigma \left[ \Gamma, x : \sigma \vdash \lambda y. M' : A \land \Gamma \vdash N : \sigma \right] \Leftarrow (\rightarrow I) \\
& \quad \exists \sigma, \tau, B \quad \left[ \Gamma, x : \sigma \vdash \lambda y. M' : B \land \Gamma \vdash N : \sigma \right] \Leftarrow (9.6) \\
& \quad \exists \tau, B \quad \left[ \Gamma, y : \tau \vdash M'[N/x] : B \land \Gamma \vdash N : A = \tau \Rightarrow B \right] \Leftarrow (\rightarrow I) \\
& \quad \Gamma \vdash \lambda y. M'[N/x] : A. \\
(M \equiv M_1 M_2) & : \quad (\Leftarrow) : \quad \Gamma \vdash M_1 M_2[N/x] : A \quad \Leftarrow (\rightarrow E) \\
& \quad \exists \sigma \left[ \Gamma \vdash M_1[N/x] : \sigma \rightarrow A \land \Gamma \vdash M_2[N/x] : \sigma \right] \Leftarrow (9.6) \\
& \quad \exists \sigma_1, \sigma_2, \sigma \quad \left[ \Gamma, x : \sigma_1 \vdash M_1 : \sigma \rightarrow A \land \Gamma \vdash N : \sigma_1 \land \Gamma, x : \sigma_2 \vdash M_2 : \sigma \land \Gamma \vdash N : \sigma_2 \right] \\
& \quad \Leftarrow (\sigma = \sigma_1 \land \sigma_2) \land (\cap I) \land (9.6)
\end{align*}
\]

Notice that, although we only present the case for strict types, we do need the property for all types in the last part.

**Corollary 9.9** If \( M =_{\beta} N \), then \( \Gamma \vdash M : A \) if and only if \( \Gamma \vdash N : A \), so the following rule is admissible in \( \vdash_{\beta} \):

\[
\frac{\map{D_1}}{\Gamma \vdash \lambda x. M} {\beta} (\map{D_1}) \quad (\map{D_1})
\]

**Proof:** By induction on the definition of \( =_{\beta} \). The only part that needs attention is that of a redex,
\[ \Gamma \vdash (\lambda x.M)N : A \iff \Gamma \vdash M[N/x] : A, \text{ where } A \in \mathcal{T}_f; \text{ all other cases follow by straightforward induction.} \] To conclude, notice that, if \( \Gamma \vdash (\lambda x.M)N : A \), then, by \( \to_E \) and \( \to_I \), there exists \( C \) such that \( \Gamma, x : C \vdash M : A \) and \( \Gamma \vdash N : C \). The result then follows from Lemma 9.8.

Interpreting a term \( M \) by its set of assignable types \( \mathcal{T}(M) = \{ A \mid \exists \Gamma \ [\Gamma \vdash_M M : A] \} \) gives a semantics for \( M \), and a filter model for the Lambda Calculus (for details, see [12, 4, 6]).

**Example 9.1** Types are not invariant by \( \eta \)-reduction. For example, notice that \( \lambda xy.x y \to_{\eta} \lambda x.x \); we can derive \( \emptyset \vdash \lambda x.x : (\varphi_1 \to \varphi_2) \to \varphi_1 \cap \varphi_3 \to \varphi_2 \), but not \( \emptyset \vdash \lambda x.x : (\varphi_1 \to \varphi_2) \to \varphi_1 \cap \varphi_3 \to \varphi_2 \).

**Proof:** By induction on the structure of lambda terms in head normal form.

\[ \vdash x : \varphi_1 \to \varphi_2, y : \varphi_1 \cap \varphi_3 \vdash x : \varphi_1 \to \varphi_2 \]

\[ \vdash x : \varphi_1 \to \varphi_2, y : \varphi_1 \cap \varphi_3 \vdash y : \varphi_1 \to \varphi_1 \]

We cannot derive \( \emptyset \vdash \lambda x.x : (\varphi_1 \to \varphi_2) \to \varphi_1 \cap \varphi_3 \to \varphi_2 \), 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, [4]).

The intersection type assignment system allows for a very nice characterisation, through assignable types, of normalisation, head-normalisation, and strong normalisation [12, 6].

**Theorem 9.10** If \( M \) is in normal form, then there are \( \Gamma \) and \( A \) such that \( \Gamma \vdash_M M : A \), and in this derivation \( \top \) does not occur.

**Proof:** By induction on the structure of lambda terms in normal form.

\[ i) \ M \equiv x. \text{ Take } A \text{ such that } \top \text{ does not occur in } A. \text{ Then } x : A \vdash x : A. \]

\[ ii) \ M \equiv \lambda x.M', \text{ with } M' \text{ in normal form. By induction there are } \Gamma \text{ and } B \text{ such that } \Gamma \vdash_M M' : B \text{ and } \top \text{ does not occur in this derivation. In order to perform the } \to_I \text{-step, } \Gamma \text{ must contain (whether or not } x \text{ is free in } M') \text{ a statement with subject } x \text{ and predicate, say, } \sigma. \text{ But then of course } \Gamma \setminus x \vdash \lambda x.M' : \sigma \to B \text{ and } \top \text{ does not occur in this derivation.} \]

\[ iii) \ M \equiv xM_1 \ldots M_n, \text{ with } M_1, \ldots, M_n \text{ in normal form. By induction there are } \Gamma_1, \ldots, \Gamma_n \text{ and } \sigma_1, \ldots, \sigma_n \text{ such that for every } i \in \{1, \ldots, n\}, \Gamma_i \vdash_M M_i : \sigma_i \text{ and } \top \text{ does not occur in these derivations.} \]

Take \( \tau \text{ strict, such that } \top \text{ does not occur in } \tau, \text{ and } \Gamma = \cap_{i \in \{1, \ldots, n\}} \Gamma_i \cap x : \sigma_1 \to \cdots \sigma_n \to \tau. \text{ Then } \Gamma \vdash_M xM_1 \ldots M_n : \tau \text{ and in this derivation } \top \text{ does not occur.} \]

**Theorem 9.11** If \( M \) is in head normal form, then there are \( \Gamma \) and \( A \) such that \( \Gamma \vdash_M M : A \).

**Proof:** By induction on the structure of lambda terms in head normal form.

\[ i) \ M \equiv x. \text{ Then } x : A \vdash x : A, \text{ for any } A. \]

\[ ii) \ M \equiv \lambda x.N, \text{ with } N \text{ in head normal form. By induction there are } \Gamma \text{ and } B \text{ such that } \Gamma \vdash_N N : B. \text{ As in the previous theorem, } \Gamma \text{ must contain a statement with subject } x \text{ and predicate, say, } \sigma. \text{ But then of course } \Gamma \setminus x \vdash \lambda x.N : \sigma \to B. \]

\[ iii) \ M \equiv xM_1 \ldots M_n, \text{ with } M_1, \ldots, M_n \text{ lambda terms. Take } B \text{ strict, then also (with } n \text{ times } \top) \]

\[ \top \to \top \to \cdots \to \top \to B \text{ is strict, and } x : \top \to \top \to \cdots \to \top \to B \vdash_M xM_1 \ldots M_n : B. \]

**Theorem 9.12** If \( M \) has a normal form, then there exists \( \Gamma, A \) such that \( \Gamma \vdash_M M : A \) and \( \Gamma \) and \( A \) are \( \top \)-free.

**Proof:** By 9.10 and 9.9.

From Theorem 9.10 we can conclude that \( \Gamma \) and \( \sigma \) do not contain \( \top \), but by the proof of Theorem 9.9, the property that \( \top \) does not occur at all in the derivation is, in general, lost.
Theorem 9.13 If $M$ has a head normal form, then there exists $\Gamma, A$ such that $\Gamma \vdash_{\cap} M : A$.

Proof: By 9.11 and 9.9.

The converse of these last two results also holds, but requires a bulky proof.

The main characterisation properties of intersection types can now be stated as follows:

Theorem 9.14 i) There are $\top$-free $\Gamma, A$ such that $\Gamma \vdash_{\cap} M : A$ if and only if $M$ has a normal form.

ii) There are $\Gamma, A \in T$ such that $\Gamma \vdash_{\cap} M : A$ if and only if $M$ has a head normal form.

iii) $M$ is strongly normalisable, if and only if there are $\Gamma$ and $A$ such that $\Gamma \vdash_{\cap} M : A$, and in this derivation $\top$ is not used at all.

We can now reason that the converse of Corollary 9.9 does not hold: terms $N$ that do not have a head-normal form are all only typeable with $\Gamma \vdash_{\cap} N : \top$, but cannot all be converted to each other.

Because all these properties can be reduced to the halting problem, type assignment with intersection types is undecidable.

It is possible to define a notion of principal pair for lambda terms using intersection types [53, 5, 6]. A semi-algorithm is defined in [52]; if a term has a principal pair is, of course, undecidable.

9.4 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 9.15 The let is used for the case that we would like to type the redex $(\lambda x. E_1) E_1$ whenever the contractum is typeable using Curry types, but cannot:

Using rank 2 types, the let-construct is not needed, since we can type the redex $(\lambda x. E_1) E_1$ directly (let $\Gamma' = \Gamma, x : A[B/\varphi] \cap A[C/\varphi]$):

10 Featherweight Java

In this section we will focus on Featherweight Java (fj) [34], a restriction of Java defined by removing all but the most essential features of the full language; fj bears a similar relation to Java as the $\lambda$-calculus does to languages such as ML [45] and Haskell [33]. We illustrate the
expressive power of this calculus by showing that it is Turing complete through an embedding of Combinatory Logic (cl) – and thereby also the \( \lambda \)-calculus.

As in other class-based object-oriented languages, \( \mathcal{FJ} \) defines classes, which represent abstractions encapsulating both data (stored in fields) and the operations to be performed on that data (encoded as methods). Sharing of behaviour is accomplished through the inheritance of fields and methods from parent classes. Computation is mediated by instances of these classes (called objects), which interact with one another by calling (also called invoking) methods on each other and accessing each other’s (or their own) fields.

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

**Definition 10.1 (\( \mathcal{FJ} \mathcal{E} \) Syntax)** A \( \mathcal{FJ} \mathcal{E} \) programs \( \text{Prog} \) consist of a class table \( \text{CT} \), comprising the class declarations, and an expression \( e \) to be run (corresponding to the body of the \text{main} method in a real Java program). They are defined by the grammar:

\[
\begin{align*}
\text{expr} & ::= x \mid \text{this} \mid \text{new } \text{class } C \mid e.f \mid e.m(\overline{v}) \\
\text{fd} & ::= \text{class } C \vs \text{extends } C' \{ \text{fd} \} \\
\text{md} & ::= D \{ C_1 \; x_1, \ldots, \; C_n \; x_n \} \{ \text{return } e; \} \\
\text{cd} & ::= \text{class } C \text{ extends } C' \{ \text{fd} \} \{ \text{md} \} \\
\text{CT} & ::= \text{cd} \\
\text{Prog} & ::= (\text{CT}, e)
\end{align*}
\]

Notice that, contrary to the \( \lambda \)-calculus, the language is first-order; there is no notion of application.

From this point, for readability, all the concepts defined are program dependent (or more precisely, parametric on the class table); however, since a program is essentially a fixed entity, the class table will be left as an implicit parameter in the definitions that follow. We also only consider programs which conform to some sensible well-formedness criteria: no cycles in the inheritance hierarchy, and fields and methods in any given branch of the inheritance hierarchy are uniquely named. An exception is made to allow the redeclaration of methods, providing that only the body of the method (so not the types) is allowed to differ from the previous declaration (in the parlance of class-based oo, this is called method override).

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

**Definition 10.2 (Lookup Functions)** The following lookup functions are defined to extract the names of fields and bodies of methods belonging to (and inherited by) a class.

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

\[
\begin{align*}
\text{cn}(\text{class } C \text{ extends } D \{ \text{fd} \}) & = C \\
\text{fn}(C \; f) & = f
\end{align*}
\]

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

\[
\text{CT}(C) = \text{cd} \text{ if } \text{cn}(\text{cd}) = C \text{ and } \text{cd} \in \text{CT}
\]

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

\[\text{\footnotesize{Not a variable in the traditional sense, since it is not used to express a position in the method’s body where a parameter can be passed.}}\]
\[ \mathcal{F}(\text{object}) = \epsilon \]
\[ \mathcal{F}(c) = \mathcal{F}(c') \cdot f \] if \( CT(c) = \text{class } C \text{ extends } C' \ \{ fd \ \text{md} \} \]
and \( \text{fn}(fd_i) = f_i \) for all \( i \in n. \)

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

\[
Mb(C, m) = (x, e)
\]
\[
\text{if } CT(C) = \text{class } C \text{ extends } C' \ \{ fd \ \text{md} \}, \text{ and there exist } \\
C_0, C, \text{ such that } C_0 \ m(C_1, x_1, \ldots, C_n, x_n) \ \{ \text{return } e; \} \in \text{md.}
\]

\[
Mb(C, m) = Mb(C', m)
\]
\[
\text{if } CT(C) = \text{class } C \text{ extends } C' \ \{ fd \ \text{md} \}, \text{ and there are no } \\
C_0, C, x, e \text{ such that } C_0 \ m(C_1, x_1, \ldots, C_n, x_n) \ \{ \text{return } e; \} \in \text{md.}
\]

**v)** \( \text{vars}(e) \) returns the set of variables used in the expression \( e. \)

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

\[
\text{if } Mb(C, m) = (x, e_b) \text{ then } \text{vars}(e_b) \subseteq \{ x_1, \ldots, x_n \}
\]

As is clear from the definition of classes, if \( \text{class } C \text{ extends } D \ \{ \overline{f} \ \overline{m} \} \) is a class declaration, then \( \text{new } C(\overline{e}) \) creates an object where the expression \( e; \) is attached to the field \( f_i. \)

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

**Definition 10.3** (Reduction) The reduction relation \( \rightarrow \) is defined by:

\[
\text{new } C(\overline{f}) \cdot f_i \rightarrow e_i \text{ for class name } C \text{ with } \mathcal{F}(C) = \overline{f} \text{ and } i \in n,
\]
\[
\text{new } C(\overline{f}) \cdot m(\overline{e}) \rightarrow e_S \text{ for class name } C \text{ and method } m \text{ with } Mb(C, m) = (x, e'),
\]
where \( S = \{ \text{this} \mapsto \text{new } C(\overline{f}), x_1 \mapsto e_1, \ldots, x_n \mapsto e_n \} \)

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

The standard notion of type assignment in \( fj^\ell \) is a relatively easy affair, and more or less guided by the class hierarchy.

**Definition 10.4** (Nominal Type Assignment for \( fj^\ell \))

\( i \) The set of expressions of \( fj^\ell \) is defined as in Definition 10.1, but adding the alternative cast.

\[
e ::= \cdots | (C) e
\]

Its meaning will become clear through the type assignment rule below.

\( ii \) The sub-typing relation\(^4\) \( <: \) on class types is generated by the extends construct, and is defined as the smallest pre-order satisfying: if \( \text{class } C \text{ extends } D \ \{ \overline{fd} \ \overline{md} \} \in CT, \)
then \( C <: D. \)

\( iii \) Statements are pairs of expression and type, written as \( e : C; \) contexts \( \Gamma \) are defined as sets of statements of the shape \( x : C, \) where all variables are distinct, and possibly containing a

\(^4\) Notice that this relation depends on the class-table, so the symbol \(<:\) should be indexed by \( CT; \) as mentioned above, we leave this implicit.
Oriented Combinatory Logic) is defined using the execution context given by:

\[ (\forall i \in n) \quad (F(c) = \bar{f} \& FT(c, f_i) = D_i \& C_i \ll D_i) \]

To emphasise the expressive power of \( \text{fj} \), we have seen a similar approach in part in the previous sections.

Expression type assignment for the nominal system for \( \text{fj} \) is defined in [34] through the following rules, where (VAR) is applicable to this as well.

\[
\begin{align*}
(\text{NEW}) : & \quad \Gamma \vdash e : C_i \quad (\forall i \in n) \\
& \quad \Gamma \vdash \text{new}\ C_i : C \\
(\text{INVK}) : & \quad \Gamma \vdash e : E \quad \Gamma \vdash e : C_i \quad (\forall i \in n) \\
& \quad \Gamma \vdash e.m(\bar{e}) : C \\
(\text{VAR}) : & \quad \Gamma, x : C \vdash x : C \\
& \quad (\text{FLD}) : \quad \Gamma \vdash e : D \\
& \quad \Gamma \vdash e.f : C \\
& \quad (\text{F} \langle D, f \rangle = C) \\
(\text{U-CAST}) : & \quad \Gamma \vdash e : D \\
& \quad \Gamma \vdash (C)e : C \\
& \quad (D \ll C) \\
(\text{D-CAST}) : & \quad \Gamma \vdash e : D \\
& \quad (C \ll D, C \neq D) \\
& \quad \Gamma \vdash (C)e : C \\
(\text{S-CAST}) : & \quad \Gamma \vdash e : D \\
& \quad \Gamma \vdash (C)e : C \\
& \quad (C \not\ll D, D \not\ll C) \\
\end{align*}
\]

where (VAR) is applicable to this as well.

A declaration of method \( m \) is well typed in \( C \) when the type returned by \( MT(m, C) \) determines a type assignment for the method body.

\[
(\text{METH}) : \quad \text{\{\text{return} } e_b; \text{\}} \quad \text{OK IN} \quad \text{class}\ C \quad \text{extends}\ D \quad \{\cdots\} \\
\]

Classes are well typed when all their methods are and a program is well typed when all the classes are and the expression is typeable.

\[
(\text{CLASS}) : \quad \text{md}_i \quad \text{OK IN} \quad C \quad (\forall i \in n) \quad \text{class}\ C \quad \text{extends}\ D\{\text{md}_i; \cdots\} \quad \text{OK} \\
\]

Remark that we have seen a similar approach in part in the previous sections.

10.1 Encoding Combinatory Logic in \( \text{fj}^\mathcal{E} \)

To emphasise the expressive power of \( \text{fj}^\mathcal{E} \), we will now encode Combinatory Logic.

**Definition 10.5** The encoding of Combinatory Logic (CL) into the \( \text{fj}^\mathcal{E} \) program oocl (Object-Oriented Combinatory Logic) is defined using the execution context given by:

```java
class Combinator extends Object {
    Combinator app(Combinator x) { return this; }
}
class K extends Combinator {
    Combinator app(Combinator x) { return new K(x); }
}
class K_1 extends K {
    Combinator x;
    Combinator app(Combinator y) { return this.x; }
}
class S extends Combinator {
    Combinator app(Combinator x) { return new S(x); }
}
class S_1 extends S {
    Combinator x;
    Combinator app(Combinator y) { return new S(x, y); }
}
class S_2 extends S {
    Combinator y;
    Combinator app(Combinator z) {
```
and the function \[ \llbracket \cdot \rrbracket \] which translates terms of CL into \( fj^\ell \) expressions, and is defined as follows:

\[
\begin{align*}
\llbracket x \rrbracket &= x \\
\llbracket t_1 t_2 \rrbracket &= \llbracket t_1 \rrbracket . \ app ( \llbracket t_2 \rrbracket ) \\
\llbracket K \rrbracket &= \text{new} \ K() \\
\llbracket S \rrbracket &= \text{new} \ S()
\end{align*}
\]

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

**Theorem 10.6** If \( t_1, t_2 \) are terms of CL and \( t_1 \rightarrow^* t_2 \), then \( \llbracket t_1 \rrbracket \rightarrow^* \llbracket t_2 \rrbracket \) in oocl.

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

\[
\begin{align*}
\llbracket S t_1 t_2 t_3 \rrbracket &\triangleq ((\text{new} \ S(). \ app ( \llbracket t_1 \rrbracket )). \ app ( \llbracket t_2 \rrbracket )). \ app ( \llbracket t_3 \rrbracket ) \\
&\rightarrow ((\text{new} \ S_1(\llbracket t_1 \rrbracket ). \ x. \ llbracket t_2 \rrbracket )). \ app ( \llbracket t_3 \rrbracket ) \\
&\rightarrow ((\text{new} \ S_2(\llbracket t_1 \rrbracket , \ llbracket t_2 \rrbracket )). \ app ( \llbracket t_3 \rrbracket ) \\
&\rightarrow (\text{this}. \ x. \ app (z). \ app (\text{this}. \ y. \ app (z)))
\end{align*}
\]

\[
\begin{align*}
&\rightarrow (\text{this} \mapsto \text{new} \ S_2(\llbracket t_1 \rrbracket , \ llbracket t_2 \rrbracket ), \ z \mapsto \llbracket t_3 \rrbracket ) = \\
&\rightarrow ((\text{new} \ S_2(\llbracket t_1 \rrbracket , \ llbracket t_2 \rrbracket ). \ x. \ app (\llbracket t_3 \rrbracket )). \\
&\quad \ app (\text{new} \ S_2(\llbracket t_1 \rrbracket , \ llbracket t_2 \rrbracket ). \ y. \ app (\llbracket t_3 \rrbracket )) \rightarrow^* \\
&\rightarrow (\llbracket t_1 \rrbracket . \ app (\llbracket t_3 \rrbracket )). \ app (\llbracket t_2 \rrbracket . \ app (\llbracket t_3 \rrbracket )) \triangleq
\end{align*}
\]

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

Given the Turing completeness of CL, this result shows that \( fj^\ell \) is also Turing complete.

### 10.2 A functional type system for \( fj^\ell \)

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

**Definition 10.7 (Functional Type Assignment for \( fj^\ell \))** i) Functional types for \( fj \) are defined by:

\[
A, B ::= C \mid \phi \mid \ell:A, \ldots, f_n:A, m_1: \overline{A} \rightarrow B, \ldots, m_k: \overline{C} \rightarrow D \ (n + k \geq 1)
\]

We will write \( R \) for record types, \( \ell \) for arbitrary labels, \( (\ell:A) \in R \) when \( \ell:A \) occurs in \( R \), and assume that all labels are distinct in records.

ii) A context is a mapping from term variables (including \text{this}) to types.

iii) Functional type assignment is defined through:
Using a modified notion of unification, it is possible to define a notion of principal pair for \( \eta^\ell \) expressions, and show completeness.

The elegance of this functional approach is that we can now link types assigned in functional languages to types assignable to object-oriented programs. To show type preservation, we need to define what the equivalent of Curry’s types are in terms of our \( \eta^\ell \) types. To this end, we define the following translation of Curry types.

**Definition 10.8 (Type Translation)** The function \( \llbracket \cdot \rrbracket \), which transforms Curry types\(^5\), is defined as follows:

\[
\llbracket \phi \rrbracket = \phi \\
\llbracket A \to B \rrbracket = \langle \text{app}(\llbracket A \rrbracket) \to \llbracket B \rrbracket \rangle
\]

It is extended to contexts as follows: \( \llbracket \Gamma \rrbracket = \{ x : \llbracket A \rrbracket \mid x : A \in \Gamma \} \).

We can now show the type preservation result.

**Theorem 10.9 (Preservation of Types)** If \( \Gamma \vdash c : A \) then \( \llbracket \Gamma \rrbracket \vdash \llbracket t \rrbracket : \llbracket A \rrbracket \).

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

\[
\begin{align*}
\Pi & \vdash \text{this} : (y : \langle \text{app}(B) \to C \rangle) \quad \text{(VAR)} \\
\Pi & \vdash \text{this}.y : \langle \text{app}(B) \to C \rangle \quad \text{(FLD)} \\
\Pi & \vdash z : B \quad \text{(VAR)} \\
\Pi & \vdash \text{this}.y.\text{app}(z) : C \quad \text{(INVK)} \\
\Pi & \vdash \text{this}.x : \langle \text{app}(B) \to \langle \text{app}(C) \to D \rangle \rangle \quad \text{(FLD)} \\
\Pi & \vdash \text{this}.x.\text{app}(z) : \langle \text{app}(C) \to D \rangle \quad \text{(NEWM)} \\
\Pi & \vdash \text{this}.x.\text{app}(z).\text{app}(\text{this}.y.\text{app}(z)) : D \\
\Pi & \vdash \text{this}.x.\text{app}(z) : \langle \text{app}(B) \to \langle \text{app}(C) \to D \rangle \rangle \quad \text{(FLD)} \\
\Pi & \vdash \text{this}.y : B_2 \quad \text{(VAR)} \\
\Pi & \vdash \text{this}.x : B_1 \quad \text{(FLD)} \\
\Pi & \vdash \text{S}_2(\text{this}.x) : \langle \text{app}(B) \to D \rangle \quad \text{(NEWM)} \\
\Pi & \vdash \text{this}.x : B_1 \quad \text{(VAR)} \\
\Pi & \vdash x : B_1 \quad \text{(VAR)} \\
\Pi & \vdash y : B_2 \quad \text{(NEWF)} \\
\Pi & \vdash \text{S}_2(\text{this}.x) : \langle \text{app}(B) \to D \rangle \quad \text{(NEWM)} \\
\Pi & \vdash \text{this}.x : B_1 \quad \text{(VAR)} \\
\Pi & \vdash \text{S}_1(\text{this}.x) : \langle \text{app}(B) \to D \rangle \quad \text{(NEWM)} \\
\Pi & \vdash S : \langle \text{app}(B) \to \langle \text{app}(B) \to D \rangle \rangle \quad \text{(NEWM)}
\end{align*}
\]

---

\(^5\) Note we have overloaded the notation \( \llbracket \cdot \rrbracket \), which we also use for the translation of CL terms to \( \eta^\ell \) expressions.
(where \( B_1 = \langle \text{app}(B) \rightarrow \text{app}(C) \rightarrow D \rangle \), and \( B_2 = \langle \text{app}(B) \rightarrow C \rangle \), \( II' = \text{this} : \langle x : B_1, y : B_2 \rangle \), and \( II = \text{this} : \langle x : B_1, y : B_2 \rangle \)).

Furthermore, since Curry’s well-known translation of the simply typed \( \lambda \)-calculus into \( \text{CL} \) preserves typeability, we also construct a type-preserving encoding of the \( \lambda \)-calculus into \( \text{FJ}^\ell \).

References


