Chapter 16

# Higher-Order Unification and Matching

Gilles Dowek

Second readers: Jean Goubault-Larrecq and Gopalan Nadathur.

*Contents*

HANDBOOK OF AUTOMATED REASONING
Edited by Alan Robinson and Andrei Voronkov

## 1. Type Theory and Other Set Theories

After the discovery of reasonably efficient proof search methods for first-order logic in the middle of the sixties, a next step seemed to be the extension of these methods to higher-order logic (also called simple type theory), i.e. a logic allowing quantification over sets (i.e. predicates) and functions. Indeed, many problems require such a quantification for a natural expression. A strong advocate for the automatization of higher-order logic was J.A. Robinson who wrote in 1968 that "its adoption, in place of the restricted predicate calculus [i.e. first-order logic], as the basic formalism used in mechanical theorem proving systems, [was] an absolute necessary step if such system [were] ever going to be developed to the point of providing a genuinely useful mathematical service, or of helping to bring a deeper understanding of the process of mathematical thinking" [Robinson 1969].

Replying to Robinson, M. Davis recalled that higher-order logic was just one among several variants of set theory that all permit to reason about sets and functions, and that the choice of this particular variant could only be justified if it was more adequate for automatization than others: "Since higher-order logics are just notational variants of set theories formalized in first-order logic, the question of the use of higher-order formalisms in mechanical theorem-proving is simply a matter of whether or not such formalisms suggest useful algorithms" [Davis 1969].

As we shall see, it is indeed the case that higher-order logic is, so far, more adequate for automatization than other variants of set theory.

### *1.1. Naive Set Theory*

Naive set theory permits the definition of sets *in comprehension*, i.e. by a characteristic property of their elements. For instance we can define the interval of numbers between 4 and 6 by the property $z \in \mathbb{R} \wedge 4 \leq z \wedge z \leq 6$. The *comprehension scheme* is thus stated

$$\forall x_1 \ ... \ \forall x_n \ \exists y \ \forall z \ ((z \in y) \Leftrightarrow P)$$

where $P$ is an arbitrary proposition, $z$ a variable, $x_1, ..., x_n$ the free variables of $P$ except $z$ and $y$ a fresh variable, i.e. a variable different from $z$ and not occurring in $P$.

For example, an instance of this scheme is

$$\exists y \ \forall z \ ((z \in y) \Leftrightarrow (z \in \mathbb{R} \wedge 4 \leq z \wedge z \leq 6))$$

Then, another axiom, the *extensionality axiom*, defines the equality of two sets: two sets are equal if they have the same elements

$$\forall x \ \forall y \ ((\forall z \ (z \in x \Leftrightarrow z \in y)) \Rightarrow x = y)$$

If we want a notation for the objects whose existence is asserted by the comprehension scheme, we skolemize it and introduce function symbols $f_{x_1,...,x_n,z,P}$. The skolemized comprehension scheme

$$\forall x_1 \ ... \ \forall x_n \ \forall z \ ((z \in f_{x_1,...,x_n,z,P}(x_1,...,x_n)) \Leftrightarrow P)$$

is then called *conversion scheme*. If we write the term $f_{x_1,...,x_n,z,P}(x_1,...,x_n)$ as $\{z \mid P\}$ (such a term is called an *abstraction*), the conversion scheme is written

$$\forall x_1 \ ... \ \forall x_n \ \forall z \ ((z \in \{z \mid P\}) \Leftrightarrow P)$$

With this convention, the proposition $P$ in an abstraction $\{z \mid P\}$ cannot contain further abstractions (because the comprehension scheme is only stated for propositions containing no Skolem symbols). If we write $\{z \mid (t_1/x_1,...,t_n/x_n)P\}$ for the term $f_{x_1,...,x_n,z,P}(t_1,...,t_n)$, where the terms $t_1,...,t_n$ may contain abstractions, the propositions in abstractions may then contain further abstractions, but unlike the occurrences of $z$ in the proposition $P$, the occurrences of $z$ in the terms $t_1,...,t_n$ are not bound by the abstraction $\{z \mid (t_1/x_1,...,t_n/x_n)P\}$. In fact, it is easy to prove that the theory allowing nested abstractions binding all the variables is equivalent (see, for instance, [Henkin 1953, Dowek 1995]). Thus, we can consider the construction $\{z \mid P\}$ as a basic term construction, and state the conversion axiom

$$\forall x_1 \ ... \ \forall x_n \ \forall z \ ((z \in \{z \mid P\}) \Leftrightarrow P)$$

### *1.2. Plotkin-Andrews Quotient*

Using a standard proof-search method with such a conversion scheme is rather inefficient. Indeed, if we want to prove, for instance, the proposition

$$2 \in \{x \mid x = 2 \vee x = 3\}$$

using the conversion scheme, we can transform it into the equivalent one

$$2 = 2 \vee 2 = 3$$

and conclude with the axioms of equality. But, going in the wrong direction, we may also transform this proposition into the equivalent one

$$2 \in \{x \mid x \in \{y \mid y = 2\} \vee x = 3\}$$

Thus, using the conversion axiom, a proof search algorithm could spend most of its time uselessly expanding and reducing propositions.

This remark reminds that of G. Plotkin, who noticed in 1972 that with the associativity axiom

$$\forall x \ \forall y \ \forall z \ ((x + y) + z = x + (y + z))$$

a proof search method could spend most of its time uselessly rearranging brackets [Plotkin 1972]. More generally, in any equational theory, a proof search method may spend too much time randomly replacing equals by equals.

The well-know solution proposed by Plotkin is to identify propositions equivalent modulo the congruence generated by the axioms. When there is a confluent and terminating rewrite system such that two propositions are equivalent if and only if they have the same normal forms, normal forms can be chosen as representatives of their equivalence classes in the quotient. For instance, with the associativity axiom, we identify propositions with their normal forms for the rewrite system

$$(x+y)+z \rhd x+(y+z)$$

This way, the associativity axiom is equivalent to the proposition

$$\forall x\ \forall y\ \forall z\ (x+(y+z) = x+(y+z))$$

thus it is a simple consequence of the axioms of equality and it can be dropped. On the other hand, a unifier of two propositions is now a substitution making the propositions equal *modulo the equivalence* (such a problem is called *equational unification*). For instance the propositions

$$a = X + d \quad \text{and} \quad a = b + (c + d)$$

are unifiable (while they are not for the usual notion of unification) because substituting $X$ by $b + c$ yields

$$a = (b+c) + d$$

which reduces to

$$a = b + (c+d)$$

In other words, the associativity axiom is now mixed with the unification algorithm.

A similar program: mixing the conversion axiom and unification algorithm was proposed in 1971 by P.B. Andrews, in the context of type theory: "[First-order resolution] is an elegant combination of substitution and cut [...]. An important open problem concerning resolution in type theory is to find an equally elegant way of combining [substitution], [conversion] and [cut]" [Andrews 1971].

To achieve this goal in naive set theory, we would consider the rewrite system

$$t \in \{z \mid P\} \rhd (t/z)P$$

identify propositions with their normal forms and drop the conversion axiom. A unifier of two propositions would be a substitution making the propositions having the same normal form.

### *1.3. Type Theory*

Unfortunately, naive set theory has several drawbacks: first, as is well-known, it is inconsistent, second: the above rewrite system is not terminating.

Inconsistency is given by Russell's paradox. The proposition "the set of sets that do not belong to themselves belongs to itself"

$$\{x \mid \neg x \in x\} \in \{x \mid \neg x \in x\}$$

is equivalent, by the conversion scheme, to its negation. Thus, both it and its negation are provable. This proposition is also a counter-example to termination, as it rewrites to its negation.

To avoid Russell's paradox, and to get a (hopefully) consistent theory of sets, we can restrict naive set theory in two ways. The first method is to restrict the comprehension scheme to some particular propositions (for instance Zermelo's set theory permits four constructions : pairs, unions, power sets and subsets), the other is to move to a many-sorted theory with a sort (called 0) for atoms a sort (called 1) for sets of atoms, a sort (called 2) for sets of sets of atoms, etc. and allow propositions of the form $t \in_n u$ only when $t$ is of sort $n$ and $u$ of sort $n + 1$ (which permits to construct unions, power sets and subsets but disallows arbitrary pairs). The formalism obtained this way is called *higher-order logic* or *simple type theory.* The original formulation of A.N. Whitehead and B. Russell [Whitehead and Russell 1910-1913, 1925-1927] has been modified by L. Chwistek, F. Ramsey and finally by A. Church [Church 1940].

Although, as remarked by W.V.O. Quine [Quine 1969], the difference between these two methods is rather shallow (as a many-sorted theory can always be relativized as a single-sorted one, and introducing sorts is thus also a way to restrict the comprehension scheme to relativizations of sorted propositions), it is important for automatization.

- First, as some meaningless propositions such as $\mathbb{N} \in \mathbb{N}$ are forbidden by the syntax, they are systematically avoided by the proof search method.
- Then, the rewrite system terminates in higher-order logic and not set theory. Indeed, given a set $A$ and a proposition $P$, set theory allows to define the set $\{z \in A \mid P\}$ of the members of $A$ verifying the proposition $P$, and the rewrite rule associated to this restriction of the comprehension scheme

  $$t \in \{z \in A \mid P\} \rhd t \in A \wedge (t/z)P$$

  does not terminate. A counter-example, which is an adaptation of Russell's paradox is Crabbé's proposition $C$

  $$\{x \in A \mid \neg x \in x\} \in \{x \in A \mid \neg x \in x\}$$

  This proposition rewrites to

  $$\{x \in A \mid \neg x \in x\} \in A \wedge \neg\{x \in A \mid \neg x \in x\} \in \{x \in A \mid \neg x \in x\}$$

  i.e. to $B \wedge \neg C$ where $B$ is the proposition $\{x \in A \mid \neg x \in x\} \in A$.
  Thus, the Plotkin-Andrews quotient cannot be applied, in a simple way, to set theory, while it can be applied to higher-order logic. Equational unification modulo conversion is called *higher-order unification* .
- At last, most proof-search method rely on cut elimination (sometimes taking the form of Herbrand's theorem). Both higher-order logic and set theory introduce more cuts than those already there in first-order logic with no axioms. These cuts can be eliminated in higher-order logic [Takahashi 1967, Prawitz

1968, Girard 1970, Girard 1972] but, Crabbé's counter-example shows that they cannot be eliminated in set theory. Indeed, Crabbé's proposition $C$ rewrites to $B \wedge \neg C$. The proposition $\neg B$ has the following natural deduction proof

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{axiom}
\UnaryInfC{$B, C \vdash B \wedge \neg C$}
\RightLabel{$\wedge$-elim}
\UnaryInfC{$B, C \vdash \neg C$}
\AxiomC{}
\RightLabel{axiom}
\UnaryInfC{$B, C \vdash C$}
\RightLabel{$\neg$-elim}
\BinaryInfC{$B, C \vdash \bot$}
\RightLabel{$\neg$-intro}
\UnaryInfC{$B \vdash \neg C$}
\AxiomC{}
\RightLabel{axiom}
\UnaryInfC{$B \vdash B$}
\AxiomC{}
\RightLabel{axiom}
\UnaryInfC{$B, C \vdash B \wedge \neg C$}
\RightLabel{$\wedge$-elim}
\UnaryInfC{$B, C \vdash \neg C$}
\AxiomC{}
\RightLabel{axiom}
\UnaryInfC{$B, C \vdash C$}
\RightLabel{$\neg$-elim}
\BinaryInfC{$B, C \vdash \bot$}
\RightLabel{$\neg$-intro}
\UnaryInfC{$B \vdash \neg C$}
\RightLabel{$\wedge$-intro}
\BinaryInfC{$B \vdash C$}
\RightLabel{$\neg$-elim}
\BinaryInfC{$B \vdash \bot$}
\RightLabel{$\neg$-intro}
\UnaryInfC{$\vdash \neg B$}
\end{prooftree}
$$

that contains a cut (the negation is first introduces then eliminated). But it is easy to check that this proposition does not have a cut-free proof (notice that after eliminating twice a cut in the proof above we get back the same proof). See [Hallnäs 1983, Ekman 1994] for a more detailed explanation.

### *1.4. Church's Type Theory*

Instead of considering only sets, Church's type theory considers also relations of an arbitrary number of arguments (unary relations are sets, zero-ary relations are naturally identified with propositions). Then, functions are primitive objects and are distinct from their graphs that are relations. Functions of several arguments are curried, for instance the function mapping two numbers $n$ and $m$ to $n + m + 2$ is identified with the function mapping $n$ to the function mapping $m$ to $n + m + 2$. Just as we have a membership symbol $\in$ to build a proposition from two terms in set theory, we have an application symbol $\alpha$ to build a term from two terms. The term $\alpha(t, u)$ is written $(t\ u)$. At last, relations are expressed as functions mapping their arguments to a zero-ary relation i.e. a proposition. For instance, the relation $\leq$ is expressed as the function mapping $a$ and $b$ to the proposition $a \leq b$. Thus, if $A$ is a set (unary relation), the notation $x \in A$ is an abbreviation for $(A\ x)$, i.e. $\alpha(A, x)$.

The sorts of the system are called *simple types*, they contain two base types $\iota$ for atoms and $o$ for zero-ary relations (propositions), and whenever $T$ and $U$ are two types, $T \rightarrow U$ is also a type (the type of functions mapping objects in $T$ to objects in $U$).

As we have a comprehension scheme for sets and relations, we have a comprehension scheme for functions

$$\forall x_1\ ...\ \forall x_n\ \exists f\ \forall y_1\ ...\ \forall y_p\ ((f\ y_1\ ...\ y_p) = t)$$

and an extensionality axiom for functions

$$\forall f\ \forall g\ ((\forall x\ (f\ x) = (g\ x)) \Rightarrow f = g)$$

Skolemizing the comprehension scheme yields an explicit language for functions where the function mapping $y_1, ..., y_p$ to $t$ is written $y_1, ..., y_p \mapsto t$. Again, although the skolemized comprehension scheme provides only such terms when $t$ does not contain Skolem symbols, it is easy to prove that the theory allowing such nested abstractions is equivalent (see, for instance, [Henkin 1953, Dowek 1995]). Then, curried functions of $p$ arguments can be written $y_1 \mapsto (...y_p \mapsto t...)$. Following Church's original notation, the term $x \mapsto t$ is written $\lambda x\ t$. The conversion axiom, called *$\beta$-conversion* axiom is then stated

$$\forall x\ \forall y_1\ ...\ \forall y_p\ ((\lambda x\ t)\ x) = t$$

Notice that the functional comprehension scheme, asserts the existence of very few functions. For instance, if the atoms are taken to be natural numbers and the language contains a symbol 0 of type $\iota$ and $S$ of type $\iota \to \iota$, for functions of type $\iota \to \iota$ the conversion scheme only asserts the existence of the constant functions and the functions adding a constant to their argument. Similarly, these functions are the only ones that are explicitly definable by a term (e.g. $\lambda x\ S(S(0))$ and $\lambda x\ S(S(x))$). For instance, the comprehension scheme does not assert the existence of the function $\chi$ mapping 0 to 0 and the other natural numbers to 1. In contrast, the graph of this function (which is a binary relation, i.e. an object of type $\iota \to \iota \to o$) can easily be defined $G = \lambda x\ \lambda y\ ((x = 0 \wedge y = 0) \vee (\neg(x = 0) \wedge (y = 1)))$. This motivates the introduction of another axiom: *the descriptions axiom*

$$\exists D\ \forall x\ ((\exists_1 y\ (x\ y)) \Rightarrow (x\ (D\ x)))$$

where $\exists_1 y\ P(y)$ is a proposition expressing the existence and unicity of an object verifying the property $P$, i.e. the proposition

$$\exists y\ P(y) \wedge \forall y_1\ \forall y_2\ ((P(y_1) \wedge P(y_2)) \Rightarrow y_1 = y_2)$$

When skolemizing this axiom, we introduce a Skolem symbol called the *descriptions operator* and the axiom

$$\forall x\ (\exists_1 y\ (x\ y)) \Rightarrow (x\ (D\ x))$$

This descriptions operator, that picks the element in every one element set, can be extended to a *choice operator* (also called Hilbert's $\varepsilon$ operator, or Bourbaki's $\tau$ operator) that picks an element in any nonempty set. In this case the axiom is rephrased

$$\forall x\ (\exists y\ (x\ y)) \Rightarrow (x\ (D\ x))$$

and it is a form of the axiom of choice.

The descriptions operator and axiom permit to relate the functional relations and the functions. The function $\chi$ above can be defined by the term $\lambda x\ (D\ (\lambda y\ (G\ x\ y)))$. Then we can prove, for instance that $\chi(2) = 1$. Notice however that this theorem is not a consequence of the conversion axiom alone, the descriptions axiom is also needed.

When searching for proofs in higher-order logic, we transform the $\beta$-conversion axiom in a rewrite rule called *$\beta$-reduction*

$$((\lambda x\ t)\ u) \rhd (u/x)t$$

and we also take another rewrite rule called *η-reduction* which is a consequence of the extensionality axiom

$$\lambda x\ (t\ x) \rhd t \text{ provided } x \text{ does not appear free in } t$$

The extensionality axiom itself and the descriptions axiom remain as axioms of the theory.

1.1. Remark. In the presentation above, when we want to substitute the predicate $Q(.,.)$ for the variable $P$ in the proposition $P(a) \wedge P(b)$ to get the proposition $Q(a,a) \wedge Q(b,b)$, we first construct a term $\lambda z\ Q(z,z)$, then we substitute it for the variable $P$ yielding $(\lambda z\ Q(z,z))(a) \wedge (\lambda z\ Q(z,z))(b)$ and at last we prove that this term is equivalent to $Q(a,a) \wedge Q(b,b)$, or we reduce it to $Q(a,a) \wedge Q(b,b)$.

An alternative, frequently used in second-order logic [Church 1956, Goldfarb 1981, Farmer 1988, Krivine 1993], is to a define a substitution operation $(Q(x,x)/P(x))A$ in such a way that $(Q(x,x)/P(x))(P(a) \wedge P(b))$ is $Q(a,a) \wedge Q(b,b)$. This way the reduction is included in the definition of substitution.

### *1.5. Equational Higher-order Unification*

Higher-order unification is equational unification modulo $\beta\eta$-equivalence. As remarked above, as the function $\chi$ is defined with the descriptions operator, the proposition $\chi(2) = 1$ needs the descriptions axiom to be proved and the term $\chi(2)$ does not reduce to the term 1. Thus, we may want to extend the rewrite system above, for instance with rules

$$\chi(0) \rhd 0$$

$$\chi(S(x)) \rhd S(0)$$

In the same way, we may want to add rewrite rules for addition (which is also defined using the descriptions operator)

$$(+\ 0\ y) \rhd y$$

$$(+\ (S\ x)\ y) \rhd (S\ (+\ x\ y))$$

A rather general extension is to consider rewrite rules for the recursor $R$ (which is also defined using the descriptions operator [Andrews 1986])

$$(R\ x\ f\ 0) \rhd x$$

$$(R\ x\ f\ (S\ y)) \rhd (f\ y\ (R\ x\ y))$$

This rewrite system is called *Gödel system T* [Gödel 1958, Girard, Lafont and Taylor 1989].

Equational unification modulo a rewrite system containing $\beta$, $\eta$ and other rules like the ones above is called *equational higher-order unification*.

### *1.6. Expectations and Achievements*

Initiated in the sixties, the search for an automated theorem proving method for higher-order logic was motivated by big expectations. "Providing a genuinely useful mathematical service" is one of the goals mentioned in Robinson's quotation above (although this quotation is still moderated for the sixties). With the passing of time, we know that fully automated theorem proving methods have not, or very rarely, permitted to solve really difficult mathematical problems.

On the other hand, automated theorem proving methods have found other fields where they have provided genuinely useful services (logic programming, deductive data bases, etc.). The major applications of proof search in higher-order logic are higher-order logic programming and logical frameworks ($\lambda$-Prolog [Nadathur and Miller 1998], Elf [Pfenning 1991*a*], Isabelle [Paulson 1991], etc., see also [Pfenning 2001], Chapter 17 of this Handbook) and tools to prove easy but cumbersome lemmas in interactive proof construction systems, see [Barendregt and Geuvers 2001] (Chapter 18 of this Handbook).

Besides automated theorem proving, higher-order unification has also been used to design of type reconstruction algorithms for some programming languages [Pfenning 1988], in computational linguistics [Miller and Nadathur 1986, Dalrymple, Shieber and Pereira 1991], program transformation [Huet and Lang 1978, Hannan and Miller 1988, Hagiya 1990], higher-order rewriting [Nipkow 1991, Nipkow and Prehofer 1998, Mayr and Nipkow 1998], proof theory [Parikh 1973, Farmer 1991*b*], etc.

## 2. Simply Typed $\lambda$-calculus

In this section, we give the definitions and elementary properties of simply typed $\lambda$-calculus which is the term-language of higher-order logic. The proofs of these properties can be found in [Barendregt 1984, Hindley and Seldin 1986, Krivine 1993].

### *2.1. Types*

We consider a finite set whose elements are called *atomic types*.

2.1. DEFINITION. (Types)
Types are inductively defined by:

- atomic types are types,
- if $T$ and $U$ are types then $T \to U$ is a type.

**Notation** The expression $T_1 \to T_2 \to ... \to T_n \to U$ is a notation for the type $T_1 \to (T_2 \to ... \to (T_n \to U)...)$.

2.2. Definition. (Size of a type)
The size of a type is defined as follows

- $|T| = 1$ if $T$ is atomic,
- $|T \rightarrow U| = |T| + |U|$.

2.3. Definition. (Order of a type)
If $T$ is a type, the *order* of $T$ is defined by:

- $o(T) = 1$ if $T$ is atomic,
- $o(T_1 \rightarrow T_2) = max\{1 + o(T_1), o(T_2)\}$.

### *2.2. Terms*

We consider a finite set of constants, to each constant is assigned a type. We assume that we have at least one constant in each atomic type. This assumption corresponds to the fact that we do not allow empty types.

For each type, we consider an infinite set of variables. Two different types have disjoint sets of variables.

2.4. Definition. ($\lambda$-terms)
$\lambda$-terms are inductively defined by:

- constants are terms,
- variables are terms,
- if $t$ and $u$ are two terms then $(t\ u)$ (i.e. $\alpha(t, u)$) is a term,
- if $x$ is a variable and $t$ a term then $\lambda x\ t$ is a term.

**Notation** The expression $(u\ v_1\ ...\ v_n)$ is a notation for the term $(...(u\ v_1)\ ...\ v_n)$.

2.5. Definition. (Size of a term)
The size of a term is defined as follows

- $|x| = |c| = 1$,
- $|(u\ v)| = |u| + |v|$,
- $|\lambda x\ u| = |u|$.

2.6. Definition. (Type of a term)
A term $t$ is said to have the type $T$ if either:

- $t$ is a constant of type $T$,
- $t$ is a variable of type $T$,
- $t = (u\ v)$ and $u$ has type $U \rightarrow T$ and $v$ has type $U$ for some type $U$,
- $t = \lambda x\ u$, the variable $x$ has type $U$, the term $u$ has type $V$ and $T = U \rightarrow V$.

A term $t$ is said to be *well-typed* if there exists a type $T$ such that $t$ has type $T$. In this case $T$ is unique and is called *the type of* $t$.

2.7. REMARK. In this chapter, we use an *explicitly typed $\lambda$-calculus*. For instance, the term $\lambda x\ x$ has a single type $T \to T$ where $T$ is the type of the variable $x$. We could alternatively have used a *type assignment system*, with a single class of variables and rules assigning types to terms, for instance any type of the form $T \to T$ to the term $\lambda x\ x$.

In the rest of this chapter we consider only well-typed terms.

### 2.3. Substitution

2.8. DEFINITION. (Variables, free variables)

Let $t$ be a term, the set $Var(t)$ is the set of all variables occurring in $t$, it is defined by induction on the structure of $t$ by:

- $Var(c) = \emptyset$,
- $Var(x) = \{x\}$,
- $Var((t\ u)) = Var(t) \cup Var(u)$,
- $Var(\lambda x\ t) = Var(t) \cup \{x\}$.

In contrast, the set $FVar(t)$ is the set of the variables occurring freely in $t$, it is defined by induction on the structure of $t$ by:

- $FVar(c) = \emptyset$,
- $FVar(x) = \{x\}$,
- $FVar((t\ u)) = FVar(t) \cup FVar(u)$,
- $FVar(\lambda x\ t) = FVar(t) \setminus \{x\}$.

A term with no free variables is called a *closed term*.

2.9. EXAMPLE. The variable $x$ occurs in the terms $\lambda x\ x$, $\lambda x\ y$ and $\lambda y\ x$, but it occurs freely only in the third of these terms.

2.10. DEFINITION. (Substitution)

A *substitution* is a finite set of pairs $\{\langle x_1, t_1\rangle, ..., \langle x_n, t_n\rangle\}$ where for each $i$, $x_i$ is a variable and $t_i$ a term of the same type and such that if $\langle x, t\rangle$ and $\langle x, t'\rangle$ are both in this set then $t = t'$. Such a substitution is written $t_1/x_1, ..., t_n/x_n$.

We now want to define the operation of substituting the terms $t_1, ..., t_n$ for the variables $x_1, ..., x_n$ in a term $u$. A first attempt leads to the following definition.

2.11. DEFINITION. (Replacement in a term)

If $\theta = t_1/x_1, ..., t_n/x_n$ is a substitution and $t$ a term, then the term $\langle\theta\rangle t$ is defined as follows

- $\langle\theta\rangle c = c$,
- $\langle\theta\rangle x_i = t_i$ and $\langle\theta\rangle x = x$, if $x$ is a variable not among $x_1, ..., x_n$,
- $\langle\theta\rangle(t\ u) = (\langle\theta\rangle t\ \langle\theta\rangle u)$,
- $\langle\theta\rangle(\lambda x\ t) = \lambda x\ \langle\theta\rangle t$,

This notion of replacement has two drawbacks: first if we replace the variable $x$ by the variable $y$ in the term $\lambda x\ x$ we get the term $\lambda x\ y$, while the variable $x$ is bound in the term $\lambda x\ x$, and thus we would rather expect the term $\lambda x\ x$. Then if we replace the variable $x$ by the variable $y$ in the term $\lambda y\ x$ we get the term $\lambda y\ y$ where the variable $y$ has been captured, we would rather want to get the term $\lambda z\ y$.

Nevertheless, this notion of replacement is useful in some situations. For instance, it will be used in section 5.3 and it is also useful to define the notion of equivalence of two terms modulo bound variable renaming ($\alpha$-equivalence).

2.12. Definition. ($\alpha$-equivalence)

The $\alpha$-equivalence of two terms is inductively defined by:

- $c \equiv c$,
- $x \equiv x$,
- $(t\ u) \equiv (t'\ u')$ if $t \equiv t'$ and $u \equiv u'$,
- $\lambda x\ t \equiv \lambda y\ u$ if $\langle z/x \rangle t \equiv \langle z/y \rangle u$ for some variable $z$ different from $x$ and $y$ and occurring neither in $t$ nor in $u$.

2.13. Example. The terms $\lambda x\ x$ and $\lambda y\ y$ are $\alpha$-equivalent.

2.14. Proposition. *$\alpha$-equivalence is an equivalence relation. Moreover, the operations on terms (application and abstraction with respect to a variable) are compatible with this relation. Thus, they are defined on equivalence classes.*

In the following we shall identify $\alpha$-equivalent terms, i.e. consider terms as representatives of their $\alpha$-equivalence class.

Now, we can define the substitution operation. To avoid capture, when we substitute the variable $y$ for the variable $x$ in the term $\lambda y\ x$, we need to rename the bound variable $y$, and get for instance the term $\lambda z\ y$. The choice of the variable $z$ is purely arbitrary, thus the substitution operation is in fact defined on terms modulo $\alpha$-equivalence, i.e. on $\alpha$-equivalence classes.

2.15. Definition. (Substitution in a term)

If $\theta = t_1/x_1, ..., t_n/x_n$ is a substitution and $t$ a term then the term $\theta t$ is defined as follows

- $\theta c = c$,
- $\theta x_i = t_i$ and $\theta x = x$, if $x$ is a variable not among $x_1, ..., x_n$,
- $\theta(t\ u) = (\theta t\ \theta u)$,
- $\theta(\lambda x\ u) = \lambda y\ \theta(y/x)u$, where $y$ is a fresh variable, with the same type as $x$, i.e. a variable that does not occur in $t$ nor in $t_1, ..., t_n$ and is different from $x_1, ..., x_n$.

2.16. Proposition. *If $t$ is a term of type $T$, $x$ is a variable of type $U$ and $u$ a term of type $U$ then the term $(u/x)t$ has type $T$.*

2.17. Definition. (Composition of substitutions)
Let $\theta$ and $\theta'$ two substitutions and $x_1, ..., x_n$ be the variables bound by one substitution or the other. We let

$$\theta \circ \theta' = \theta\theta' x_1/x_1, ..., \theta\theta' x_n/x_n$$

2.18. Definition. (More general)
A substitution $\theta_1$ is said to be *more general* than a substitution $\theta_2$ ($\theta_1 \leq \theta_2$) if there exists a substitution $\eta$ such that $\theta_2 = \eta \circ \theta_1$.

2.19. Definition. (Size of a substitution)
The size of a substitution $t_1/x_1, ..., t_n/x_n$ is defined as

$$|t_1/x_1, ..., t_n/x_n| = |t_1| + ... + |t_n|$$

### *2.4. Reduction*

2.20. Definition. ($\beta\eta$-reduction)
The $\beta\eta$-reduction (in one step), written $\rhd$, is inductively defined by:

- $\beta$: $((\lambda x\ t)\ u) \rhd (u/x)t$ ,
- $\eta$: $\lambda x\ (t\ x) \rhd t$ if $x$ is not free in $t$,
- $\mu$: if $u \rhd u'$ then $(t\ u) \rhd (t\ u')$,
- $\nu$: if $t \rhd t'$ then $(t\ u) \rhd (t'\ u)$,
- $\xi$: if $t \rhd t'$ then $\lambda x\ t \rhd \lambda x\ t'$.

The $\beta\eta$-reduction (in several steps), written $\rhd^*$, is the reflexive-transitive closure of the relation $\rhd$, it is inductively defined by:

- if $t \rhd u$, then $t \rhd^* u$,
- $t \rhd^* t$,
- $t \rhd^* u$ and $u \rhd^* v$ then $t \rhd^* v$.

2.21. Proposition. *If a term $t$ has type $T$ and $t$ reduces to $u$ then $u$ has type $T$.*

2.22. Proposition. *Substitution and reduction commute, i.e. if $t \rhd^* u$ then $(v/x)t \rhd^* (v/x)u$.*

2.23. Theorem. *The $\beta\eta$-reduction relation is strongly normalizable and confluent on typed terms, and thus each term has a unique $\beta\eta$-normal form modulo $\alpha$-conversion.*

2.24. Proposition. *Let $t$ be a normal well-typed term of type $T_1 \to ... \to T_n \to U$ ($U$ atomic), the term $t$ has the form*

$$t = \lambda x_1\ ...\ \lambda x_m\ (y\ u_1\ ...\ u_p)$$

*where $m \leq n$ and $y$ is a constant or a variable.*

*The symbol $y$ is called the* head symbol *of the term.*

2.25. Definition. (Long normal form) If $t = \lambda x_1 \ ... \ \lambda x_m \ (y \ u_1 \ ... \ u_p)$ is a normal term of type $T_1 \to ... \to T_n \to U$ ($U$ atomic) ($m \leq n$) then its long normal form is the term

$$t' = \lambda x_1 \ ... \ \lambda x_m \ \lambda x_{m+1} \ ... \ \lambda x_n \ (y \ u'_1 \ ... \ u'_p \ x'_{m+1} \ ... \ x'_n)$$

where $u'_i$ is the long normal form of $u_i$ and $x'_i$ is the long normal form of $x_i$.

This definition is made by induction on a pair whose first component is the size of the term and the second the size of its type.

The long normal form of an arbitrary term is that of its normal form.

2.26. Remark. The $\beta$-normal form of a term is its normal form for the following rewrite system which is also strongly normalizing and confluent.

- $\beta$: $((\lambda x \ t) \ u) \rhd_\beta (u/x)t$,
- $\mu$: if $u \rhd_\beta u'$ then $(t \ u) \rhd_\beta (t \ u')$,
- $\nu$: if $t \rhd_\beta t'$ then $(t \ u) \rhd_\beta (t' \ u)$,
- $\xi$: if $t \rhd_\beta t'$ then $\lambda x \ t \rhd_\beta \lambda x \ t'$.

Because $\eta$-reduction can be delayed with respect to $\beta$-reduction, the long normal form of a term is also that of its $\beta$-normal form. Thus to compute the long normal form of a term, we do not need to perform $\eta$-reductions.

2.27. Remark. Two terms have the same long normal form if and only if they have the same genuine normal form. Thus, as representatives of classes of terms we can either chose the genuine short normal form or the long normal form. Choosing the long one simplifies many problems. So in the rest of this chapter, “normal form” will always mean “long normal form”.

### *2.5. Unification*

2.28. Definition. (Unification problem, Unifier)

An *equation* is a pair of terms $t, u$. A *unification problem* is a finite set of equations. A *solution* or a *unifier* of such a problem is a substitution $\theta$ such that for each pair $t, u$ of the problem, the terms $\theta t$ and $\theta u$ have the same normal form.

2.29. Definition. (Minimal unifier, Most general unifier)

A unifier $\theta$ of a problem is said to be *minimal* if all the unifiers of the problem more general than $\theta$ are renamings of $\theta$, i.e. substitutions of the form $\eta \circ \theta$ with $\eta = y_1/x_1, ..., y_n/x_n$.

A unifier of a problem is said to be the *smallest* or the *the most general unifier* if it is more general than all the unifiers.

## 3. Undecidability

### *3.1. Higher-order Unification*

In this section, we show that higher-order unification is undecidable, i.e. there is no algorithm that takes as argument a unification problem and answers if it has a solution or not. To achieve this goal, we reduce another undecidable problem: Hilbert's tenth problem.

3.1. THEOREM. *(Matiyacevich-Robinson-Davis [Matiyacevich 1970, Davis 1973]) Hilbert's tenth problem is undecidable, i.e. there is no algorithm that takes as arguments two polynomials $P(X_1, ..., X_n)$ and $Q(X_1, ..., X_n)$ whose coefficients are natural numbers and answers if there exists natural numbers $m_1, ..., m_n$ such that*

$$P(m_1, ..., m_n) = Q(m_1, ..., m_n)$$

We have seen that very few functions can be expressed in simply typed $\lambda$-calculus alone. With Peano numbers (i.e. with a symbol 0 of type $\iota$ and $S$ of type $\iota \to \iota$), we can only define the constant functions and the functions adding a constant to one of their arguments. The descriptions operator is needed to define addition and multiplication and thus polynomials. Nevertheless, we can use another definition of natural numbers: Church numbers.

3.2. DEFINITION. (Church numbers)
With each natural number $n$, we associate its *Church number*

$$\overline{n} = \lambda x\ \lambda f\ (f\ (...(f\ x)...))$$

with $n$ occurrences of the symbol $f$. This term has type $\iota \to (\iota \to \iota) \to \iota$.

Moving from Peano numbers to Church numbers increases only slightly the set of functions that can be expressed in simply typed $\lambda$-calculus: as proved by H. Schwichtenberg [Schwichtenberg 1976], the expressible functions are the so-called *extended polynomials*, i.e. the polynomials extended by the characteristic functions of $\{0\}$ and $\mathbb{N}\setminus\{0\}$ (for instance the function mapping $n$ to $2^n$ still needs the descriptions operator). But polynomials are precisely what are needed to reduce Hilbert's tenth problem.

3.3. PROPOSITION. *Consider the terms*

$$add = \lambda n\ \lambda m\ \lambda x\ \lambda f\ (n\ (m\ x\ f)\ f)$$

$$mult = \lambda n\ \lambda m\ \lambda x\ \lambda f\ (n\ x\ (\lambda z\ (m\ z\ f)))$$

*The normal form of the term $(add\ \overline{n}\ \overline{m})$ is $\overline{n+m}$. The normal form of the term $(mult\ \overline{n}\ \overline{m})$ is $\overline{n \times m}$. Thus for every polynomial $P$ there exists a $\lambda$-term $p$ such that the normal form of the term $(p\ \overline{m_1}\ ...\ \overline{m_n})$ is the term $\overline{P(m_1, ..., m_n)}$.*

Obviously, if the polynomial equation

$$P(X_1, ..., X_n) = Q(X_1, ..., X_n)$$

has a solution $m_1, ..., m_n$ then the substitution $\overline{m_1}/X_1, ..., \overline{m_n}/X_n$ is a solution to the unification problem

$$(p\ X_1\ ...\ X_n) = (q\ X_1\ ...\ X_n)$$

The converse of this proposition is not obvious, indeed, there are terms of type $\iota \rightarrow (\iota \rightarrow \iota) \rightarrow \iota$, for instance variables, that are not Church numbers.

Thus we shall add more equations to the problem to force the solutions to be Church numbers.

3.4. Proposition. *A normal term $t$ of type $\iota \rightarrow (\iota \rightarrow \iota) \rightarrow \iota$ is a Church number if and only if $t/X$ is a solution of the equation*

$$\lambda z\ (X\ z\ (\lambda y\ y)) = \lambda z\ z$$

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

Thus we can conclude.

3.5. Theorem. *There is no algorithm that takes as argument a unification problem and answers if it has a solution or not.*

Proof. With each polynomial equation

$$P(X_1, ..., X_n) = Q(X_1, ..., X_n)$$

we associate the unification problem

$$(p\ X_1\ ...\ X_n) = (q\ X_1\ ...\ X_n)$$

$$\lambda z\ (X_1\ z\ (\lambda y\ y)) = \lambda z\ z$$

$$...$$

$$\lambda z\ (X_n\ z\ (\lambda y\ y)) = \lambda z\ z$$

where $p$ is the term expressing the polynomial $P$ and $q$ the term expressing the polynomial $Q$.

If the polynomial equation has a solution $m_1, ..., m_n$ then the substitution $\overline{m_1}/X_1, ..., \overline{m_n}/X_n$ is a solution of the unification problem. Conversely, if the unification problem has a solution $c_1/X_1, ..., c_n/X_n$ then the normal form of each $c_i$ is a Church number $c_i = \overline{m_i}$. The natural numbers $m_1, ..., m_n$ are a solution to the polynomial equation. □

3.6. Remark. This theorem has been proved independently in 1972 by G. Huet [Huet 1972, Huet 1973*b*] and C.L. Lucchesi [Lucchesi 1972]. The original proofs did not reduce Hilbert's tenth problem, but Post's correspondence problem.

The unification problems built when reducing Post's correspondence problem have the property that the variables free in the problem are applied only to terms that do not contain further variables free in the problem. Thus, reducing Post's correspondence problem permits to sharpen the theorem and prove that there is no algorithm that takes as argument a unification problem of this special form and answers if it has a solution or not. Such a sharpened undecidability theorem is useful, for instance to prove the undecidability of type reconstruction in the extension of simply typed $\lambda$-calculus with dependent types [Dowek 1993*c*].

### *3.2. Second-order Unification*

But, reducing Hilbert's tenth problem, we can sharpen the result in another direction. The variables $X_1, ..., X_n$ above have the type $\iota \to (\iota \to \iota) \to \iota$. This is the type of functionals taking in arguments an atom of type $\iota$ and a function of type $\iota \to \iota$. Using the definition 2.3, this type has order 3. We may try to sharpen the result and allow only variables of order 2 or even 1. If we restrict the types of the free variables to be first-order, the problem is just a variant of first-order unification and thus it is decidable. If we restrict the types of the free variables to be at most second-order, we get *second-order unification*, which has been proved to be undecidable by W.D. Goldfarb [Goldfarb 1981].

Goldfarb's proof relies on a expression of numbers that is a degeneracy of Church's. Taking $\lambda x\ \lambda f\ (f\ (...(f\ x)...))$ leads to a third-order type, thus the idea is to drop the abstraction on $f$ and to take $\lambda x\ (f\ (...(f\ x)...))$ where $f$ is a constant of type $\iota \to \iota$. Thus numbers now have the second-order type $\iota \to \iota$. More precisely, Goldfarb number $\overline{n}$ is the term $\lambda x\ (g\ a\ (...(g\ a\ x)...))$ where $a$ and $g$ are constants of type $\iota$ and $\iota \to \iota \to \iota$.

Goldfarb numbers can still be characterized by a unification problem.

3.7. Proposition. *A normal term $t$ of type $\iota \to \iota$ is a Goldfarb number if and only if $t/X$ is a solution to the equation*

$$(g\ a\ (X\ a)) = (X\ (g\ a\ a))$$

Addition can still be expressed by the term

$$add = \lambda n\ \lambda m\ \lambda x\ (n\ (m\ x))$$

but multiplication cannot be expressed this way. Thus, it will be expressed, not by a term, but by a unification problem.

3.8. Proposition. *(Goldfarb's lemma) The unification problem*

$$(Y\ a\ b\ (g\ (g\ (X_3\ a)\ (X_2\ b))\ a)) = (g\ (g\ a\ b)\ (Y\ (X_1\ a)\ (g\ a\ b)\ a))$$

$$(Y\ b\ a\ (g\ (g\ (X_3\ b)\ (X_2\ a))\ a)) = (g\ (g\ b\ a)\ (Y\ (X_1\ b)\ (g\ a\ a)\ a))$$

*has a solution* $\overline{m_1}/X_1, \overline{m_2}/X_2, \overline{m_3}/X_3, u/Y$ *if and only if* $m_1 \times m_2 = m_3$.

3.9. Theorem. *(Goldfarb) Second-order unification is undecidable.*

Proof. By reduction of Hilbert's tenth problem. Every equation of the form

$$P(X_1, ..., X_n) = Q(X_1, ..., X_n)$$

can be decomposed into a system of equations of the form

$$X_i + X_j = X_k$$

$$X_i \times X_j = X_k$$

$$X_i = p$$

With such a system we associate a unification problem containing:

- for each variable $X_i$, an equation as in proposition 3.7,
- for each equation of the form $X_i + X_j = X_k$, the equation $(add\ X_i\ X_j) = X_k$,
- for each equation of the form $X_i \times X_j = X_k$, two equations as in proposition 3.8,
- for each equation of the form $X_i = p$, the equation $X_i = \overline{p}$.

□

3.10. Remark. Goldfarb's result has been sharpened by W.M. Farmer [Farmer 1991*a*], J. Levy and M. Veanes [Levy and Veanes 1998] who study the number of variables, the number of variable occurrences and the arity of the variables that are needed to get undecidability.

3.11. Remark. Reducing Hilbert's tenth problem is a powerful tool, but as the proof of undecidability of Hilbert's tenth problem itself is rather complicated, one may want to find a simpler undecidability proof, i.e. one reducing a problem that is simpler to prove undecidable (the halting problem, the semi-Thue problem, Post's correspondence problem, etc.). We have seen that such reductions are possible for third-order unification.

A. Schubert [Schubert 1998] has given another undecidability proof for second-order unification, reducing the halting problem of a two-counter automaton. This proof permits also to sharpen the result proving that there is no algorithm that takes as argument a unification problem where the variables free in the problem are applied only to terms that do not contain further variables free in the problem and answers if it has a solution or not. This sharpened undecidability theorem is applied to prove the undecidability of type reconstruction in some extension of simply typed $\lambda$-calculus with polymorphic types [Schubert 1998].

## 4. Huet's Algorithm

Higher-order unification is undecidable, but it is semi-decidable, i.e. we can build an algorithm that takes a unification problem as argument, terminates and returns a solution when the problem has one, but may loop forever when it does not. Indeed, given a problem and a substitution, it is possible to decide whether the substitution is a solution of the problem or not: it suffices to apply the substitution to both members of each equation, normalize the terms and check that their normal forms are equal. Thus, a naive generate and test algorithm terminates if the problem has a solution.

Such a generate and test algorithm is, of course, of no practical use. But as we shall see, it can be gradually improved so that we reach an algorithm that finds a solution rather quickly when such a solution exists and reports failure in many cases where the equation has no solutions (of course, not all of them, since the problem is undecidable).

### *4.1. A "Generate and Test" Algorithm*

#### *4.1.1. Generating Long Normal Closed Terms*

Recall that we have assumed in section 2.2 that we had a constant $c_U$ in each atomic type $U$. Thus, in a type $T_1 \to ... \to T_p \to U$, we always have a closed term $\lambda y_1 \, ... \, \lambda y_p \; c_U$. Obviously, if a substitution $t_1/X_1, ..., t_n/X_n$ is a solution to a problem then the substitution obtained by substituting each free variable of $t_1, ..., t_n$ by the term $\lambda y_1 \, ... \, \lambda y_p \; c_U$ corresponding to its type is also a solution. Moreover, this solution is closed, i.e. all the terms substituted to the variables $X_1, ..., X_n$ are closed terms. Thus, if a problem has a solution, it has also a closed solution and instead of enumerating all the terms $t_1, ..., t_n$ to be substituted for the variables $X_1, ..., X_n$ we can restrict to the closed ones. Similarly, if a substitution $t_1/X_1, ..., t_n/X_n$ is a solution to a problem then the substitution obtained by taking the long normal form of the terms $t_1, ..., t_n$ is also a solution. Thus, we can restrict the enumeration the long normal closed terms.

Using definition 2.25 long normal closed terms of type $T_1 \to ... \to T_p \to U$, where $U$ is an atomic type, have the shape

$$\lambda y_1 \; ... \; \lambda y_p \; (h \; u_1 \; ... \; u_r)$$

where $y_1, ..., y_p$ are variables of type $T_1, ..., T_p$, the head symbol $h$ is either one of the variables $y_1, ..., y_p$ or a constant, and $u_1, ..., u_r$ are terms whose number and type depend on the type of the symbol $h$.

Thus a method to enumerate all the normal terms of a given type is to proceed step by step, enumerating all the possible head symbols of the term and then using recursively the same method to enumerate the terms $u_1, ..., u_r$.

A first, but naive, idea would be to use variables $H_1, ..., H_r$ to hold for the terms $u_1, ..., u_r$, i.e. to consider the term

$$\lambda y_1 \; ... \; \lambda y_p \; (h \; H_1 \; ... \; H_r)$$

and then to enumerate the terms to be substituted for the variables $H_1, ..., H_r$. Unfortunately, such an idea does not work, because the variables $y_1, ..., y_p$ may occur in the terms $u_1, ..., u_r$, substituting such terms to the variables $H_1, ..., H_r$ would introduce captures, and substitution renames bound variables to avoid such captures. Thus, for instance, such a method would not generate the term $\lambda x\ \lambda f\ (f\ x)$ of type $\iota \to (\iota \to \iota) \to \iota$: a first step would consider the term $\lambda x\ \lambda f\ (f\ H)$ but then substituting the term $x$ to the variable $H$ would yield the term $\lambda y\ \lambda f\ (f\ x)$ and not $\lambda x\ \lambda f\ (f\ x)$.

A solution to this problem is to express functionally the dependence of the terms $u_1, ..., u_r$ with respect to the variables $y_1, ..., y_p$, considering the term

$$\lambda y_1\ ...\ \lambda y_p\ (h\ (H_1\ y_1\ ...\ y_p)\ ...\ (H_r\ y_1\ ...\ y_p))$$

Then the term $\lambda x\ \lambda f\ (f\ x)$ is generated in two steps: first, we generate the term $\lambda x\ \lambda f\ (f\ (H\ x\ f))$ then we substitute the term $\lambda x\ \lambda f\ x$ for the variable $H$.

A substitution of the form

$$\lambda y_1\ ...\ \lambda y_p\ (h\ (H_1\ y_1\ ...\ y_p)\ ...\ (H_r\ y_1\ ...\ y_p))/X$$

is called an *elementary substitution*.

4.1. Definition. We consider the following inference system

$$\frac{t}{(\lambda y_1\ ...\ \lambda y_p\ (h\ (H_1\ y_1\ ...\ y_p)\ ...\ (H_r\ y_1\ ...\ y_p))/X)t}$$

where the terms are normal (i.e. $(\lambda y_1\ ...\ \lambda y_p\ (h\ (H_1\ y_1\ ...\ y_p)\ ...\ (H_r\ y_1\ ...\ y_p))/X)t$ actually stands for the long normal form of this term), $X$ is a free variable of $t$ of type $T_1 \to ... \to T_p \to U$ ($U$ atomic), $h$ is one of the variables $y_1, ..., y_p$ or a constant, the target type of $h$ is $U$ and the variables $H_1, ..., H_r$ are fresh variables of the appropriate type.

4.2. Proposition. *All the long normal closed terms of type $T$ are produced from a variable of type $T$ by the inference system above.*

Proof. By induction on the size of $t$. □

4.3. Remark. The inference system above has two forms of non-determinism: first the choice of the variable $X$ of the term $t$ to be substituted, then the choice of the head symbol $h$ in the substituted term. The choice of the variable $X$ is a *don't care* non-determinism, the choice of the head symbol $h$ is a *don't know* non-determinism.

This *don't know* non-determinism can be handled by building a search tree as follows. Nodes are labeled by terms and leaves by closed terms. In each internal node, we chose a variable and we draw an edge corresponding to each possible head symbol.

Another solution is to consider an inference system defined on finite sets of terms, deriving from $A \cup \{t\}$, the set $A \cup \{\sigma_1 t, ..., \sigma_n t\}$ where $\sigma_1, ..., \sigma_n$ are the elementary substitutions corresponding to the different possible head symbols.

### *4.1.2. A Unification Algorithm*

4.4. Definition. (Generate and test algorithm)

We consider the inference system

$$\frac{E}{(\lambda y_1 \; ... \; \lambda y_p \; (h \; (H_1 \; y_1 \; ... \; y_p) \; ... \; (H_r \; y_1 \; ... \; y_p))/X)E}$$

where $E$ is a unification problem (i.e. a finite set of equations), $X$ is a free variable of $E$ of type $T_1 \to ... \to T_p \to U$ ($U$ atomic) and $h$ is one of the variables $y_1, ..., y_p$ or a constant and the variables $H_1, ..., H_r$ are fresh variables of the appropriate type.

4.5. Proposition. *The above algorithm is sound and complete, i.e. a problem has a solution if and only if a trivial problem (i.e. a problem where each equation relates identical terms) can be derived from it.*

Proof. The soundness property is an obvious induction on the length of the derivation. The completeness property is proved by induction on the size of a long normal closed solution. □

## *4.2. Huet's Algorithm*

In the generate and test algorithm, the unification problem is completely passive, it is only used to test if a given substitution is a solution or not. In Huet's algorithm it is used in a much more active way to restrict the search space.

For instance, consider the problem $0 = S(X)$, whatever closed term we may substitute for $X$, we will get two terms which have a different head symbol and thus are different. Similarly the problem $S(u) = S(v)$ can be simplified into the problem $u = v$ that has the same solutions. Such a term where the head symbol is a constant or a bound variable and thus cannot be changed by a substitution is called *rigid*.

4.6. Definition. (Rigid, flexible term)

A term is said to be *rigid* if its head symbol is a constant or a bound variable, it is said to be *flexible* if its head symbol is a free variable.

### *4.2.1. Rigid-rigid Equations*

A first improvement that can be made to the generate and test algorithm is to simplify problems using the rules

$$\frac{E \cup \{\lambda x_1 \; ... \; \lambda x_n \; (f \; u_1 \; ... \; u_p) = \lambda x_1 \; ... \; \lambda x_n \; (g \; v_1 \; ... \; v_q)\}}{\bot} \; \textit{Fail}$$

$$\frac{E \cup \{\lambda x_1 \; ... \; \lambda x_n \; u_1 = \lambda x_1 \; ... \; \lambda x_n \; v_1, ..., \lambda x_1 \; ... \; \lambda x_n \; u_p = \lambda x_1 \; ... \; \lambda x_n \; v_p\}}{E \cup \{\lambda x_1 \; ... \; \lambda x_n \; (f \; u_1 \; ... \; u_p) = \lambda x_1 \; ... \; \lambda x_n \; (f \; v_1 \; ... \; v_p)\}} \; \textit{Simplify}$$

where the simplified equation relates two rigid terms (i.e. the symbols $f$ and $g$ are either constants or among $x_1, ..., x_n$) the head symbols of these terms are different for the rule *Fail* and identical for the rule *Simplify*.

Notice that these rules derive unification problems, i.e. finite set of equations, and that we have conventionally added an “unsolvable problem” $\bot$.

4.7. Proposition. *If a problem $E'$ is derived from a problem $E$ by the rule* Fail *or the rule* Simplify*, then $E$ and $E'$ have the same solutions.*

4.8. Proposition. *The application of the rules* Fail *and* Simplify *terminates and produces a problem that does not contain rigid-rigid equations.*

### *4.2.2. Flexible-rigid Equations*

When the problem has an equation relating a flexible term and a rigid one $\lambda x_1 \ ... \ \lambda x_n \ (X \ u_1 \ ... \ u_p) = \lambda x_1 \ ... \ \lambda x_n \ (f \ v_1 \ ... \ v_q)$ we can decide to generate the substitutions to be substituted for the variable $X$. As in the generate and test algorithm, we try all the substitutions of the form $\lambda y_1 \ ... \ \lambda y_p \ (h \ (H_1 \ y_1 \ ... \ y_p) \ ... \ (H_r \ y_1 \ ... \ y_p))$ where $h$ is a constant or among the variables $y_1, ..., y_p$.

In this case, if $h$ is a constant different from the head $f$ of the rigid term, this substitution leads to an unsolvable rigid-rigid equation. Thus such an enumeration can be avoided and we can restrict the symbol $h$ to be among $y_1, ..., y_p$ (such a substitution is called a *projection*) or the symbol $f$, if this symbol is a constant (such a substitution is called an *imitation* ).

$$\frac{E}{(\lambda y_1 \ ... \ \lambda y_p \ (h \ (H_1 \ y_1 \ ... \ y_p) \ ... \ (H_r \ y_1 \ ... \ y_p))/X)E} \ \textit{Generate}$$

where $E$ contains an equation of the form

$$\lambda x_1 \ ... \ \lambda x_n \ (X \ u_1 \ ... \ u_p) = \lambda x_1 \ ... \ \lambda x_n \ (f \ v_1 \ ... \ v_q)$$

or

$$\lambda x_1 \ ... \ \lambda x_n \ (f \ v_1 \ ... \ v_q) = \lambda x_1 \ ... \ \lambda x_n \ (X \ u_1 \ ... \ u_p)$$

and $h$ is among $y_1, ..., y_p, f$ when $f$ is a constant and among $y_1, ..., y_p$ otherwise.

### *4.2.3. Flexible-flexible Equations*

Thus, while in a problem $E$ we have a rigid-rigid equation, a flexible-rigid one or a rigid-flexible one, we do not need to use the blind generation of the potential solutions, but we can restrict to the rules *Fail*, *Simplify* and *Generate*. When all the equations are flexible-flexible, it seems that we have no way to restrict the blind enumeration anymore.

However, Huet’s lemma shows that flexible-flexible equations always have solutions and thus, that if we are not interested in all the unifiers, but simply in the *existence* of such unifiers, we do not need to solve flexible-flexible equations.

4.9. Definition. (Solved problem)

If all the equations of a problem $E$ relate flexible terms, then the problem $E$ is said to be *solved*.

4.10. Proposition. *(Huet) Any solved problem has a solution.*

Proof. For each atomic $U$ type consider a constant $c_U$. Let $\theta$ the substitution that binds every variable $X$ of type $T_1 \to ... \to T_p \to U$ of $E$ to the term $\lambda y_1 \, ... \, \lambda y_p \; c_U$. The substitution $\theta$ is a solution of $E$, indeed applying $\theta$ to an equation of the form $\lambda x_1 \, ... \, \lambda x_n \; (X \; u_1 \, ... \, u_p) = \lambda x_1 \, ... \, \lambda x_n \; (Y \; v_1 \, ... \, v_q)$ yields $\lambda x_1 \, ... \, \lambda x_n \; c_U = \lambda x_1 \, ... \, \lambda x_n \; c_U$. □

In higher-order logic, testing unifiability is much simpler than enumerating unifiers. This motivates the design of proof-search methods, such as *constrained resolution* [Huet 1972, Huet 1973*a*], that require only the testing of unifiability and not the enumeration of solutions, see [Andrews 2001] (Chapter 15 of this Handbook).

### *4.2.4. Correctness*

We want to prove the soundness and completeness of the inference system *Fail*, *Simplify* and *Generate*.

4.11. Proposition. *(Soundness) If from a problem $E$ we can infer a solved problem $E'$ with the rules* Fail*,* Simplify *and* Generate*, then the problem $E$ has a solution.*

Proof. By induction on the length of the derivation.

If the derivation is empty, we conclude with the proposition 4.10.

If the first rule is *Fail* or *Simplify*, we conclude with the induction hypothesis and the proposition 4.7.

If the first rule is *Generate*, deriving the problem $E'$ from the problem $E$, then by induction hypothesis the problem $E'$ has a solution $\theta'$ and the substitution $\theta = \theta' \circ (\lambda y_1 \, ... \, \lambda y_p \; (h \; (H_1 \; y_1 \, ... \, y_p) \, ... \, (H_r \; y_1 \, ... \, y_p))/X)$ is a solution of $E$. □

4.12. Proposition. *(Completeness) If the problem $E$ has a solution $\theta$, then from a problem $E$ we can derive a solved problem $E'$ with the rules* Fail*,* Simplify *and* Generate*.*

Proof. By induction on the size of the substitution $\theta$. First, we apply the rules *Fail* and *Simplify* to the problem $E$. By proposition 4.8 this process terminates and returns a problem $E'$ that does not contain rigid-rigid equations and by proposition 4.7, the substitution $\theta$ is a solution of the problem $E'$. If the problem $E'$ is solved, we have a derivation from $E$ to a solved problem.

Otherwise, the problem $E'$ contains a flexible-rigid equation (or a rigid-flexible one) $\lambda x_1 \, ... \, \lambda x_n \; (X \; u_1 \, ... \, u_p) = \lambda x_1 \, ... \, \lambda x_n \; (f \; v_1 \, ... \, v_q)$. Let $\lambda y_1 \, ... \, \lambda y_p \; (h \; w_1 \, ... \, w_r)$ be the term $\theta X$. The symbol $h$ is among $y_1, ..., y_p, f$ if the symbol $f$ is a constant, and among $y_1, ..., y_p$ otherwise.

By the rule *Generate* we derive the problem

$$E'' = (\lambda y_1 \, ... \, \lambda y_p \, (h \, (H_1 \, x_1 \, ... \, x_p) \, ... \, (H_r \, x_1 \, ... \, x_p))/X)E'$$

A solution to this problem is the substitution

$$\theta' = \theta - \{\theta X/X\} \cup \{\lambda y_1 \, ... \, \lambda y_p w_1/H_1, ..., \lambda y_1 \, ... \, \lambda y_p w_r/H_r\}$$

This substitution is smaller than the substitution $\theta$ and thus, by induction hypothesis, we can derive a solved problem from the problem $E''$. Thus, we can derive a solved problem from the problem $E$. □

### *4.2.5. Non-determinism*

The proof of the completeness lemma gives a complete strategy for applying these rules. While a problem contains rigid-rigid equations, the rules *Fail* and *Simplify* can be applied. The choice of applying these rules and the choice of the equation is *don't care*, i.e. we never need to backtrack to try another rule or another equation.

The *Generate* rule can be applied only to simplified problems, i.e. problems containing no rigid-rigid equation. The choice of the flexible-rigid equation is *don't care*, but the choice of the head-variable is *don't know* and may lead to backtrack.

Again, this *don't know* non-determinism can be handled by building a search tree called *unification tree*. Nodes are labeled by simplified problems. Leaves are solved problems and the unsolvable problem ($\perp$). In each internal node, we chose an equation and we draw an edge corresponding to each possible head symbol.

Another solution is to consider an inference system on finite set of unification problems, deriving from $A \cup \{E\}$ the set $A \cup \{\sigma_1 E, ..., \sigma_n E\}$ where $\sigma_1, ..., \sigma_n$ are the elementary substitutions corresponding to the different possible head symbols.

In some presentations, an equation is considered as an atomic proposition in a *unification logic*. A unification problem (finite set of equations) is then the conjunction of the atomic propositions corresponding to the equations. Sets of unification problems are then considered as disjunctions.

### *4.2.6. Empty Types*

Above we have used the fact that we had a constant in each atomic type and thus that every type was inhabited and that the existence of a solution to a unification problem was equivalent to the existence of a closed solution.

If we allow empty types, finding a closed solution to a unification problem is more difficult that finding a (possibly open) solution. For instance, if we have a variable $X$ of type $T$, the empty unification problem with respect to this variable (or, if we prefer, the problem $X = X$) has a trivial open solution, but has a closed solution only if the type $T$ is inhabited.

When we have empty types, flexible-flexible equations do not always have closed solutions, for instance $X = X$ does not if the type of $X$ is empty (the existence of a such a solution is even undecidable [Miller 1992]). Thus we cannot avoid solving flexible-flexible equations.

Notice that any type inhabitation problem can be expressed as a unification problem, taking a variable of type $T$ and searching for a solution to the empty problem (or to the problem $X = X$).

### *4.2.7. Unification Modulo the Rule $\beta$ Alone*

A long normal term of type $T_1 \rightarrow ... \rightarrow T_p \rightarrow U$ ($U$ atomic) has the form $\lambda x_1 \ ... \ \lambda x_p \ (h \ u_1 \ ... \ u_r)$ where the number of abstractions is $p$, i.e. the arity of its type.

To define the long normal form, we need to have the rule $\eta$, which is a consequence of the extensionality axiom. If we drop the extensionality axiom, we have to unify terms modulo the rule $\beta$ alone.

A $\beta$-normal term of type $T_1 \rightarrow ... \rightarrow T_p \rightarrow U$ is now of the form $\lambda x_1 \ ... \ \lambda x_q \ (h \ u_1 \ ... \ u_r)$ with $q \leq p$. Thus we must consider more elementary substitutions where the number of abstractions ranges from 0 to $n$. Such an algorithm is described in [Huet 1975, Huet 1976].

For instance with the rules $\beta$ and $\eta$, the problem

$$(X \ a) = (f \ a)$$

has the two solutions $\lambda x \ (f \ a)/X$ and $\lambda x \ (f \ x)/X$. But, with the rule $\beta$ alone, it has also a third one $f/X$ which is not equivalent to $\lambda x \ (f \ x)/X$ anymore.

## *4.3. Regular Trees, Regular Solutions*

We have seen (proposition 3.4) that the solutions of the problem

$$\lambda z \ (X \ z \ (\lambda y \ y)) = \lambda z \ z$$

where $X$ is a variable of type $\iota \rightarrow (\iota \rightarrow \iota) \rightarrow \iota$, are all the substitutions of the form $t/X$ where $t$ is a Church number.

When we apply the elementary substitution $\lambda x \ \lambda f \ x/X$ to this problem and simplify it, we get the empty problem that is solved. And when we apply the elementary substitution $\lambda x \ \lambda f \ (f \ (Y \ f \ x))/X$ we get the problem

$$\lambda z \ (Y \ z \ (\lambda y \ y)) = \lambda z \ z$$

which is a renaming of the initial problem. Thus, only a finite number of problems (in this case, two) can be generated. In other words, the unification tree is regular and can be represented by a finite skeleton.

M. Zaionc [Zaionc 1987] has remarked that when the number of problems we can generate from a given problem is finite (in other words when the problem has a regular unification tree) we can compute this set of problems (or the skeleton of the unification tree). If this finite set does not contain a solved problem, then we know that the problem is unsolvable. This way he has sharpened Huet's algorithm and proposed an algorithm that reports failures more often that Huet's. For instance, for the problem

$$(X\ a) = (f\ (X\ a))$$

Huet's algorithm constructs an infinite tree with no solved problem, while after an imitation step $\lambda x\ (f\ (H\ x))/X$ yielding after simplification to the problem

$$(H\ a) = (f\ (H\ a))$$

Zaionc's algorithm reports a failure.

Moreover, when the number of problems generated by a given problem is finite and the problem has solutions, the set of minimal unifiers may be infinite, but it can be described by a grammar. For instance, for the problem

$$\lambda z\ (X\ z\ (\lambda y\ y)) = \lambda z\ z$$

calling $\sigma$ the elementary substitution $\lambda x\ \lambda f\ x/X$ and $\tau$ the elementary substitution $\lambda x\ \lambda f\ (f\ (X\ f\ x))/X$, all the solutions have the form $\tau \circ \tau \circ ... \circ \tau \circ \sigma$. Such a substitution can be represented by the word $\tau\tau ... \tau\sigma$ and the set of words corresponding to the minimal unifiers is produced by the grammar

$$s \rightarrow \sigma$$

$$s \rightarrow \tau s$$

### *4.4. Equational Higher-order Unification*

Several extensions of Huet's algorithm have been proposed to higher-order equational unification (see section 1.5). Some aim at giving a general algorithm for an arbitrary higher-order equational theory (see, for instance, [Avenhaus and Loría-Sáenz 1994, Müller and Weber 1994, Prehofer 1994*b*, Prehofer 1995, Qian 1994, Qian and Wang 1992, Snyder 1990]). Others consider special theories (see, for instance, [Curien 1995, Qian and Wang 1994, Saïdi 1994, Boudet and Contejean 1997]).

## 5. Scopes Management

The idea, underlying Huet's algorithm is to build the terms substituted for the variables step by step and to transform the equations at each step substituting the part of the solution constructed so far. This is a rather general approach in equational unification. As compared to first-order equational unification methods, higher-order unification presents the particularity of a rather subtle management of scopes. For instance, as already mentioned, we cannot take the elementary substitution $\lambda y_1\ ...\ \lambda y_p\ (f\ H_1\ ...\ H_r)/X$ but we must express the dependence of the arguments of $f$ with respect to the variables $y_1, ..., y_p$ in a functional way, taking the substitution $\lambda y_1\ ...\ \lambda y_p\ (f\ (H_1\ y_1\ ...\ y_p)\ ...\ (H_r\ y_1\ ...\ y_p))/X$. This is due to the fact that substitution in $\lambda$-calculus renames bound variables to avoid captures.

We might want to build the solutions with smaller steps, for instance substituting a variable of a functional type $T \to U$ by a term of the form $\lambda y\ H/X$ and then substitute $H$. But this is not possible as expressing functionally the dependence in such a substitution would yield $\lambda y\ (H\ y)/X$. i.e. $H/X$ and thus the inductive argument in the completeness proof would not go through. (Because of the functional encoding of scopes we have to take $|\lambda x\ t| = |t|$ and not $|\lambda x\ t| = |t| + 1$, thus instantiating a variable by an abstraction does not let the problem progress).

In the same way, we might want to simplify an equation of the form $\lambda x\ u = \lambda x\ v$ into $u = v$, but such a simplification rule is unsound. For instance the equation $\lambda x\ Y = \lambda x\ (f\ x\ x)$ has no solution (as the substitution $(f\ x\ x)/Y$ would rename the bound variable to avoid the capture), while the equation $Y = (f\ x\ x)$ has the solution $(f\ x\ x)/Y$.

All these particularities of higher-order unification come from the particularities of the substitution in $\lambda$-calculus, and this particularities come from the fact that $\lambda$-calculus contains a binding operator $\lambda$.

### *5.1. Mixed Prefixes*

To have the simplification rule

$$\frac{E \cup \{\lambda x\ t = \lambda x\ u\}}{E \cup \{t = u\}}$$

we must add to the simplified problem an *occurrence constraint* forbidding the variable $x$ to appear in the term substituted for the variables free in $t$ and $u$. This way both problems

$$\lambda x\ Y = \lambda x\ (f\ x\ x)$$

and

$$Y = (f\ x\ x),\ x \text{ not available to } Y$$

have no solution, and more generally the simplification of abstractions is sound. Such occurrence constraints can be elegantly expressed in a unification logic.

In a unification logic, unification problems are expressed as propositions and unification rules as deduction rules in such a way that a proposition $P$ is provable if and only if it expresses a unifiable problem. An equation $t, u$ is expressed as an atomic proposition $t = u$ introducing a predicate symbol $=$. A unification problem (i.e. a finite set of equations) is represented as the conjunction of the atomic propositions corresponding to equations. The variables occurring in the problem are then existentially quantified at the head of the problem and the constants can be considered as universally quantified variables.

For instance the problem

$$\lambda x\ Y = \lambda x\ (f\ x\ x)$$

is expressed as the proposition

$$\forall f\ \exists Y\ (\lambda x\ Y = \lambda x\ (f\ x\ x))$$

Unification problems are thus usually expressed as propositions of the form $\forall\exists$. D. Miller [Miller 1992] has proposed to consider propositions with a more complex alternation of quantifiers (*mixed prefixes*), in particular propositions of the form $\forall\exists\forall$. Then the problem

$$\forall f\ \exists Y\ (\lambda x\ Y = \lambda x\ (f\ x\ x))$$

can be simplified into

$$\forall f\ \exists Y\ \forall x\ (Y = (f\ x\ x))$$

in which the usual scoping rules for quantifiers manipulation express that the variable $x$ is not available to $Y$ and forbid the substitution $(f\ x\ x)/Y$. This way, he has been able to give more natural simplification rules. The study of quantifier permutation in such problems has also permitted to identify a decidable subcase of higher-order unification (see section 6.2).

Thus, mixed prefixes permit to give more natural simplification rules, but not to give more natural generation rules.

### *5.2. Combinators*

Recall that higher-order logic is just one among several variants of set theory [Davis 1969]. Like other variants of set theory, it can be expressed in first-order logic.

When we express higher-order logic as a first-order theory, the term language is a first-order term language and thus, as opposed to $\lambda$-calculus it contains no binding operator. Thus, the substitution does not need to avoid captures and scope management is simpler. Expressing this way higher-order logic as a first-order theory permits also to use standard technique for proof search and in particular standard first-order equational unification algorithms for higher-order unification.

When we express higher-order logic as a (many-sorted) first-order theory, we need to distinguish zero-ary relations that are expressed by terms of sort $o$ and propositions. We introduce a unary predicate symbol $\varepsilon$ of rank $(o)$ and if $t$ is a term of type $o$, the corresponding proposition is written $\varepsilon(t)$. For each pair of type, we introduce also a function symbol $\alpha_{T,U}$ of rank $(T \to U, T, U)$ and the term $(t\ u)$ is a notation for $\alpha_{T,U}(t, u)$. We may also introduce symbols $=_T$ of type $T \to T \to o$ for equality.

As seen in section 1 we state the comprehension schemes

$$\forall x_1\ ...\ \forall x_n\ \exists f\ \forall y_1\ ...\ \forall y_p\ (\varepsilon(f\ y_1\ ...\ y_p) \Leftrightarrow P))$$

$$\forall x_1\ ...\ \forall x_n\ \exists f\ \forall y_1\ ...\ \forall y_p\ \varepsilon((f\ y_1\ ...\ y_p) = t)$$

These schemes are equivalent to the closed schemes

$$\exists f\ \forall y_1\ ...\ \forall y_p\ (\varepsilon(f\ y_1\ ...\ y_p) \Leftrightarrow P))$$

$$\exists f\ \forall y_1\ ...\ \forall y_p\ \varepsilon((f\ y_1\ ...\ y_p) = t)$$

where all the free variables of $t$ and $P$ are required to be among $y_1, ..., y_p$.

As seen in section 1, to have a language for the relations and functions we skolemize these axioms and introduce symbols that we may write $\lambda y_1\ ...\ \lambda y_p\ t$ and $\lambda y_1\ ...\ \lambda y_p\ P$ if we want, but we must recall that (1) in such expressions, the free variables of $P$ and $t$ must be among $y_1, ..., y_p$, (2) the proposition $P$ and the term $t$ do not contain further abstractions and (3) such terms are individual symbols. These symbols are called *combinators* [Curry 1942, Curry and Feys 1958, Hindley and Seldin 1986].

We can also chose to restrict the comprehension schemes to a finite number of instances that are equivalent to the full scheme. This way we have a finite number of combinators (e.g. $S, K, \Rightarrow, \wedge, \vee, \neg, \forall, \exists$).

We have said in section 1 that this language was equivalent to $\lambda$-calculus, but moving from this language to the more convenient notation of $\lambda$-calculus introduces the binding operator $\lambda$ and thus the notion of substitution with renaming. An alternative is to keep this language and to perform unification modulo the combinators conversion axioms.

The use of combinators, instead of $\lambda$-calculus, was already investigated by J.A. Robinson in 1970 [Robinson 1970] (but, apparently, without stressing the relation to Davis' remark that higher-order logic could be expressed as a first-order theory). This approach has been pursued in [Dougherty 1993]. Using combinators instead of $\lambda$-calculus permits to use standard first-order equational unification algorithms to perform unification modulo the conversion axioms.

The translations from $\lambda$-calculus to combinators [Curry 1942, Curry and Feys 1958, Hindley and Seldin 1986, Hughes 1982, Johnsson 1985, Dowek 1995], such as $\lambda$-lifting, are correct if the extensionality axiom is taken, but not when this axiom is dropped: the theory of the conversion axiom alone are not equivalent in $\lambda$-calculus and in the theory of combinators. In other words, some proofs, for instance that of the proposition

$$((\lambda x\ \lambda y\ \lambda z\ x)\ w\ w) = ((\lambda x\ \lambda y\ \lambda z\ y)\ w\ w)$$

do not require the use of the extensionality axiom in $\lambda$-calculus and requires it with combinators.

If unification is seen as a part of resolution, then resolution in the $\lambda$-calculus presentation of higher-order logic is equivalent to resolution in the combinators presentation of higher-order logic, i.e. a proposition is provable in one system if and only if its translation is provable in the other (although the proofs may be different in the two systems, in particular one may need to use the extensionality axiom, while the other does not).

If unification is seen as an independent problem then combinator unification is weaker than higher-order unification, i.e. it is not the case that a problem has a solution in $\lambda$-calculus if and only if its translation has one in combinators, but combinator unification may be adapted to get the same power as higher-order unification, using a glimpse of extensionality [Dougherty 1993]. This algorithm is, however, more redundant than Huet's.

Notice, at last, that the higher-order unification algorithm itself uses part of the translation of $\lambda$-calculus to combinators. In particular the functional encoding of scopes is reminiscent of $\lambda$-lifting.

### *5.3. Explicit Substitutions*

To avoid the problems with extensionality introduced by the use of combinators, another solution is to keep $\lambda$-calculus, but to avoid the difficulties of scopes management with the use the *replacement*, allowing capture (see definition 2.11) instead of substitution. In other words, when we have an equation $a = b$ we do not look for a substitution $\theta$ such that $\theta a = \theta b$ but for a substitution $\theta$ such that $\langle\theta\rangle a = \langle\theta\rangle b$.

Using such a notion of replacement permits to decompose the simplification rules into a rule simplifying equations of the form $\lambda x\ u = \lambda x\ v$ into $u = v$ an other one simplifying equations of the form $(f\ u_1\ ...\ u_p) = (f\ v_1\ ...\ v_p)$ into $u_1 = v_1, ..., u_p = v_p$ and $(f\ u_1\ ...\ u_p) = (g\ v_1\ ...\ v_q)$ into $\perp$ when $f$ and $g$ are different. The generation rule can also be simplified: if $X$ is a variable of type $T \to U$ we can replace it by a term $\lambda x\ Y$ where $Y$ is a variable of type $U$, and when $X$ has an atomic type, we replace it by the term $(h\ H_1\ ...\ H_r)$.

But, this notion of replacement raises two new difficulties. First replacement does not commute with reduction and thus it cannot be defined on the quotient of terms modulo reduction. For instance, the term $((\lambda x\ Y)\ a)$ reduces to $Y$, but replacing $x$ for $Y$ yields $((\lambda x\ x)\ a)$ that reduces to $a$ and not to $x$. To avoid this difficulty, a solution is to delay the substitution of $a$ for $x$ in $Y$ until $Y$ is replaced and we know whether it contains an occurrence of $x$ or not (when using substitution with renaming such a delay is not needed (proposition 2.22) because a term containing the variable $x$ cannot be substituted for the variable $Y$).

Delaying this way the substitutions initiated by $\beta$-reduction requires an extension of $\lambda$-calculus with *explicit substitutions* [Abadi, Cardelli, Curien and Lévy 1991, Curien, Hardin and Lévy 1996, Nadathur and Wilson 1998]. Besides constants, variables, applications and abstractions, the calculus of explicit substitutions introduces another construction the *closure* $[\sigma]t$ where $\sigma$ is a substitution and $t$ a term. The $\beta$-reduction rule is replaced by the rule

$$((\lambda x\ t)\ u) \rhd [u/x]t$$

and more reduction rules permit to propagate the explicit substitution $u/x$ in the term $t$. The simplest rules permits to distribute a substitution on an application

$$[\sigma](t\ u) \rhd ([\sigma]t\ [\sigma]u)$$

When such a substitution $[u/x]$ reaches the variable $x$ the term $[u/x]x$ reduces to the term $u$, when it reaches another variable $y$ the term $[u/x]y$ reduces to $y$, but when it reaches a *metavariable* $Y$ the term $[u/x]Y$ cannot be reduced and thus the substitution is delayed until the metavariable $Y$ is replaced. Thus, when we express a higher-order unification problem in the calculus of explicit substitutions free variables are expressed as metavariables and bound variables as ordinary variables.

The second difficulty is that some problems have solutions for replacement while they have none for substitution. This is the case for instance for the problem $\lambda x\, Y = \lambda x\ (f\ x\ x)$. If we use replacement, $(f\ x\ x)/Y$ is a solution, while the problem has no solution if we use substitution. This problem is solved again by the use of explicit substitutions. In this system, there are explicit renaming operators and thus we can use such an operator to protect the metavariable $Y$ from being replaced by a term containing the variable $x$.

Thus we can define a translation from $\lambda$-calculus to $\lambda$-calculus with explicit substitutions such that a unification problem has a solution for substitution if and only if its translation has one for replacement. In other words, the substitution of $\lambda$-calculus is decomposed into an (explicit) renaming and a replacement.

This approach has been investigated in [Dowek, Hardin and Kirchner 1995, Borovanský 1995, Nadathur 1998, Nadathur and Mitchell 1999].

### *5.4. De Bruijn Indices*

Like combinators, the calculus of explicit substitutions permits to avoid the subtle scope management of higher-order unification and it avoids also the use of extensionality. But, so far, it does not permit to use the standard first-order equational unification techniques because $\lambda$-calculus (with explicit substitutions or not) is still not a first-order language.

In fact, independently of combinators, N.G. de Bruijn [de Bruijn 1972] has proposed another notation for $\lambda$-calculus that happened to be also a first-order language.

The idea of de Bruijn notation, is that the name of bound variables is only used to indicate the binder they depend on. This dependency may also be indicated by the height of this binder above the variable. For instance, the term $\lambda x\ \lambda y\ (x\ \lambda z\ (x\ z))$ may be written $\lambda\lambda(2\ \lambda(3\ 1))$ because the first occurrence of the variable $x$ refers to the second $\lambda$ above it, the second occurrence of the variable $x$ refers to the third $\lambda$ above it and the occurrence of the variable $z$ refers to the first $\lambda$ above it.

In de Bruijn notation, the operator $\lambda$ is not a binding operator anymore and thus $\lambda$-calculus can be represented as a first-order term language with a unary function symbol $\lambda$, a binary function symbol $\alpha$ and an infinite number of constant symbol 1, 2, etc.

Because of the presence of the substitution in the $\beta$-reduction rule, the reduction system in this language is not a first-order rewrite system, but the reduction system in $\lambda$-calculus with de Bruijn indices and explicit substitutions is first-order. In fact, the standard presentation of the calculus of explicit substitutions uses de Bruijn indices and not named variables. The metavariables of the calculus of explicit substitutions are the variables of the free algebra built on this language.

With de Bruijn indices and explicit substitutions, we can use first-order techniques to perform unification, we do not have scopes management problems nor those created by the use extensionality in translating $\lambda$-calculus to combinators [Dowek et al. 1995].

These investigations have also lead to another first-order presentation of higher-order logic based on de Bruijn indices and explicit substitutions that is extensionally equivalent to the presentation using $\lambda$-calculus [Dowek, Hardin and Kirchner 2001].

Presenting this way $\lambda$-calculus as a first-order language and higher-order unification as first-order equational unification modulo an equational theory $T_1$ permits to consider also equational higher-order unification modulo an equational theory $T_2$ as equational first-order unification modulo $T_1 \cup T_2$ [Dougherty and Johann 1992, Goubault 1994, Kirchner and Ringeissen 1997].

## 6. Decidable Subcases

As usual when a problem is undecidable, besides building a semi-decision algorithm, we are also interested in identifying decidable subcases. In this section, we present a few decidable subcases of higher-order unification. These subcases are obtained by restricting the order, the arity or the number of occurrences of variables, or by taking terms of a special form. For some subcases, Huet's algorithm terminates, for others it does not and we must design another algorithm to prove decidability.

The main conjectures in this area are the decidability of pattern-matching, i.e. the subcase of unification where variables occur only in a single side of equations and the decidability of context unification.

### *6.1. First-order Unification*

The first decidable subcase of higher-order unification is obviously first-order unification. When all the variables of a problem have first-order, i.e. atomic, types (see definition 2.3), all the constants have at most second-order types and the terms in the equations have first-order types, then the problem is just a rephrasing of a first-order unification problem. Notice however that Huet's algorithm does not always terminate on such problems. For instance the problem

$$X = f(X)$$

leads to an infinite search. In other words, Huet's algorithm does not detect failure by occur-check. However, it can be sharpened, adding a rule called *rigid paths occur-check* [Huet 1975, Huet 1976] that forces failure in more cases and in particular for all the first-order unsolvable unification problems.

### *6.2. Patterns*

When we define a function by an equation, for instance,

$$\forall x\ \forall y\ ((F\ x\ y) = x + y + x \times y)$$

we actually mean

$$F = \lambda x\ \lambda y\ (x + y + x \times y)$$

But the first definition can also be used because the equation has a single solution $\lambda x\ \lambda y\ (x + y + x \times y)$. In contrast, the definitions

$$\forall x\ ((F\ x\ x) = x + x + x \times x)$$

or

$$\forall x\ ((F\ 0\ x) = x)$$

are incorrect because the equations have more than one solution.

This remark motivates the study of unification problems where the higher-order free variables can only be applied to distinct bound variables.

A *pattern* [Miller 1991] is a term $t$ such that for every subterm of the form $(F\ u_1\ ...\ u_n)$ where $F$ is a free variable, the terms $u_1, ..., u_n$ are distinct variables bound in $t$. Unification of patterns is decidable and when a unification problem has a unifier, it has a most general unifier [Miller 1991].

For instance the problem

$$\lambda x\ \lambda y\ \lambda z\ (F\ x\ z) = (f\ (\lambda x\ \lambda y\ (G\ y\ x))\ (\lambda x\ \lambda y\ (F\ x\ y)))$$

is a patterns unification problem.

Patterns unification extends first-order unification. It has the same properties (polynomial time decidability and most general unifier) and the algorithms have some similarities (in particular, the occur-check plays an essential role in both cases). The correspondence between first-order unification and patterns unification is better understood when we study quantifier permutation in mixed prefixes (see section 5.1) as patterns unification problems can be obtained by permuting quantifiers in first-order problems [Miller 1992]. This is also the way patterns unification was discovered.

Patterns unification is used in higher-order logic programming [Nadathur and Miller 1998, Pfenning 1991*a*].

Patterns unification with explicit substitutions is studied in [Dowek, Hardin, Kirchner and Pfenning 1996], the decidability and unicity of solution rely there on invertibility properties of explicit substitutions.

This subcase of unification called *patterns unification* must not be confused with *pattern matching* discussed in section 6.6.

### *6.3. Monadic Second-order Unification*

Goldfarb's undecidability proof requires a language with a binary constant $g$. Thus, a natural problem to investigate is unification in second-order languages containing only unary constants, i.e. constants with a single argument. This problem, called *unary or monadic second-order unification* has been proved decidable by Farmer [Farmer 1988].

Farmer's proofs relies on the fact that a closed term of an atomic type in such a language has the form $(f_1\ (f_2\ ...\ (f_n\ c)...))$ and thus can be represented by the word $f_1 f_2 ... f_n$. Thus, a unification problem in such a language can be reduced to word unification problem, and such problems are known to be decidable.

In this case, the set of minimal unifiers may be infinite. For instance the unification problem

$$\lambda z\ (f\ (X\ z)) = \lambda z\ (X\ (f\ z))$$

which is equivalent to the word problem $fX = Xf$ has an infinite number of minimal solutions where the terms $\lambda x\ x$, $\lambda x\ (f\ x)$, $\lambda x\ (f\ (f\ x))$, $\lambda x\ (f\ (f\ (f\ x)))$, $\lambda x\ (f\ (f\ (f\ (f\ x))))$, etc. are substituted for the variable $X$, corresponding to the solutions of the word problem $\varepsilon$, $f$, $ff$, $fff$, etc. Farmer proposes to describe minimal unifiers using so called *parametric terms*, reminding of Zaionc's description by a grammar. For instance the parametric term $\lambda x\ (f^n\ x)$ (corresponding to the parametric word $f^n$) is the most general unifier of the problem above.

### *6.4. Context Unification*

*Context unification* is a variant of second order unification with the extra condition that terms substituted to second order variables have to be *contexts*, i.e. normal terms of the form $\lambda x_1\ ...\ \lambda x_n\ t$ where the variables $x_1, ..., x_n$ occur once in $t$. Such terms can be seen as first-order terms with holes. This problem is related to unification in linear lambda-calculus [Pfenning and Cervesato 1997].

The decidability of this problem is open, [Comon 1998, Schmidt-Schauß 1994, Levy 1996, Niehren, Pinkal and Ruhrberg 1997, Schmidt-Schauß and Schulz 1999, Schmidt-Schauß 1999, Levy and Villaret 2000, Niehren, Tison and Treinen 2000] give partial results.

### *6.5. Second-order Unification with Linear Occurrences of Second-order Variables*

In second-order unification, when we have an equation

$$\lambda x_1\ ...\ \lambda x_n\ (X\ a_1\ ...\ a_p) = \lambda x_1\ ...\ \lambda x_n\ (f\ b_1\ ...\ b_q)$$

and we perform a projection, we replace a variable $X$ by a closed term $\lambda x_1 ... \lambda x_n\ x_i$, thus the number of variables in the problem decreases. When we perform an imitation and simplify the problem, we get the equations

$$\lambda x_1\ ...\ \lambda x_n\ (H_1\ a_1\ ...\ a_p) = \lambda x_1\ ...\ \lambda x_n\ b_1$$

$$...$$

$$\lambda x_1\ ...\ \lambda x_n\ (H_q\ a_1\ ...\ a_p) = \lambda x_1\ ...\ \lambda x_n\ b_q$$

which seem to be smaller than the equation we started with. Hence, it seems that Huet's algorithm should terminate, in contradiction with Goldfarb's undecidability result.

Actually, the variable $X$ may have occurrences in the terms $a_1, ..., a_p, b_1, ..., b_q$ and in fact we get the equations

$$\lambda x_1 \; ... \; \lambda x_n \; (H_1 \; a'_1 \; ... \; a'_p) = \lambda x_1 \; ... \; \lambda x_n \; b'_1$$

$$...$$

$$\lambda x_1 \; ... \; \lambda x_n \; (H_q \; a'_1 \; ... \; a'_p) = \lambda x_1 \; ... \; \lambda x_n \; b'_q$$

where the variable $X$ has been substituted everywhere. These equation need not be smaller than the equation we started with and thus the algorithm does not always terminate.

However this argument can be used to prove that second-order unification with linear occurrences of second-order variables is decidable, i.e. that there is an algorithm that decides unifiability of second-order problems where each second-order variable has a single occurrence (see, for instance, [Dowek 1993*d*]). In fact, to ensure that linearity is preserved by imitation we must first transform equations into *superficial* equations, i.e. equations where the second-order variables can occur only at the head of the members of the equations.

This algorithm has been extended by Ch. Prehofer [Prehofer 1994*a*, Prehofer 1995] mixing linearity conditions and patterns conditions.

G. Amiot [Amiot 1990] had used a similar transformation to prove that superficial second-order unification is undecidable.

Besides linear unification, a similar argument using the number of variables and the size of equations will be used in section 6.6.1 to prove the decidability of second-order matching.

### *6.6. Pattern Matching*

A higher-order matching equation is an equation whose right hand side does not contain free variables. A higher-order matching problem is a finite set of matching equations. The decidability of higher-order matching, Huet's conjecture [Huet 1976], has been an open problem for more than twenty years.

#### *6.6.1. Second-order Matching*

The first positive result is the decidability of second-order matching.

6.1. Proposition. *(Huet [Huet 1976, Huet and Lang 1978]) Second-order matching is decidable, i.e. there is an algorithm that takes in argument a matching problem whose free variables are at most second-order (in the sense of definition 2.3) and whose bound variables and constants are at most third-order and answers if it has a solution or not.*

Proof. For second-order matching problems, Huet's algorithm terminates. Indeed, the pair $(n, p)$ where $n$ is the sum of the sizes of the right hand sides of equations and $p$ the number of variables in the problem decreases at each step (i.e. each application of the *Generate* rules followed by a simplification) for the lexicographic order.

Imitations are always followed by a simplification, and thus the sum of the sizes of the right hand sides of equations decreases in such a step. Projections have the form $\lambda x_1 \ ... \ \lambda x_n \ x_i$ thus they do not introduce new variables $H_1, ..., H_r$ and the number of variables in the problem decreases in such a step and the closed right hand sides are never substituted, thus the sum of their sizes never increases. □

6.2. Remark. In a matching problem there is no flexible-flexible equations. Thus the only solved problem is the empty problem and a second-order matching problem has a finite set of minimal solutions.

6.3. Remark. L.D. Baxter has proved that the second order matching problem is NP-complete [Baxter 1977].

6.4. Remark. The condition that bound variables and constants are at most third-order can be weakened (see, for instance, [Dowek 1991*c*]), but patterns-like terms need to be used in the algorithm.

### *6.6.2. Infinite Set of Solutions and Pumping*

As soon as we have a third-order variable, Huet's algorithm may fail to terminate and may produce an infinite number of minimal solutions. For instance, as seen above (proposition 3.4) the problem

$$\lambda z \ (X \ z \ (\lambda y \ y)) = \lambda z \ z$$

has an infinite number of solutions of the form $t/X$ where $t$ is any Church number $\lambda x \ \lambda f \ (f \ ... \ (f \ x)...)$.

Thus if we look for a terminating algorithm, we cannot use Huet's algorithm, and we cannot use any other algorithm enumerating all the minimal solutions. Thus, all the algorithms proposed so far (for restricted cases) all reduce the search space, dropping some solutions, but hopefully keeping at least one if the problem has solutions.

As an illustration we can use such a method to prove the decidability, in the domain of natural numbers, of polynomial equations with a constant right hand side (whereas Matiyacevich-Robinson-Davis [Matiyacevich 1970, Davis 1973] theorem proves the undecidability of polynomial equations in general). Notice that in this case also, a problem may have an infinite number of solutions (consider for instance the equation $XY + 4 = 4$).

6.5. Proposition. *There is an algorithm that takes as arguments a polynomial $P$ whose coefficients are natural numbers and a natural number $b$ and answers if the equation $P(X_1, ..., X_n) = b$ has a solution or not in the domain of natural numbers.*

Proof. If this equation has a solution $a_1, ..., a_n$ then it has a solution $a'_1, ..., a'_n$ such that $a'_1 \leq b$. Indeed either $Q(X) = P(X, a_2, ..., a_n)$ is not a constant polynomial and for all $n$, $Q(n) \geq n$, so $a_1 \leq b$, or the polynomial $Q$ is identically equal to $b$ and $\langle 0, a_2, ..., a_n \rangle$ is also a solution. A simple induction on $n$ proves that if the equation has a solution then it also has a solution in $\{0, ..., b\}^n$ and an enumeration of this set gives a decision algorithm. □

For instance, for the equation $XY + 4 = 4$, starting with the solution $\langle 1000, 0 \rangle$ we get the solution $\langle 0, 0 \rangle$. The method that transforms the solution $\langle 1000, 0 \rangle$ into $\langle 0, 0 \rangle$ is called *pumping*. It permits to know whether a solution exists in an infinite domain just by looking into a finite part of the domain, because this finite part mirrors all the domain.

### *6.6.3. Finite Models*

Such an idea has been investigated by R. Statman using model theoretic techniques. H. Friedman's completeness theorem [Friedman 1975] is that if we interpret the atomic types by infinite sets and types of the form $A \to B$ by the set of all functions from the interpretation of $A$ to the interpretation of $B$, then two terms have the same denotation if and only if they are $\beta\eta$-convertible.

Obviously, this theorem cannot be generalized to the case where the interpretation of atomic types are finite. Indeed, if the interpretation of the type $\iota$ is finite, that of the type $\iota \to (\iota \to \iota) \to \iota$ also and thus at least two different Church numbers have the same denotation, while they are not convertible.

However Statman's finite completeness theorem [Statman 1979, Statman and Dowek 1992] shows that for each $\lambda$-term $b$, there is a natural number $n$ such that, in the finite model built from a base sets of cardinal $n$, the terms that have the same denotation as $b$ are those convertible to $b$.

Thus, if a matching problem $(a\ X_1\ ...\ X_n) = b$ ($b$ closed) has a solution, the corresponding equation in the model has a solution too, and as the denotation of each type in the model is finite, we can enumerate all the potential solutions and test one after another. Unfortunately, when we find a solution in the model this solution corresponds to a solution in $\lambda$-calculus only if the element of the model is the denotation of some $\lambda$-term. Thus the higher-order matching conjecture was reduced this way to the $\lambda$-definability decidability conjecture (Plotkin-Statman conjecture) [Statman 1979, Statman and Dowek 1992].

Another formulation, that strengthen the link to the pumping method is that assuming that we can decide whether an element is $\lambda$-definable or not we can compute a number $n$ such that all the definable elements of the model of a given type are defined by a term of size lower than $n$. Thus, if the problem has a solution, then it has also a solution of size lower than $n$ and to decide whether a problem has a solution, we only need to enumerate the terms of size lower to that bound. After this bound, the terms are redundant, i.e. their denotation is also a denotation of smaller terms and if they are solutions to the matching problem smaller terms also.

Unfortunately the $\lambda$-definability decidability conjecture has been refuted by R. Loader [Loader 1994].

However, V. Padovani has shown that $\lambda$-definability was decidable in other models: the *minimal models* where the interpretation of the type $A \to B$ contains only the $\lambda$-definable functions and from this result, he has deduced the decidability of the atomic higher-order matching problem (i.e. the higher-order matching problem where the right hand side is a constant) [Padovani 1996*a*, Padovani 1996*b*].

### 6.6.4. *Third and Fourth-order Matching*

A similar approach has permitted to prove the decidability of third-order and fourth-order matching problems i.e. matching problems whose free variables are at most third or fourth order (in the sense of definition 2.3).

Consider a variable $X$ of type $\iota \to (\iota \to \iota) \to \iota$, the equation

$$(X\ c\ (\lambda y\ (g\ (h\ y)))) = c$$

and the potential normal solution $t = \lambda x\ \lambda f\ u$ for $X$. The term $(t\ c\ \lambda y\ (g\ (h\ y)))$ reduces to the normal form of $(c/x, \lambda y\ (g\ (h\ y))/f)u$ and, a simple induction on the depth of the structure of $u$ shows that this term has a depth greater than or equal to that of $u$. For instance, taking $t = \lambda x\ \lambda f\ (f\ (f\ (f\ x)))$ and applying it to $c$ and $\lambda y\ (g\ (h\ y))$ yields $(g\ (h\ (g\ (h\ (g\ (h\ c))))))$ where each $f$ has been replaced by a $g$ and a $h$. Thus, if such a term is to be a solution of the above problem $u$ must be smaller than $c$. Thus, enumerating the terms smaller than $c$ gives an algorithm to find all the solutions of this problem. In fact, the only solutions are $\lambda x\ \lambda f\ c$ and $\lambda x\ \lambda f\ x$.

But such a reasoning does not work for all the problems, for instance

$$(X\ c\ (\lambda y\ y)) = c$$

$$(X\ d\ (\lambda y\ e)) = e$$

has solutions of an arbitrary depth: all nonzero Church numbers

$$\lambda x\ \lambda f\ (f\ (...(f\ x)...)).$$

This can only happen when all the second arguments of $X$ are either of the form $\lambda x_1\ ...\ \lambda x_n\ x_i$ (e.g. $\lambda y\ y$) or an irrelevant term i.e. a term where a bound variable does not occur in the body (e.g. $\lambda y\ e$). In this case any sequence of $f$ has the same effect as a single $f$ thus, any solution of the form $\lambda x\ \lambda f\ (f\ (...(f\ x)...))$ is redundant with the smaller solution $\lambda x\ \lambda f\ (f\ x)$.

Erasing, this way, all the useless occurrences of variables permits to get smaller solutions whose depth can be bounded by a function in the depth of $b$. Thus, we can compute a bound such that if the problem has a solution, then it has also a solution whose depth is lower than that bound and hence achieve decidability.

The simpler case for which such a method works is the third-order interpolation problems.

6.6. Definition. (Interpolation problem)

An *interpolation problem* is a finite set of equations of the form $(X\ a_1\ ...\ a_n) = b$ where the terms $a_1, ..., a_n, b$ are closed.

Using the pumping method described above, we can prove the decidability of third-order interpolation problems. Then, the bound on the depth of solutions can be lifted to arbitrary third-order matching problems and this proves the decidability of third-order matching problems [Dowek 1994].

A. Schubert [Schubert 1997] has proved that the decidability of higher-order interpolation problems implies that of higher-order matching problems, unfortunately his transformation does not preserve the order of the variables.

V. Padovani [Padovani 1995] has proved that the decidability of the dual interpolation problem implies that of higher-order matching and his transformation preserves the order of the variables thus the decidability of the dual interpolation problem of order $n$ implies that of the matching of order $n$ (a dual interpolation problem is a pair $(E, F)$ of interpolation problems and a solution to such a problem is a substitution that is solution to the equation of $E$ but not to that of $F$).

Using this result, Padovani has proved the decidability of the fourth-order matching problem [Padovani 1994, Padovani 1996*b*].

### *6.6.5. Automata*

All these proofs are rather technical (in particular the decidability of fourth-order matching is a real technical *tour de force*) because they all proceed by transforming potential solutions into smaller ones cutting and pasting term pieces. H. Comon and Y. Jurski [Comon and Jurski 1997] have proposed to reformulate these ideas in a much simpler way.

Instead of transforming a potential solution into a smaller one. Comon and Jurski propose, in a similar way as Zaionc (see section 4.3) and Farmer (see section 6.3) to build an automaton that recognizes the solutions of a given problem.

For instance, in the problem

$$(X\ c\ (\lambda y\ y)) = c$$

$$(X\ d\ (\lambda y\ e)) = e$$

the fact that any sequence of $f$ has the same effect as a single $f$ and thus that any solution of the form $\lambda x\ \lambda f\ (f\ (...(f\ x)...))$ is redundant with the smaller solution $\lambda x\ \lambda f\ (f\ x)$ is expressed as the fact that the automaton stays in the same state recognizing the sequence of $f$ in the solution $\lambda x\ \lambda f\ (f\ (...(f\ x)...))$. This way a finite state automaton can recognize the infinite set of solutions and decidability is a consequence of the decidability of the nonemptiness of a set of terms recognized by an automaton.

This way they have given simpler decidability proofs for third-order and fourth-order matching. They have also proved that third order matching was NP-complete, hence that is not more complex than second-order matching.

### *6.6.6. Wolfram's Algorithm*

A last approach has been investigated by D. Wolfram [Wolfram 1989]. Wolfram has proposed a pruning of the search tree for the full higher-order matching that produces a finite search tree. Thus, Wolfram's algorithm always terminates, but its completeness is still a conjecture.

## 7. Unification in $\lambda$-calculus with Dependent Types

To conclude this chapter we shall review unification algorithms in extensions of simply typed $\lambda$-calculus. We have already seen in section 1.5 and 4.4 that more reduction rules could be added, we can also consider richer type structure such as dependent types and polymorphism.

### *7.1. $\lambda$-calculus with Dependent Types*

#### *7.1.1. Types Parametrized by Terms*

The $\lambda$-calculus with dependent types is an extension of simply typed $\lambda$-calculus where types contain more information on terms than their functional degree. For instance in simply typed $\lambda$-calculus, we may consider lists (i.e. finite sequences) of natural numbers as atoms and thus have an atomic type *list* and two symbols $\varepsilon$ of type *list* for the empty list and . of type $list \to nat \to list$) to add an element at the end of a list. For instance the list $1, 1, 2, 3, 5$ is expressed by the term $(.\ (.\ (.\ (.\ (.\ \varepsilon\ 1)\ 1)\ 2)\ 3)\ 5)$.

But we may want to enrich the type system in such a way that the length of the list is a part of its type, i.e. we want to have a family of types $(list\ 0)$, $(list\ 1)$, $(list\ 2)$, etc. parametrized by a term of type $nat$.

The type of a function taking as argument a natural number $n$ and returning a list of length $n$, cannot be written $nat \to (list\ n)$ but we must express the information that the variable $n$ refers to the argument of the function, thus we write such a type $\Pi n_{nat}\ (list\ n)$. When we apply such a function to, for instance, the term 4 we get a term of type $(list\ 4)$, i.e. a list of four elements. From now on, the notation $A \to B$ is just an abbreviation for $\Pi x_A\ B$ where $x$ does not occur in $B$. The symbol *list* is not a type but it has type $nat \to Type$ where $Type$ is a new base type.

As types contain terms, the type of a variable may be changed by a substitution, for instance if $x$ is a variable of type $(list\ n)$ the term $\lambda x\ x$ has type $(list\ n) \to (list\ n)$, but substituting $n$ by 4 changes the type of $x$ to $(list\ 4)$ and the type of $\lambda x\ x$ to $(list\ 4) \to (list\ 4)$. In such a system, we usually indicate the type of each variable by a subscript at its binding occurrence, writing, for instance $\lambda x_{(list\ n)}\ x$.

#### *7.1.2. Types Parametrized by Types*

In the same way, we may want to parametrize the type *list* by the type of the elements of the list, in order to construct lists of natural numbers, lists of sets of natural numbers, lists of lists of natural numbers, etc. i.e. we want to have a family of type $(list\ nat)$, $(list\ (nat \to o))$, $(list\ (list\ nat))$, etc. parametrized by a type. When we have such types parametrized by types we need also to parametrize terms by types, i.e. to have terms taking a type as argument, for instance the symbol

$\varepsilon$ must be parametrized by a type in such a way that $(\varepsilon\ nat)$ be a term of type $(list\ nat)$, $(\varepsilon\ (nat \to o))$ a term of type $(list\ (nat \to o))$, etc.

Taking none, one or several of the three features: types parametrized by terms (dependent types), types parametrized by types (type constructors), terms parametrized by types (polymorphic types), we get $2^3 = 8$ calculi ($\lambda\Pi$-calculus [Harper, Honsell and Plotkin 1993], systems $F$ and $F_\omega$ [Girard 1970, Girard 1972], the Calculus of constructions [Coquand 1985, Coquand and Huet 1988], etc.) that are usually represented as the vertices of a cube [Barendregt 1992].

### 7.1.3. *Proofs as Objects*

These extensions of simply typed $\lambda$-calculus are needed when we consider extensions of higher-order logic where proofs are objects. In higher-order logic, the number 2 is expressed by a term, the set $E$ of even numbers too, the proposition $(E\ 2)$ that 2 is even also, but the proof that this proposition holds is not a term. *Intuitionistic type theory* [Martin-Löf 1984] and the *Calculus of Constructions* [Coquand 1985, Coquand and Huet 1988] are extensions of higher-order logic where such proofs are terms of the formalism too.

These formalisms use Brouwer-Heyting-Kolmogorov notion of proof : proofs of atomic propositions are atoms, proofs of propositions of the form $A \Rightarrow B$ are functions mapping proofs of $A$ to proofs of $B$ (for instance, the term $\lambda x_{P'}\ \lambda y_{Q'}\ x$ is a proof of $P \Rightarrow Q \Rightarrow P$) and proofs of propositions of the form $\forall x_T\ P$ are functions mapping every object $a$ of the type $T$ to a proof of $(a/x)P$.

As remarked by H.B. Curry [Curry and Feys 1958], N.G. de Bruijn [de Bruijn 1980] and W. Howard [Howard 1980], the type of such a term is isomorphic to the proposition itself, i.e. proofs of propositions of the form $A \Rightarrow B$ have type $A' \to B'$ where $A'$ is the type of proofs of $A$ and $B'$ the type of proofs of $B$. Proofs of propositions of the form $\forall x_T\ P$ have type $\Pi x_T\ P'$ where $P'$ the type of the proofs of $P$.

As usual, we identify isomorphic objects and thus identify $A \Rightarrow B$ and $A \to B$, $\forall x_T\ P$ and $\Pi x_T\ P$,

## 7.2. *Unification in $\lambda$-calculus with Dependent Types*

### 7.2.1. *$\lambda\Pi$-calculus*

The first unification algorithm for such an extension of simply typed $\lambda$-calculus has been proposed by C.M. Elliott [Elliott 1989, Elliott 1990] and D. Pym [Pym 1990] for $\lambda\Pi$-calculus i.e. a calculus where types may be parametrized by terms, but not by types and terms cannot be parametrized by types either. The main idea in this algorithm is still the same: simplify rigid-rigid equations, construct solutions to flexible-rigid equations incrementally with elementary substitutions, substituting variables by terms of the form $\lambda y_1\ _{T_1}\ ...\ \lambda y_p\ _{T_p}\ (h\ (H_1\ y_1\ ...\ y_p)\ ...\ (H_r\ y_1\ ...\ y_p))$ where $h$ is either a bound variable or the head variable of the rigid term, and avoid solving flexible-flexible equations that always have solutions.

The main difference concerns the typing of substitutions. In simply typed $\lambda$-calculus, if we have a variable $X$ of type $T \to U \to T$ and an equation $(X\ a\ b) = a$ then the potential elementary substitutions substitute the terms $\lambda x\ \lambda y\ x$, $\lambda x\ \lambda y\ y$, $\lambda x\ \lambda y\ a$ for the variable $X$. But the second term has type $T \to U \to U$ and thus cannot be substituted to $X$ (see definition 4.1). We select this way the elementary substitutions that are well-typed, i.e. replace a variable by a term of the same type.

In $\lambda\Pi$-calculus a type may contain variables and thus a type may be changed by a substitution. Thus, when applying a substitution $t/X$ we must not check that the type of $X$ and $t$ are the same, but we must unify them, or add an equation relating their types to the problem.

For instance, if the variable $X$ has type $(list\ 0) \to (list\ Y) \to (list\ 0)$ and we have the problem $(X\ \varepsilon\ b) = \varepsilon$, although the elementary substitution $\lambda x_{(list\ 0)} \lambda y_{(list\ Y)}\ y/X$ is not well typed (the variable has type $(list\ 0) \to (list\ Y) \to (list\ 0)$ and the term $(list\ 0) \to (list\ Y) \to (list\ Y)$) we must not reject it. Indeed, this substitution will be well-typed when we substitute the term 0 for the variable $Y$ leading to the solution $0/Y, \lambda x_{(list\ 0)} \lambda y_{(list\ 0)}\ y/X$. Thus we must consider all the potential elementary substitutions, well-typed or not, and when we perform such a substitution, we must add to the unification problem the *accounting equation* of this substitution, i.e. the equation relating the type of the variable and the type of the term.

In the example above the accounting equation is

$$(list\ 0) \to (list\ Y) \to (list\ Y) = (list\ 0) \to (list\ Y) \to (list\ 0)$$

and it simplifies to $Y = Y, Y = 0$.

As we consider ill-typed substitutions, we have to consider ill-typed, and thus potentially nonnormalizable, equations. In fact, Elliott and Pym have proved that, in $\lambda\Pi$-calculus, provided the simplification of the accounting equation succeeds, the equations, although ill-typed, always normalize [Elliott 1989, Elliott 1990, Pym 1990].

### *7.2.2. Polymorphism, Type Constructors, Inductive Types*

When we consider also polymorphic types and types constructors, i.e. terms parametrized by types and types parametrized by types, we still need accounting equations, but new phenomena happen: the number of arguments of the head variable in an elementary substitution is not fixed by its type anymore, for instance if the variable $h$ has type $\Pi x_{Type}\ x$ and we want to build a term of type $A$ we can build the term

$$(h\ (\underbrace{A \to ... \to A}_{n \text{ times}} \to A)\quad \underbrace{a\ ...\ a}_{n \text{ times}}\ )$$

where the variable $h$ has $n+1$ arguments. Thus we need to consider elementary substitutions where the number of arguments of the head variable is arbitrary [Dowek 1993*a*, Dowek 1991*a*].

Another difference is that flexible-flexible equation do not always have solutions, for instance if the variable $X$ has type $\Pi x_{Type}\ x$, the equation

$$(X\ (A \rightarrow B)\ a) = (X\ B)$$

has no solution [Dowek 1991*a*]. Thus we must enumerate the elementary substitutions for flexible-flexible equations too.

At last, we loose the normalization property for ill-typed equations but we can prove that in any situation there is always at least a variable that has a well-typed type and that we can instantiate.

Besides dependent types, polymorphic types and types constructors, we can also consider inductive types, i.e. reduction rules for recursor on some data types (see section 1.5 and 4.4) [Gödel 1958, Girard et al. 1989, Martin-Löf 1984, Paulin-Mohring 1993, Werner 1994] and extend the unification algorithm to these systems [Saïdi 1994, Cornes 1997].

### *7.3. Closed Solutions*

In $\lambda\Pi$-calculus we cannot assume anymore that every atomic type is inhabited. For instance consider a type family $(even\ 0)$, $(even\ 1)$, $(even\ 2)$, etc. such that $(even\ n)$ is the type of proofs that $n$ is even. When $n$ is odd, for instance for $n = 1$, this type must be empty.

Thus, flexible-flexible equations do not always have closed solutions (see section 4.2.6). Like in simply typed $\lambda$-calculus, the existence of a closed solution to a flexible-flexible unification problem is undecidable. In $\lambda\Pi$-calculus type inhabitation is undecidable (see [Bezem and Springintveld 1996]) and thus even unification problems with no equations (or unification problems on the form $X = X$) are undecidable.

Thus, when looking for closed solutions in $\lambda\Pi$-calculus, we cannot avoid solving flexible-flexible equations.

### *7.4. Automated Theorem Proving as Unification*

Using Curry-de Bruijn-Howard isomorphism, a provability problem in propositional minimal logic can be be expressed as a type inhabitation problem in simply typed $\lambda$-calculus and thus as an higher-order unification problem [Zaionc 1988]. In the same way a provability problem in first-order minimal logic can be expressed as a unification problem in $\lambda\Pi$-calculus [Hagiya 1991, Pfenning 1991*a*] and a provability problem in higher-order intuitionistic logic can be expressed as a unification problem in the Calculus of constructions [Dowek 1993*a*, Dowek 1991*a*].

Thus, in $\lambda$-calculi with dependent types, provided we search for closed solutions, unification is not a subroutine of automated theorem proving methods anymore, but automated theorem proving can be reduced to unification.

For instance, consider a context where we have

- a type symbol $T$, i.e. a symbol $T$ of type $Type$,

- two symbols $a$ and $b$ of type $T$,
- a binary relation $\leq$, of type $T \rightarrow T \rightarrow Type$,
- an axiom $h$: $\forall x_T \forall y_T \forall z_T ((x \leq y) \Rightarrow (y \leq z) \Rightarrow (x \leq z))$, i.e. a symbol $h$ of type $\Pi x_T \Pi y_T \Pi z_T ((x \leq y) \rightarrow (y \leq z) \rightarrow (x \leq z))$,
- two axioms $v$ and $w$: $a \leq b$ and $b \leq c$, i.e. two symbols $v$ and $w$ of type $a \leq b$ and $b \leq c$.

We search a proof of the proposition $a \leq c$. We start with a variable $X$ of type $a \leq c$ and no equation (or the equation $X = X$). As we have no equation, all the equations are flexible-flexible, thus we must try all the possible head variables for term substituted for $X$, among them we consider the substitution $(h\ Y_1\ Y_2\ Y_3\ Y_4\ Y_5)/X$ introducing variables $Y_1$, $Y_2$ and $Y_3$ of type $T$, $Y_4$ of type $Y_1 \leq Y_2$ and $Y_5$ of type $Y_2 \leq Y_3$. The accounting equation of this substitution is

$$(a \leq c) = (Y_1 \leq Y_3)$$

which simplifies to

$$a = Y_1$$

$$c = Y_3$$

These equations are flexible-rigid, the only possible elementary substitution for $Y_1$ is $a/Y_1$ and the only possible one for $Y_3$ is $c/Y_3$. The accounting equations of these substitution are trivial ($T = T$), thus we get a problem with three variables ($Y_2$, $Y_4$, $Y_5$) and no equations.

Instantiating, for instance, the variable $Y_5$ we must try all the possible head variables, among them we consider the substitution $w/Y_5$ leading to the accounting equation

$$(Y_2 \leq c) = (b \leq c)$$

and it simplifies to

$$Y_2 = b$$

This equation is flexible-rigid, the only possible solution for $Y_2$ is $b/Y_2$. The accounting equation of this substitution is trivial ($T = T$) and thus we get a problem with one variable ($Y_4$) and no equation.

Instantiating this variable, we must try all the possible head variables, among them we consider the substitution $v/Y_4$. The accounting equation of this substitution is trivial ($(a \leq b) = (a \leq b)$) and thus, we get a problem with no variables and no equations. We are done. The term substituted for $X$, i.e. the proof of the proposition $a \leq c$ is $(h\ a\ b\ c\ v\ w)$.

Here, all the proof search has been performed by the unification algorithm. Notice that the elementary substitutions $(h\ Y_1\ Y_2\ Y_3\ Y_4\ Y_5)/X$, $w/Y_5$, $v/Y_4$ would be considered as resolution steps in more traditional approaches while the elementary substitutions $a/Y_1$, $c/Y_3$ and $b/Y_2$ would be considered as genuine unification steps.

Patterns unification is decidable in all the calculi with dependent types [Pfenning 1991*b*]. Pattern matching is undecidable in most of the calculi with dependent types [Dowek 1991*b*, Dowek 1993*b*, Dowek 1991*a*] but second-order matching is decidable

[Dowek 1991*c*, Dowek 1991*a*], and third-order matching is decidable in some systems [Springintveld 1995*a*, Springintveld 1995*b*, Springintveld 1995*c*].

With dependent types also, the unification steps can be decomposed using explicit substitutions [Magnusson 1994, Muñoz 1997].

## Acknowledgments

I want to thank Gérard Huet who has initiated me into the theory of higher-order unification.

I also want to thank Peter Andrews, Jean Goubault-Larrecq, and Gopalan Nadathur for their comments on this chapter.

## Bibliography

ABADI M., CARDELLI L., CURIEN P.-L. AND LÉVY J.-J. [1991], 'Explicit substitutions', *Journal of Functional Programming* **1**(4), 375–416.

AMIOT G. [1990], 'The undecidability of the second order predicate unification problem', *Archive for mathematical logic* **30**, 193–199.

ANDREWS P. [2001], Classical type theory, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. II, Elsevier Science, chapter 15, pp. 965–1007.

ANDREWS P. B. [1971], 'Resolution in type theory', *The Journal of Symbolic Logic* **36**(3), 414–432.

ANDREWS P. B. [1986], *An introduction to mathematical logic and type theory: to truth through proof*, Academic Press.

AVENHAUS J. AND LORÍA-SÁENZ C. A. [1994], Higher-order conditional rewriting and narrowing, *in* J.-P. Jouannaud, ed., 'International Conference on Constaints in Computational Logic', Vol. 845 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 269–284.

BARENDREGT H. AND GEUVERS H. [2001], Proof-assistants using dependent type systems, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. II, Elsevier Science, chapter 18, pp. 1149–1238.

BARENDREGT H. P. [1984], *The Lambda-calculus, its syntax and semantics*, North Holland. Second edition.

BARENDREGT H. P. [1992], Lambda calculi with types, *in* S. Abramsky, D. M. Gabbay and T. S. E. Maibaum, eds, 'Handbook of logic in computer science', Vol. 2, Clarendon Press, pp. 118–309.

BAXTER L. D. [1977], The complexity of unification, PhD thesis, University of Waterloo.

BEZEM M. AND SPRINGINTVELD J. [1996], 'A simple proof of the undecidability of inhabitation in $\lambda P$', *Journal of Functional Programming* **6**(5), 757–761.

BOROVANSKÝ P. [1995], Implementation of higher-order unification based on calculus of explicit substitution, *in* M. Bartošek, J. Staudek and J. Wiedermann, eds, 'SOFSEM : Theory and Practice of Informatics', number 1012 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 363–368.

BOUDET A. AND CONTEJEAN E. [1997], AC-unification of higher-order patterns, *in* G. Smolka, ed., 'Principles and Practice of Constraint Programming', Vol. 1330 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 267–281.

CHURCH A. [1940], 'A formulation of the simple theory of types', *The Journal of Symbolic Logic* **5**(1), 56–68.

CHURCH A. [1956], *Introduction to mathematical logic*, Princeton University Press.

Comon H. [1998], 'Completion of rewrite systems with membership constraints. Part II: Constraint solving', *Journal of Symbolic Computation* **25**, 421–453.

Comon H. and Jurski Y. [1997], Higher-order matching and tree automata, *in* M. Nielsen and W. Thomas, eds, 'Conference on Computer Science Logic', Vol. 1414 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 157–176.

Coquand T. [1985], Une théorie des constructions. Thèse de troisième cycle, Université Paris VII.

Coquand T. and Huet G. [1988], 'The calculus of constructions', *Information and Computation* **76**, 95–120.

Cornes C. [1997], Conception d'un langage de haut niveau de représentation de preuves. récurrence par filtrage de motifs. unification en présence de types inductifs primitifs. synthèse de lemmes d'inversion. Thèse de Doctorat, Université de Paris VII.

Curien P.-L., Hardin T. and Lévy J.-J. [1996], 'Confluence properties of weak and strong calculi of explicit substitutions', *Journal of the Association for Computing Machinery* **43**(2), 362–397.

Curien R. [1995], Outils pour la preuve par analogie. Thèse de Doctorat, Université Henri Poincaré - Nancy I.

Curry H. B. [1942], 'The combinatory foundations of mathematical logic', *The Journal of Symbolic Logic* **7**(2), 49–64.

Curry H. B. and Feys R. [1958], *Combinatory logic*, Vol. 1, North Holland.

Dalrymple M., Shieber S. and Pereira F. [1991], 'Ellipsis and higher-order unification', *Linguistic and Philosophy* **14**, 399–452.

Davis M. [1969], Invited commentary of [Robinson 1969], *in* A. J. H. Morrell, ed., 'International Federation for Information Processing Congress, 1968', North Holland, pp. 67–68.

Davis M. [1973], 'Hilbert's tenth problem is unsolvable', *The American Mathematician Monthly* **80**(3), 233–269.

de Bruijn N. G. [1972], 'Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem', *Indagationes Mathematicae* **34**(5), 381–392.

de Bruijn N. G. [1980], A survey of the project automath, *in* J. R. Hindley and J. P. Seldin, eds, 'To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism', Academic Press.

Dougherty D. J. [1993], 'Higher-order unification via combinators', *Theoretical Computer Science* **114**, 273–298.

Dougherty D. J. and Johann P. [1992], A combinatory logic approach to higher-order E-unification, *in* D. Kapur, ed., 'Conference on Automated Deduction', Vol. 607 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 79–93.

Dowek G. [1991*a*], Démonstration automatique dans le calcul des constructions. Thèse de Doctorat, Université de Paris VII.

Dowek G. [1991*b*], 'L'indécidabilité du filtrage du troisième ordre dans les calculs avec types dépendants ou constructeurs de types (the undecidability of third order pattern matching in calculi with dependent types or type constructors)', *Comptes Rendus à l'Académie des Sciences* **I, 312**(12), 951–956. Erratum, ibid. I, 318, 1994, p. 873.

Dowek G. [1991*c*], A second-order pattern matching algorithm in the cube of typed λ-calculi, *in* A. Tarlecki, ed., 'Mathematical Foundation of Computer Science', Vol. 520 of *Lecture notes in computer science*, Springer-Verlag, pp. 151–160.

Dowek G. [1993*a*], 'A complete proof synthesis method for the cube of type systems', *Journal of Logic and Computation* **3**(3), 287–315.

Dowek G. [1993*b*], 'The undecidability of pattern matching in calculi where primitive recursive functions are representable', *Theoretical Computer Science* **107**, 349–356.

DOWEK G. [1993*c*], The undecidability of typability in the lambda-pi-calculus, *in* M. Bezem and J. F. Groote, eds, 'Typed Lambda Calculi and Applications', number 664 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 139–145.

DOWEK G. [1993*d*], A unification algorithm for second-order linear terms. Manuscript.

DOWEK G. [1994], 'Third order matching is decidable', *Annals of Pure and Applied Logic* **69**, 135–155.

DOWEK G. [1995], Lambda-calculus, combinators and the comprehension scheme, *in* M. Dezani-Ciancagliani and G. Plotkin, eds, 'Typed Lambda Calculi and Applications', number 902 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 154–170.

DOWEK G., HARDIN T. AND KIRCHNER C. [1995], Higher-order unification via explicit substitutions, *in* 'Logic in Computer Science', pp. 366–374.

DOWEK G., HARDIN T. AND KIRCHNER C. [2001], 'HOL-lambda-sigma: an intentional first-order expression of higher-order logic', *Mathematical Structures in Computer Science* **11**, 1–25.

DOWEK G., HARDIN T., KIRCHNER C. AND PFENNING F. [1996], Unification via explicit substitutions: the case of higher-order patterns, *in* M. Maher, ed., 'Joint International Conference and Symposium on Logic Programming', The MIT Press, pp. 259–273.

EKMAN J. [1994], Normal proofs in set theory. Doctoral thesis, Chalmers University and University of Göteborg.

ELLIOTT C. M. [1989], Higher-order unification with dependent function types, *in* N. Dershowitz, ed., 'Internatinal Conference on Rewriting Techniques and Applications', Vol. 355 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 121–136.

ELLIOTT C. M. [1990], Extensions and applications of higher-order unification, PhD thesis, Carnegie Mellon University.

FARMER W. M. [1988], 'A unification algorithm for second-order monadic terms', *Annals of Pure and applied Logic* **39**, 131–174.

FARMER W. M. [1991*a*], 'Simple second order languages for which unification is undecidable', *Theoretical Computer Science* **87**, 25–41.

FARMER W. M. [1991*b*], 'A unification theoretic method for investigating the *k*-provability problem', *Annals of Pure and Applied Logic* **51**, 173–214.

FRIEDMAN H. [1975], Equality between functionals, *in* R. Parikh, ed., 'Logic Colloquium', Vol. 453 of *Lecture Notes in Mathematics*, Springer-Verlag, pp. 23–37.

GIRARD J.-Y. [1970], Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types, *in* J. E. Fenstad, ed., 'Scandinavian Logic Symposium', North Holland.

GIRARD J.-Y. [1972], Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Thèse d'État, Université de Paris VII.

GIRARD J.-Y., LAFONT Y. AND TAYLOR P. [1989], *Proofs and Types*, Cambridge University Press.

GÖDEL K. [1958], 'Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes', *Dialectica* **12**.

GOLDFARB W. D. [1981], 'The undecidability of the second-order unification problem', *Theoretical Computer Science* **13**, 225–230.

GOUBAULT J. [1994], Higher-order rigid E-unification, *in* F. Pfenning, ed., '5th International Conference on Logic Programming and Automated Reasoning', number 822 *in* 'Lecture Notes in Artificial Intelligence', Springer-Verlag, pp. 129–143.

HAGIYA M. [1990], Programming by example and proving by example using higher-order unification, *in* M. Stickel, ed., 'Conference on Automated Deduction', number 449 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 588–602.

HAGIYA M. [1991], Higher-order unification as a theorem proving procedure, *in* K. Furukawa, ed., 'International Conference on Logic Programming', MIT Press, pp. 270–284.

HALLNÄS L. [1983], On normalization of proofs in set theory. Doctoral thesis, University of Stockholm.

Hannan J. and Miller D. [1988], Uses of higher-order unification for implementing programs transformers, *in* R. K. an K.A. Bowen, ed., 'International Conference and Symposium on Logic Programming', pp. 942–959.

Harper R., Honsell F. and Plotkin G. [1993], 'A framework for defining logics', *Journal of the Association for Computing Machinery* **40**(1), 143–184.

Henkin L. [1953], 'Banishing the rule of substitution for functional variables', *The Journal of Symbolic Logic* **18**(3), 201–208.

Hindley J. R. and Seldin J. P. [1986], *Introduction to combinators and λ-calculus*, Cambridge University Press.

Howard W. A. [1980], The formulæ-as-type notion of construction, *in* J. R. Hindley and J. P. Seldin, eds, 'To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism', Academic Press.

Huet G. [1972], Constrained resolution: a complete method for higher order logic, PhD thesis, Case Western University.

Huet G. [1973*a*], A mechanization of type theory, *in* 'International Joint Conference on Artificial Intelligence', pp. 139–146.

Huet G. [1973*b*], 'The undecidability of unification in third order logic', *Information and Control* **22**, 257–267.

Huet G. [1975], 'A unification algorithm for typed λ-calculus', *Theoretical Computer Science* **1**, 27–57.

Huet G. [1976], Résolution d'équations dans les langages d'ordre 1,2, ..., $\omega$. Thèse d'État, Université de Paris VII.

Huet G. and Lang B. [1978], 'Proving and applying program transformations expressed with second order patterns', *Acta Informatica* **11**, 31–55.

Hughes R. [1982], Super-combinators, a new implementation method for applicative languages, *in* 'Lisp and Functional Programming', pp. 1–10.

Johann P. and Kohlhase M. [1994], Unification in an extensional lambda calculus with ordered function sorts and constant overloading, *in* A. Bundy, ed., 'Conference on Automated Deduction', number 814 *in* 'Lecture Notes in Artificial Intelligence', Springer-Verlag, pp. 620–634.

Johnsson T. [1985], Lambda lifting: transforming programs to recursive equations, *in* J.-P. Jouannaud, ed., 'Functional Programming Languages and Computer Architecture', number 201 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 190–203.

Kirchner C. and Ringeissen C. [1997], Higher-order equational unification via explicit substitutions, *in* M. Hanus, J. Heering and K. Meinke, eds, 'Algebraic and Logic Programming, International Joint Conference ALP'97-HOA'97', Vol. 1298 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 61–75.

Krivine J.-L. [1993], *Lambda calculus, types and models*, Ellis Horwood series in computer and their applications.

Levy J. [1996], Linear second order unification, *in* H. Ganzinger, ed., 'Rewriting Techniques and Applications', Vol. 1103 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 332–346.

Levy J. and Veanes M. [1998], On unification problems in restricted second-order languages. Manuscript.

Levy J. and Villaret M. [2000], Linear second-order unification and context unification with tree-regular constraints, *in* 'Proceedings of the 11th Int. Conf. on Rewriting Techniques and Applications (RTA'00)', Vol. 1833 of *Lecture Notes in Computer Science*, Springer-Verlag, Norwich, UK, pp. 156–171.

Loader R. [1994], The undecidability of λ-definability. To appear in Church memorial volume.

Lucchesi C. L. [1972], The undecidability of the unification problem for third order languages, Technical Report CSRR 2060, Department of applied analysis and computer science, University of Waterloo.

Magnusson L. [1994], The implementation of ALF, a proof editor based on Martin-Löf monomorphic type theory with explicit substitution. Doctoral thesis, Chalmers University and University of Göteborg.

Martin-Löf P. [1984], *Intuitionistic type theory*, Bibliopolis.

Matiyacevich Y. [1970], 'Enumerable sets are diophantine', *Soviet Math. Doklady* **11**, 354–357.

Mayr R. and Nipkow T. [1998], 'Higher-order rewrite systems and their confluence', *Theoretical Computer Science* (192), 3–29.

Miller D. [1991], 'A logic programming language with lambda-abstraction, function variables, and simple unification', *Journal of Logic and Computation* **1**(4), 497–536.

Miller D. [1992], 'Unification under a mixed prefix', *Journal of Symbolic Computation* **14**, 321–358.

Miller D. and Nadathur G. [1986], Some uses of higher-order logic in computational linguistics, *in* 'Annual Meeting of the Association for Computational Linguistics', pp. 247–255.

Müller O. and Weber F. [1994], Theory and practice of minimal modular higher-order E-unification, *in* A. Bundy, ed., 'Conference on Automated Deduction', number 814 *in* 'Lecture Notes in Artificial Intelligence', Springer-Verlag, pp. 650–664.

Muñoz C. [1997], Un calcul de substitutions explicites pour la représentation de preuves partielles en théorie des types. Thèse de Doctorat, Université de Paris VII.

Nadathur G. [1998], An explicit substitution notation in a λprolog implementation, *in* 'First International Workshop on Explicit Substitutions'.

Nadathur G. and Miller D. [1998], Higher-order logic programming, *in* D. M. Gabbay, C. J. Hogger and J. A. Robinson, eds, 'Handbook of logic in artificial intelligence and logic programming', Vol. 5, Clarendon Press, pp. 499–590.

Nadathur G. and Mitchell D. [1999], System description : A compiler and abstract machine based implementation of λprolog, *in* 'Conference on Automated Deduction'.

Nadathur G. and Wilson D. [1998], 'A notation for lambda terms : A generalization of environments', *Theoretical Computer Science* **198**(1–2), 49–98.

Niehren J., Pinkal M. and Ruhrberg P. [1997], On equality up-to constraints over finite trees, context unification, and one-step rewriting, *in* 'Proceedings of the 14th Int. Conference on Automated Deduction (CADE-14)', Vol. 1249 of *Lecture Notes in Computer Science*, Springer-Verlag, Townsville, North Queensland, Australia, pp. 34–48.

Niehren J., Tison S. and Treinen R. [2000], 'On rewrite constraints and context unification', *Information Processing Letters* **74**(1-2), 35–40.

Nipkow T. [1991], Higher-order critical pairs, *in* 'Logic in Computer Science', pp. 342–349.

Nipkow T. and Prehofer C. [1998], Higher-order rewriting and equational reasonning, *in* W. Bibel and P. Schmitt, eds, 'Automated Deduction - A Basis for Applications', Vol. 1, Kluwer, pp. 399–430.

Nipkow T. and Qian Z. [1994], 'Reduction and unification in lambda calculi with a general notion of subtype', *Journal of Automated Reasoning* **12**, 389–406.

Padovani V. [1994], Fourth-order matching is decidable. Manuscript.

Padovani V. [1995], On equivalence classes of interpolation equations, *in* M. Dezani-Ciancagliani and G. Plotkin, eds, 'Typed Lambda Calculi and Applications', number 902 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 335–349.

Padovani V. [1996*a*], Decidability of all minimal models, *in* S. Berardi and M. Coppo, eds, 'Types for Proof and Programs 1995', number 1158 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 201–215.

Padovani V. [1996*b*], Filtrage d'ordre supérieur. Thèse de Doctorat, Université de Paris VII.

Parikh R. [1973], 'Some results on the length of proofs', *Transactions of the American Mathematical Society* **177**, 29–36.

Paulin-Mohring C. [1993], Inductive definitions in the system Coq, rules and properties, *in* M. Bezem and J. F. Groote, eds, 'Typed Lambda Calculi and Applications', Vol. 664 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 328–345.

Paulson L. C. [1991], Isabelle: The next 700 theorem provers, *in* P. Odifreddi, ed., 'Logic and computer science', Academic Press, pp. 361–385.

Pfenning F. [1988], Partial polymorphic type inference and higher-order unification, *in* 'Conference on Lisp and Functional Programming', pp. 153–163.

Pfenning F. [1991*a*], Logic programming in the LF logical framework, *in* G. Huet and G. Plotkin, eds, 'Logical frameworks', Cambridge University Press, pp. 149–181.

Pfenning F. [1991*b*], Unification and anti-unification in the calculus of constructions, *in* 'Logic in Computer Science', pp. 74–85.

Pfenning F. [2001], Logical frameworks, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. II, Elsevier Science, chapter 17, pp. 1063–1147.

Pfenning F. and Cervesato I. [1997], Linear higher-order pre-unification, *in* 'Logic in Computer Science'.

Plotkin G. [1972], 'Building-in equational theories', *Machine Intelligence* **7**, 73–90.

Prawitz D. [1968], 'Hauptsatz for higher order logic', *The Journal of Symbolic Logic* **33**, 452–457.

Prehofer C. [1994*a*], Decidable higher-order unification problems, *in* A. Bundy, ed., 'Conference on Automated Deduction', Vol. 814 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 635–649.

Prehofer C. [1994*b*], Higher-order narrowing, *in* 'Logic in Computer Science', pp. 507–516.

Prehofer C. [1995], Solving higher-order equations: from logic to programming. Doctoral thesis, Technische Universität München.

Pym D. [1990], Proof, search and computation in general logic. Doctoral thesis, University of Edinburgh.

Qian Z. [1994], Higher-order equational logic programming, *in* 'Principle of Programming Languages', pp. 254–267.

Qian Z. and Wang K. [1992], Higher-order equational E-unification for arbitrary theories, *in* K. Apt, ed., 'Joint International Conference and Symposium on Logic Programming'.

Qian Z. and Wang K. [1994], Modular AC unification of higher-order patterns, *in* J.-P. Jouannaud, ed., 'International Conference on Constaints in Computational Logic', Vol. 845 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 105–120.

Quine W. V. O. [1969], *Set theory and its logic*, Belknap Press.

Robinson J. A. [1969], New directions in mechanical theorem proving, *in* A. J. H. Morrell, ed., 'International Federation for Information Processing Congress, 1968', North Holland, pp. 63–67.

Robinson J. A. [1970], 'A note on mechanizing higher order logic', *Machine Intelligence* **5**, 123–133.

Saïdi H. [1994], Résolution d'équations dans le système $T$ de Gödel. Mémoire de DEA, Université de Paris VII.

Schmidt-Schauss M. [1994], Unification of stratified second-order terms, Technical Report 12, J.W.Goethe-Universität, Frankfurt.

Schmidt-Schauss M. [1999], Decidability of bounded second order unification, Technical Report 11, J.W.Goethe-Universität, Frankfurt.

Schmidt-Schauss M. and Schulz K. [1999], Solvability of context equations with two context variables is decidable, *in* H. Ganzinger, ed., 'Conference on Automated Deduction', number 1632 *in* 'Lecture Notes in Artificial Intelligence', pp. 67–81.

Schubert A. [1997], Linear interpolation for the higher order matching problem, *in* M. Bidoit and M. Dauchet, eds, 'Theory and Practice of Software Development', Vol. 1214 of *Lecture Notes in Computer science*, Springer-Verlag, pp. 441–452.

Schubert A. [1998], Second-order unification and type inference for Church-style polymorphism, *in* 'Principle of Programming Languages', pp. 279–288.

Schwichtenberg H. [1976], 'Definierbare Funktionen im $\lambda$-Kalkül mit Typen', *Archiv Logik Grundlagenforschung* **17**, 113–114.

SNYDER W. [1990], Higher-order E-unification, *in* M. E. Stickel, ed., 'Conference on Automated Deduction', Vol. 449 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 573–587.

SNYDER W. AND GALLIER J. [1989], 'Higher order unification revisited: Complete sets of tranformations', *Journal of Symbolic Computation* **8**(1 & 2), 101–140. Special issue on unification. Part two.

SPRINGINTVELD J. [1995*a*], Algorithms for type theory. Doctoral thesis, Utrecht University.

SPRINGINTVELD J. [1995*b*], Third-order matching in presence of type constructors, *in* M. Dezani-Ciancagliani and G. Plotkin, eds, 'Typed Lambda Calculi and Applications', Vol. 902 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 428–442.

SPRINGINTVELD J. [1995*c*], Third-order matching in the polymorphic lambda calculus, *in* G. Dowek, J. Heering, K. Meinke and B. Möller, eds, 'Higher-order Algebra, Logic and Term Rewriting', Vol. 1074 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 221–237.

STATMAN R. [1982], 'Completeness, invariance and $\lambda$-definability', *The Journal of Symbolic Logic* **47**(1), 17–28.

STATMAN R. AND DOWEK G. [1992], On Statman's completeness theorem, Technical Report CMU-CS-92-152, Carnegie Mellon University.

TAKAHASHI M. O. [1967], 'A proof of cut-elimination in simple type theory', *Journal of the Mathematical Society of Japan* **19**, 399–410.

WERNER B. [1994], Une théorie des constructions inductives. Thèse de Doctorat, Université de Paris VII.

WHITEHEAD A. N. AND RUSSELL B. [1910-1913, 1925-1927], *Principia mathematica*, Cambridge University Press.

WOLFRAM D. A. [1989], The clausal theory of types. Doctoral thesis, University of Cambridge.

ZAIONC M. [1987], 'The regular expression description of unifier set in the typed $\lambda$-calculus', *Fundementa Informaticae* **X**, 309–322.

ZAIONC M. [1988], 'Mechanical procedure for proof construction via closed terms in typed $\lambda$-calculus', *Journal of Automated Reasoning* **4**, 173–190.

# Index