Automated theorem proving in first-order logic modulo: on the difference between type theory and set theory

Gilles Dowek

Automated theorem proving in first-order logic modulo: on the difference between type theory and set theory

Gilles Dowek

INRIA-Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France.
Gilles.Dowek@inria.fr http://coq.inria.fr/~dowek

Abstract. *Resolution modulo* is a first-order theorem proving method that can be applied both to first-order presentations of simple type theory (also called higher-order logic) and to set theory. When it is applied to some first-order presentations of type theory, it simulates exactly higher-order resolution. In this note, we compare how it behaves on type theory and on set theory.

Higher-order theorem proving (e.g. higher-order resolution [1, 17, 18]) is different from first-order theorem proving in several respects. First, the first-order unification algorithm has to be replaced by the higher-order one [19, 20]. Even then, the resolution rule alone is not complete but another rule called *the splitting rule* has to be added. At last, the skolemization rule is more complicated [24, 25].

On the other hand, higher-order logic, also called simple type theory, can be expressed as a first-order theory [7], and first-order theorem proving methods, such as first-order resolution, can be used for this theory. Of course, first-order resolution with the axioms of this theory is much less efficient than higher-order resolution. However, we can try to understand higher-order resolution as a special automated theorem proving method designed for this theory. A motivation for this project is that it is very unlikely that such a method applies only to this theory, but it should also apply to similar theories such as extensions of type theory with primitive recursion or set theory.

In [11], together with Th. Hardin and C. Kirchner, we have proposed a theorem proving method for first-order logic, called *resolution modulo*, that when applied to a first-order presentation of type theory simulates exactly higher-order resolution. Proving the completeness of this method has required to introduce a new presentation of first-order logic, called *deduction modulo* that separates clearly computation steps and deduction steps.

Resolution modulo can be applied both to type theory and to set theory. The goal of this note is to compare how resolution modulo works for one theory

and the other. In order to remain self contained, we will first present shortly the ideas of deduction modulo and resolution modulo.

1 Resolution modulo

1.1 Deduction modulo

In deduction modulo, the notions of language, term and proposition are that of (many sorted) first-order logic. But, a theory is formed with a set of axioms Γ *and a congruence* $\equiv$ defined on propositions. In this paper, all congruences will be defined by confluent rewrite systems (as these rewrite systems are defined on propositions and propositions contain binders, these rewrite systems are in fact *combinatory reduction systems* [23]). Propositions are supposed to be identified modulo the congruence $\equiv$. Hence, the deduction rules must take into account this equivalence. For instance, the *modus ponens* cannot be stated as usual

$$\frac{A \Rightarrow B \quad A}{B}$$

but, as the two occurrences of A need not be identical, but need only to be congruent, it must be stated

$$\frac{A' \Rightarrow B \quad A}{B} \text{ if } A \equiv A'$$

In fact, as the congruence may identify implications with other propositions, a slightly more general formulation is needed

$$\frac{C \quad A}{B} \text{ if } C \equiv A \Rightarrow B$$

All the rules of natural deduction or sequent calculus may be stated in a similar way, see [11, 13] for more details.

As an example, in arithmetic, in natural deduction modulo, we can prove that 4 is an even number:

$$\dfrac{\dfrac{\dfrac{}{\forall x\ x = x}\text{ axiom}}{2 \times 2 = 4}\ (x, x = x, 4)\ \forall\text{-elim}}{\exists x\ 2 \times x = 4}\ (x, 2 \times x = 4, 2)\ \exists\text{-intro}$$

Substituting the variable x by the term 2 in the proposition $2 \times x = 4$ yields the proposition $2 \times 2 = 4$, that is congruent to $4 = 4$. The transformation of one proposition into the other, that requires several proof steps in natural deduction, is dropped from the proof in deduction modulo. It is just a computation that need not be written, because everybody can re-do it by him/herself.

In this case, the congruence can be defined by a rewriting system defined on terms

$$0 + y \longrightarrow y$$

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

$$0 \times y \longrightarrow 0$$

$$S(x) \times y \longrightarrow x \times y + y$$

Notice that, in the proof above, we do not need the axioms of addition and multiplication. Indeed, these axioms are now redundant: since the terms $0 + y$ and y are congruents, the axiom $\forall y\ 0 + y = y$ is congruent to the equality axiom $\forall y\ y = y$. Hence, it can be dropped. In other words, this axiom has been built-in the congruence [26, 1, 30].

The originality of deduction modulo is that we have introduced the possibility to define the congruence directly on propositions with rules rewriting atomic propositions to arbitrary ones. For instance, in the theory of integral rings, we can take the rule

$$x \times y = 0 \longrightarrow x = 0 \vee y = 0$$

that rewrites an atomic proposition to a disjunction.

Notice, at last, that deduction modulo is not a true extension of first-order logic. Indeed, it is proved in [11] that for every congruence $\equiv$, we can find a theory $\mathcal{T}$ such that $\Gamma \vdash P$ is provable modulo $\equiv$ if and only if $\mathcal{T}\Gamma \vdash P$ is provable in ordinary first-order logic. Of course, the provable propositions are the same, but the proofs are very different.

1.2 Resolution modulo

When the congruence on propositions is induced by a congruence on terms, automated theorem proving can be performed like in first-order logic, for instance with the resolution method, provided the unification algorithm is replaced by an *equational unification* algorithm modulo this congruence. Equational unification problems can be solved by the *narrowing* method [15, 21, 22]. The method obtained this way, called *equational resolution* [26, 30], is complete.

The situation is different when the congruence identifies atomic propositions with non atomic ones. For instance, in the theory of integral rings, the proposition

$$a \times a = 0 \Rightarrow a = 0$$

is provable because it reduces to

$$(a = 0 \vee a = 0) \Rightarrow a = 0$$

Hence the proposition

$$\exists y\ (a \times a = y \Rightarrow a = y)$$

is also provable. But, with the clausal form of its negation

$$a \times a = Y$$

$$\neg a = Z$$

we cannot apply the resolution rule successfully, because the terms $a \times a$ and a do not unify.

Hence, we need to introduce a new rule that detects that the literal $a \times a = Y$ has an instance that is reducible by the rewrite rule

$$x \times y = 0 \longrightarrow x = 0 \vee y = 0$$

instantiates it, reduces it and puts it in clausal form again. We get this way the clause

$$a = 0$$

that can be resolved with the clause $\neg a = Z$.

Hence, the rewrite rules have to be divided into two sets: the set $\mathcal{E}$ of rules rewriting terms to terms that are used by the equational unification algorithm and the set of rule $\mathcal{R}$ rewriting atomic propositions to arbitrary ones and that are used by this new rule called *extended narrowing*. The system obtained this way is called *extended narrowing and resolution* or simply *resolution modulo*. Figure 1 gives a formulation of this method where unification problems are postponed as constraints. A proposition is said to be provable with this method when, from

Extended resolution:

$$\frac{\{A_1, \ldots, A_n, B_1, \ldots, B_m\}/E_1 \quad \{\neg C_1, \ldots, \neg C_p, D_1, \ldots, D_q\}/E_2}{\{B_1, \ldots, B_m, D_1, \ldots, D_q\}/E_1 \cup E_2 \cup \{A_1 =_{\mathcal{E}} \ldots A_n =_{\mathcal{E}} C_1 =_{\mathcal{E}} \ldots C_p\}}$$

Extended narrowing:

$$\frac{C/E}{cl(C[r]_p)/(E \cup \{C_{|p} =_{\mathcal{E}} l\})} \text{ if } l \longrightarrow r \in \mathcal{R}$$

Fig. 1. Resolution modulo

the clausal form of its negation, we can deduce an empty clause constrained by a $\mathcal{E}$-unifiable set of equations.

Transforming axioms into rewrite rules enhances the efficiency of automated theorem proving as shown by this very simple example.

Example. To refute the theory $P_1 \Leftrightarrow (Q_2 \vee P_2)$, ..., $P_i \Leftrightarrow (Q_{i+1} \vee P_{i+1})$, ..., $P_n \Leftrightarrow (Q_{n+1} \vee P_{n+1})$, P_1, $Q_2 \Leftrightarrow \bot$, ..., $Q_{n+1} \Leftrightarrow \bot$, $P_{n+1} \Leftrightarrow \bot$, resolution yields $4n + 2$ clauses

$$\neg P_1, Q_2, P_2$$

$$\neg Q_2, P_1$$

$$\neg P_2, P_1$$

$$\ldots$$

$$\neg P_i, Q_{i+1}, P_{i+1}$$

$$\neg Q_{i+1}, P_i$$
$$\neg P_{i+1}, P_i$$
$$...$$
$$\neg P_n, Q_{n+1}, P_{n+1}$$
$$\neg Q_{n+1}, P_{n+1}$$
$$\neg P_{n+1}, P_{n+1}$$
$$P_1$$
$$\neg Q_2$$
$$...$$
$$\neg Q_{n+1}$$
$$\neg P_{n+1}$$

While, in resolution modulo, the propositions $P_i \Leftrightarrow (Q_{i+1} \vee P_{i+1})$, $Q_i \Leftrightarrow \bot$ and $P_{n+1} \Leftrightarrow \bot$ can be transformed into rewrite rules

$$P_i \longrightarrow Q_{i+1} \vee P_{i+1}$$
$$Q_i \longrightarrow \bot$$
$$P_{n+1} \longrightarrow \bot$$

The only proposition left is P_1. It reduces to $\bot \vee ... \vee \bot$ and its clausal form is hence the empty clause.

Of course, reducing the proposition P_1 has a cost, but this cost is much lower than that of the non deterministic search of a refutation resolution with the clauses above. Indeed, the reduction process is deterministic because the rewrite system is confluent.

1.3 Cut elimination and completeness

Resolution modulo is not complete for all congruences. For instance, take the congruence induced by the rewrite rule

$$A \longrightarrow A \Rightarrow B$$

The proposition B has a proof in sequent calculus modulo

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\text{axiom} \quad \dfrac{\dfrac{}{B \vdash B}\,\text{axiom}}{A, B \vdash B}\,\text{weak.-left}}{A, A \vdash B}\,\Rightarrow\text{-left}}{A \vdash B}\,\text{contr.-left}}{\vdash A}\,\Rightarrow\text{-right} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\text{axiom} \quad \dfrac{\dfrac{}{B \vdash B}\,\text{axiom}}{A, B \vdash B}\,\text{weak.-left}}{A, A \vdash B}\,\Rightarrow\text{-left}}{A \vdash B}\,\text{contr.-left}}{\vdash B}\,\text{cut}$$

but it is not provable by resolution modulo. Indeed, the clausal form of the negation of the proposition B is the clause

$$\neg B$$

and neither the extended resolution rule nor the extended narrowing rule can be applied successfully.

However, it may be noticed that the proposition B has no cut free proof in sequent calculus modulo. Hence sequent calculus modulo this congruence does not have the cut elimination property. We have proved in [11] that resolution modulo is complete for all congruences $\equiv$ such that the sequent calculus modulo $\equiv$ has the cut elimination property. Together with B. Werner, we have proved in [13] that cut elimination holds modulo a large class of congruences and conjectured that it holds modulo all congruences that can be defined by a confluent and terminating rewrite system.

When cut elimination does not hold, only propositions that have a cut free proof are proved by resolution modulo.

2 Simple type theory and set theory

2.1 Simple type theory

Simple type theory is a many-sorted first-order theory. The sorts of simple type theory, called *simple types*, are defined inductively as follows.

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

As usual, we write $T_1 \to ... \to T_n \to U$ for the type $T_1 \to ... \ (T_n \to U)$.

The language of simple type theory contains the individual symbols

- $S_{T,U,V}$ of sort $(T \to U \to V) \to (T \to U) \to T \to V$,
- $K_{T,U}$ of sort $T \to U \to T$,
- $\dot{\vee}$ of sort $o \to o \to o$,
- $\dot{\neg}$ of sort $o \to o$,
- $\dot{\forall}_T$ of sort $(T \to o) \to o$,

the function symbols

- $\alpha_{T,U}$ of rank $(T \to U, T, U)$,

and the predicate symbol

- ε of rank (o).

As usual, we write $(t\ u)$ for the term $\alpha(t,u)$ and $(t\ u_1\ ...\ u_n)$ for $(...(t\ u_1)\ ...\ u_n)$.

Usual presentations of simple-type theory [6, 2] define propositions as terms of type o. But, as we want type theory to be a first-order theory, we introduce a predicate symbol ε that transforms a term of type o into a genuine proposition. Then, we need an axiom relating the proposition $\varepsilon(\alpha(\alpha(\dot{\vee}, x), y))$ and the proposition $\varepsilon(x) \vee \varepsilon(y)$. For instance, the axiom

$$\forall x\ \forall y\ (\varepsilon(\dot{\vee}\ x\ y) \Leftrightarrow (\varepsilon(x) \vee \varepsilon(y)))$$

This axiom can be built in the congruence, if we take the rewrite rule

$$\varepsilon(\dot{\vee}\ x\ y) \longrightarrow \varepsilon(x) \vee \varepsilon(y)$$

This leads to the rewrite system of the figure 2. This rewrite system is confluent

$$(S\ x\ y\ z) \longrightarrow (x\ z\ (y\ z))$$
$$(K\ x\ y) \longrightarrow x$$
$$\varepsilon(\dot{\neg}\ x) \longrightarrow \neg\varepsilon(x)$$
$$\varepsilon(\dot{\vee}\ x\ y) \longrightarrow \varepsilon(x) \vee \varepsilon(y)$$
$$\varepsilon(\dot{\forall}\ x) \longrightarrow \forall y\ \varepsilon(x\ y)$$

Fig. 2. Rewriting rules for simple type theory

because it is orthogonal and we prove in [10] that it is strongly normalizing. Hence, the congruence is decidable.

It is proved in [13] that deduction modulo this congruence has the cut elimination property, i.e. every proposition provable in sequent calculus modulo this congruence has a cut free proof.

2.2 Set theory

The language of Zermelo's set theory is formed with the binary predicate symbols $\in$ and $=$. This theory contains the axioms of equality and the following axioms.
pair:

$$\forall x\ \forall y\ \exists z\ \forall w\ (w \in z \Leftrightarrow (w = x \vee w = y))$$

union:

$$\forall x\ \exists y\ \forall w\ (w \in y \Leftrightarrow \exists z\ (w \in z \wedge z \in x))$$

power set:

$$\forall x\ \exists y\ \forall w\ (w \in y \Leftrightarrow \forall z\ (z \in w \Rightarrow z \in x))$$

subset scheme:

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

where $x_1, ..., x_n$ are the free variables of P minus w.
To these axioms, we may add the extensionality axiom, the foundation axiom, the axiom of infinity, the replacement scheme and the axiom of choice.

To have a language for the objects of the theory we may skolemize these axioms introducing the function symbols $\{\}$, $\bigcup$, $\mathcal{P}$ and $f_{x_1,...,x_n,w,P}$. We then get the axioms

$$\forall x\ \forall y\ \forall w\ (w \in \{\}(x,y) \Leftrightarrow (w = x \vee w = y))$$

$$\forall x\ \forall w\ (w \in \bigcup(x) \Leftrightarrow \exists z\ (w \in z \wedge z \in x))$$

$$\forall x\ \forall w\ (w \in \mathcal{P}(x) \Leftrightarrow \forall z\ (z \in w \Rightarrow z \in x))$$

$$\forall x_1 ... \forall x_n\ \forall y\ \forall w\ (w \in f_{x_1,...,x_n,w,P}(x_1, ..., x_n, y) \Leftrightarrow (w \in y \wedge P))$$

Then, these axioms may be built in the congruence with the rewrite system of figure 3. This rewrite system is confluent because it is orthogonal. But it does

$$w \in \{\}(x,y) \longrightarrow w = x \vee w = y$$
$$w \in \bigcup(x) \longrightarrow \exists z\ (w \in z \wedge z \in x)$$
$$w \in \mathcal{P}(x) \longrightarrow \forall z\ (z \in w \Rightarrow z \in x)$$
$$v \in f_{x_1,...,x_n,w,P}(y_1, ..., y_n, z) \longrightarrow v \in z \wedge [y_1/x_1, ..., y_n/x_n, v/w]P$$

Fig. 3. Rewriting rules for set theory

not terminate. A counter-example is M. Crabbé's proposition. Let C be the term $\{x \in a \mid \neg x \in x\}$ i.e. $f_{w,\neg w \in w}(a)$. We have

$$x \in C \longrightarrow x \in a \wedge \neg x \in x$$

Hence, writing A for the proposition $C \in C$ and B for the proposition $C \in a$ we have

$$A \longrightarrow B \wedge \neg A$$

This permits to construct the infinite reduction sequence

$$A \longrightarrow B \wedge \neg A \longrightarrow B \wedge \neg(B \wedge \neg A) \longrightarrow ...$$

Up to our knowledge, the decidability of this congruence is open.

Deduction modulo this congruence does not have the cut elimination property. A counter example is again Crabbé's proposition (see [16, 14] for a discussion). As we have seen, this proposition A rewrites to a proposition of the form $B \wedge \neg A$. Hence, the proposition $\neg B$ has the following proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\text{axiom}}{A, B \vdash A}\,\text{weakening-left}}{A, B, \neg A \vdash}\,\neg\text{-left}}{A, A \vdash}\,\wedge\text{-left}}{A \vdash}\,\text{contraction-left}}{\vdash \neg A}\,\neg\text{-right} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\text{axiom}}{A, B \vdash A}\,\text{weakening-left}}{A, B, \neg A \vdash}\,\neg\text{-left}}{A, A \vdash}\,\wedge\text{-left}}{A \vdash}\,\text{contraction-left}$$

$$\dfrac{\dfrac{\dfrac{}{B \vdash B}\,\text{axiom} \qquad \dfrac{\vdash \neg A}{B \vdash \neg A}\,\text{weakening-left}}{B \vdash A}\,\wedge\text{-right} \qquad \dfrac{A \vdash}{}\,\text{cut}}{\dfrac{B \vdash}{\vdash \neg B}\,\neg\text{-right}}$$

but it is easy to check that the proposition $\neg B$, i.e. $\neg f_{w,\neg w \in w}(a) \in a$, has no cut free proof.

3 Resolution modulo in type theory and in set theory

3.1 Resolution modulo in type theory

In the rewrite system of figure 2, the first two rules

$$(S\ x\ y\ z) \longrightarrow (x\ z\ (y\ z))$$

$$(K\ x\ y) \longrightarrow x$$

rewrite terms to terms and are used by the unification algorithm. The three others

$$\varepsilon(\dot{\neg}\ x) \longrightarrow \neg\varepsilon(x)$$

$$\varepsilon(\dot{\vee}\ x\ y) \longrightarrow \varepsilon(x) \vee \varepsilon(y)$$

$$\varepsilon(\dot{\forall}\ x) \longrightarrow \forall y\ \varepsilon(x\ y)$$

rewrite propositions to propositions and are used by the extended narrowing rule.

Equational unification modulo the rules S and K is related to higher-order unification. Actually since the reduction of combinators is slightly weaker than the reduction of λ-calculus, unification modulo this reduction is slightly weaker than higher-order unification [8]. To have genuine higher-order unification, we have to take another formulation of type theory using explicit substitutions instead of combinators (see section 5).

The extended narrowing modulo the rules $\dot{\neg}$, $\dot{\vee}$ and $\dot{\forall}$ is exactly the splitting rule of higher-order resolution. A normal literal unifies with the left member of such a rule if and only if its head symbol is a variable.

The skolemization rule in this language is related to the skolemization rule of type theory. When we skolemize a proposition of the form

$$\forall x\ \exists y\ P$$

we introduce a function symbol f of rank (T, U) where T is the type of x and U the type of y (not an individual symbol of type $T \to U$) and the axiom

$$\forall x\ [f(x)/y]P$$

Hence, the Skolem symbol f alone is not a term, but it permits to build a term of type U when we apply it to a term of type T. This is, in essence, the higher-order skolemization rule, but formulated for the language of combinators and not for λ-calculus. Again, we have the genuine higher-order skolemization rule if we use the formulation of type theory using explicit substitutions instead of combinators (see section 5).

3.2 Resolution modulo in set theory

In set theory, there is no rule rewriting terms to terms. Hence, unification in set theory is simply first-order unification. Conversely, all the rules of figure 3 rewrite propositions to propositions and thus the extended narrowing is performed modulo all theses rules.

In set theory, resolution modulo is incomplete. We have seen that the proposition

$$\neg f_{w,\neg w\in w}(a) \in a$$

has a proof in set theory, but it cannot be proved by the resolution modulo method. Indeed, from the clausal form of its negation

$$f_{w,\neg w\in w}(a) \in a$$

we can apply neither the resolution rule nor the extended narrowing rule successfully.

4 On the differences between set theory and type theory

4.1 Termination

The first difference between resolution modulo in type theory and in set theory is that the rewrite system is terminating in type theory and hence all propositions have a normal form, while some propositions, e.g. Crabbé's proposition, have no normal form in set theory.

Hence, during proof search, we can normalize all the clauses while this is impossible in set theory. Formally, the method modified this way requires a completeness proof.

4.2 Completeness

Another difference is that, as type theory verifies the cut elimination property, resolution modulo this congruence is complete, while it is incomplete modulo the congruence of set theory.

A solution to recover completeness may be to use an automated theorem proving method that searches for proofs containing cuts. For instance if we add a rule allowing to refute the set of clauses S by refuting both the set $S \cup \{\neg P\}$ and the set $\{P\}$ then we can refute the proposition B above.

Another direction is to search for another presentation of set theory or for a restriction of this theory that enjoys termination and cut elimination. We conjecture that if we restrict the subset scheme to *stratifiable* propositions in the sense of W.V.O. Quine [27], we get a restriction of set theory that is sufficient to express most mathematics, that terminates and that verifies the cut elimination property. The cut elimination and completeness results obtained by S.C. Bailin [4, 5] for his formulation of set theory let this conjecture be plausible.

4.3 Typing literals

A minor difference is that when we try to prove a theorem of the form "for all natural numbers x, $P(x)$", we have to formalize this theorem by the proposition

$$\forall x \ (x \in \mathbb{N} \Rightarrow P(x))$$

in set theory. In contrast, in type theory, we can choose to take ι for the type of natural numbers and state the theorem

$$\forall x \ P(x)$$

During the search, in set theory, extra literals of the form $x \in \mathbb{N}$ appear and have to be resolved.

4.4 The role of unification and extended narrowing

In resolution modulo, like in most other methods, the main difficulty is to construct the terms that have to be substituted to the variables. In resolution modulo, these terms are constructed by two processes: the unification algorithm and the extended narrowing rule.

The main difference between resolution modulo in type theory and in set theory is the division of work between the unification and the extended narrowing. In type theory, unification is quite powerful and the extended narrowing is rarely used. In contrast, in set theory, unification is simply first-order unification and all the work is done by the extended narrowing rule.

This difference reflects a deep difference on how mathematics are formalized in a theory and the other. Indeed, the unification in type theory is rich because there are rules that rewrite terms to terms and these rules are there because the notion of function is primitive in type theory. When we have a function f and an object a we can form the term $(f \ a)$ and start rewriting this term to a normal form. In set theory, there is no such term and a term alone can never be reduced. Instead of forming the term $(f \ a)$ we can form a proposition expressing that b is the image of a by the function f, $< a, b > \in f$, that then can be rewritten.

For example, in the proof of Cantor's theorem we have a function f from a set B to its power set and we want to form Cantor's set of objects that do not belong to their image.

If x is an element of B, in type theory we can express its image $(f\ x)$, then the term of type o reflecting the proposition expressing that x belongs to its image $(f\ x\ x)$, the term of type o reflecting its negation $\dot{\neg}(f\ x\ x)$ and then Cantor's set $\lambda x\ \dot{\neg}(f\ x\ x)$ that, with combinators, is expressed by the term

$$C = (S\ (K\ \dot{\neg})\ (S\ (S\ (K\ f)\ (S\ K\ K))\ (S\ K\ K)))$$

In contrast, in set theory, we cannot form a term expressing the image of x by the function f. Instead of saying that x does not belong to its image we have to say that it does not belong to any objet that happens to be its image.

$$C = \{x \in B \mid \forall y\ (< x, y > \in f \Rightarrow \neg x \in y)\}$$

This requires to introduce two more logical symbols $\Rightarrow$ and $\forall$. These symbols cannot be generated by the unification algorithm and are generated by the extended narrowing rule.

It is not completely clear what is the best division of work between unification and extended narrowing. Experiences with type theory show that the unification algorithm is usually well controlled while the splitting rule is very productive. Loading the unification and unloading the extended narrowing seems to improve efficiency.

However, two remarks moderate this point of view. First, in type theory, the functions that can be expressed by a term are very few. For instance, if we take the type ι for the natural numbers and introduce two symbols O and $Succ$ for zero and the successor function, we can only express by a term the constant functions and the functions adding a constant to their argument. The other functions are usually expressed with the description operator (or the choice operator) and hence as relations. We may enrich the language of combinators and the rewrite system, for instance with primitive recursion, but then it is not obvious that unification is still so well controlled.

Another remark is that having a decidable and unitary unification (such as first-order unification) permits to solve unification problems on the fly instead of keeping them as constraints. This permits to restrict the use of the extended narrowing rule. For instance, in type theory, when we have a literal $\varepsilon(P\ x)$ and we apply the extended narrowing rule yielding two literals $\varepsilon(A)$ and $\varepsilon(B)$ and a constraint

$$\varepsilon(P\ x) = \varepsilon(A \dot{\vee} B)$$

we keep this constraint frozen and we may need to apply the extended narrowing rule to other literals starting with the variable P. In contrast, in set theory, if we have a literal $x \in P$ and we apply the extended narrowing rule yielding two literals $y = a$ and $y = b$ and a constraint $(x \in P) = (y \in \{a, b\})$. The substitution $\{a, b\}/P$ can be immediately propagated to all the occurrences of P initiating reductions that let the extended narrowing steps be useless.

5 Advanced formulations of type theory and set theory

As an illustration of this discussion, we want to compare resolution modulo proofs of Cantor's theorem in type theory and in set theory. However, the presentations of type theory and set theory above are a little too rough to be really practicable. In both cases, we shall use a more sophisticated presentation where the language contains a full binding operator.

Indeed, in type theory, we want to express Cantor's set by the term

$$C = \lambda x\ \dot{\neg}(f\ x\ x)$$

and not by the term

$$C = (S\ (K\ \dot{\neg})\ (S\ (S\ (K\ f)\ (S\ K\ K))\ (S\ K\ K)))$$

Similarily, in set theory we want to express this set as

$$C = \{x \in B \mid \forall y\ (< x, y > \in R \Rightarrow \neg x \in y)\}$$

where $< x, y >$ is a notation for the set $\{\{x, y\}, \{x\}\}$ i.e. $\{\}(\{\}(x, y), \{\}(x, x))$, and not by the term

$$C = \{x \in B \mid \forall y\ (\forall u\ ((\forall v\ (v \in u \Leftrightarrow (\forall w\ (w \in v \Leftrightarrow (w = x \vee w = y)) \vee \forall w\ (w \in v \Leftrightarrow w = x)))) \Rightarrow u \in R) \Rightarrow \neg x \in y)\}$$ [1]

For type theory, such a first-order presentation with a general binding operator has been proposed in [12]. It uses an expression of λ-calculus as a first-order language based on de Bruijn indices and explicit substitutions. In this presentation, the sorts are of the form $\Gamma \vdash T$ or $\Gamma \vdash \Delta$ where T is a simple type and Γ and Δ are finite sequences of simple types. The language contains the following symbols

- 1_A^Γ of sort $A\Gamma \vdash A$,
- $\alpha_{A,B}^\Gamma$ of rank $(\Gamma \vdash A \to B, \Gamma \vdash A, \Gamma \vdash B)$,
- $\lambda_{A,B}^\Gamma$ of rank $(A\Gamma \vdash B, \Gamma \vdash A \to B)$,
- $[]_A^{\Gamma,\Gamma'}$ of rank $\Gamma' \vdash A, \Gamma \vdash \Gamma', \Gamma \vdash A)$,
- id^Γ of sort $\Gamma \vdash \Gamma$,
- $\uparrow_A^\Gamma$ of sort $A\Gamma \vdash \Gamma$,

[1] In the presentation of set theory above, there is no instance of the subset scheme for the proposition

$$\forall y\ (< x, y > \in R \Rightarrow \neg x \in y)$$

because it contains Skolem symbols. Hence, we replace the proposition $< x, y > \in R$ by the equivalent one

$$\forall u\ ((\forall v\ (v \in u \Leftrightarrow (\forall w\ (w \in v \Leftrightarrow (w = x \vee w = y)) \vee \forall w\ (w \in v \Leftrightarrow w = x)))) \Rightarrow u \in R)$$

Then we can build the set C with the function symbol introduced by the skolemization of this instance of the scheme. The proposition $x \in C$ is then provably equivalent to $x \in B \wedge \forall y\ (< x, y > \in R \Rightarrow \neg x \in y)$ but it does not reduce to it.

– $\dot{\cdot}_A^{\Gamma,\Gamma'}$ of rank $(\Gamma \vdash A, \Gamma \vdash \Gamma', \Gamma \vdash A\Gamma')$,
– $\circ^{\Gamma,\Gamma',\Gamma''}$ of rank $(\Gamma \vdash \Gamma'', \Gamma'' \vdash \Gamma', \Gamma \vdash \Gamma')$,
– $\dot{\vee}$ of sort $\vdash o \to o \to o$,
– $\dot{\neg}$ of sort $\vdash o \to o$,
– $\dot{\forall}_T$ of sort $\vdash (T \to o) \to o$,
– ε of rank $(\vdash o)$.

And the rewrite system is that of figure 4.

β-reduction and η-reduction:

$$(\lambda a)b \longrightarrow a[b.id]$$
$$\lambda(a\ 1) \longrightarrow b \text{ if } a =_\sigma b[\uparrow]$$

σ-reduction:

$$(a\ b)[s] \longrightarrow (a[s]\ b[s])$$
$$1[a.s] \longrightarrow a$$
$$a[id] \longrightarrow a$$
$$(\lambda a)[s] \longrightarrow \lambda(a[1.(s \circ \uparrow)])$$
$$(a[s])[t] \longrightarrow a[s \circ t]$$
$$id \circ s \longrightarrow s$$
$$\uparrow \circ (a.s) \longrightarrow s$$
$$(s_1 \circ s_2) \circ s_3 \longrightarrow s_1 \circ (s_2 \circ s_3)$$
$$(a.s) \circ t \longrightarrow a[t].(s \circ t)$$
$$s \circ id \longrightarrow s$$
$$1. \uparrow \longrightarrow id$$
$$1[s].(\uparrow \circ s) \longrightarrow s$$

reduction of propositions:

$$\varepsilon(\dot{\vee}\ x\ y) \longrightarrow \varepsilon(x) \vee \varepsilon(y)$$
$$\varepsilon(\dot{\neg}\ x) \longrightarrow \neg\varepsilon(x)$$
$$\varepsilon(\dot{\forall}_T\ x) \longrightarrow \forall y\ \varepsilon(x\ y)$$

Fig. 4. The rewrite rules of type theory with explicit substitutions

A formulation of set theory with a general binder has been given in [9]. But it is not expressed in a first-order setting yet. Waiting for such a theory, for the example of Cantor's theorem, we add a constant C and an *ad hoc* rewrite rule

$$x \in C \longrightarrow x \in B \wedge \forall y\ (< x, y > \in R \Rightarrow \neg x \in y)$$

6 Three proofs of Cantor's theorem

We now give three resolution modulo proofs of Cantor's theorem that there is no surjection from a set to its power set. The first is in type theory with a function expressing the potential surjection from a set to its power set. The second is also in type theory, but this potential surjection is expressed by a relation. The last one is in set theory and the surjection is, of course, expressed by a relation.

Automated theorem proving for Cantor's theorem in type theory is discussed in [17, 18, 3].

6.1 In type theory with a function

In type theory, a set is expressed by a term of type $T \to o$. Here, we choose to consider only the set of all objects of type ι. Its power set is the set of all objects of type $\iota \to o$. Hence we want to prove that there is no surjection from the type ι to $\iota \to o$. The first solution is to represent this potential surjection by a function f of type $\iota \to \iota \to o$. The surjectivity of this function can be expressed by the existence of a right-inverse g to this function, i.e. a function of type $(\iota \to o) \to \iota$ such that for all x, $(f\ (g\ x)) = x$. Using Leibniz' definition of equality this proposition is written

$$\forall x\ \forall p\ (\varepsilon(p\ (f\ (g\ x))) \Leftrightarrow \varepsilon(p\ x))$$

Putting this proposition in clausal form yields the clauses

$$\neg\varepsilon(P\ (f\ (g\ X))), \varepsilon(P\ X)$$

$$\varepsilon(Q\ (f\ (g\ Y))), \neg\varepsilon(Q\ Y)$$

The search is described on figure 5. It returns the empty clause constrained by

1	$\neg\varepsilon(P\ (f\ (g\ X))), \varepsilon(P\ X)$
2	$\varepsilon(Q\ (f\ (g\ Y))), \neg\varepsilon(Q\ Y)$
3 narr. (1)	$\neg\varepsilon(P\ (f\ (g\ X))), \neg\varepsilon(R)/c_1$
4 narr. (2)	$\varepsilon(Q\ (f\ (g\ Y))), \varepsilon(S)/c_2$
5 res. (3, 4)	$\Box/c_1, c_2, c_3, c_4, c_5$

with

$c_1\ (P\ X) = \dot{\neg}R$
$c_2\ (Q\ Y) = \dot{\neg}S$
$c_3\ (P\ (f\ (g\ X))) = R$
$c_4\ (Q\ (f\ (g\ Y))) = S$
$c_5\ (P\ (f\ (g\ X))) = (Q\ (f\ (g\ Y)))$

Fig. 5. Cantor's theorem in type theory with a function

the equations

$$\begin{array}{l}(P\ X) = \dot{\neg}R\\ (Q\ Y) = \dot{\neg}S\\ (P\ (f\ (g\ X))) = R\\ (Q\ (f\ (g\ Y))) = S\\ (P\ (f\ (g\ X))) = (Q\ (f\ (g\ Y)))\end{array}$$

that have the solution

$$\begin{array}{l}X = Y = \lambda\dot{\neg}[\uparrow](f[\uparrow]\ 1\ 1)\\ P = Q = \lambda(1\ (g[\uparrow]\ \lambda\dot{\neg}[\uparrow^2](f[\uparrow^2]\ 1\ 1)))\\ R = S = (f\ (g\ \lambda\dot{\neg}[\uparrow](f[\uparrow]\ 1\ 1))\ (g\ \lambda\dot{\neg}[\uparrow](f[\uparrow]\ 1\ 1)))\end{array}$$

6.2 In type theory with a relation

Instead of using the primitive notion of function of set theory, we can code the functions as functional relations R of type $\iota \to (\iota \to o) \to o$. The surjectivity and functionality of this relation are expressed by the propositions

$$E : \forall y\ \exists x\ \varepsilon(R\ x\ y)$$

and

$$F : \forall x\ \forall y\ \forall z\ (\varepsilon(R\ x\ y) \Rightarrow \varepsilon(R\ x\ z) \Rightarrow \forall p\ (\varepsilon(p\ y) \Leftrightarrow \varepsilon(p\ z)))$$

Putting these propositions in clausal form yields the clauses

$$\varepsilon(R\ g(U)\ U)$$

$$\neg\varepsilon(R\ X\ Y), \neg\varepsilon(R\ X\ Z), \neg\varepsilon(P\ Y), \varepsilon(P\ Z)$$

$$\neg\varepsilon(R\ X\ Y), \neg\varepsilon(R\ X\ Z), \neg\varepsilon(P\ Z), \varepsilon(P\ Y)$$

The search is then described on figure 6 where we simplify the constraints and substitute the solved constraints at each step. It returns the empty clause constrained by the equations

$$\begin{array}{l}(P\ Y) = \dot{\neg}\dot{\forall}G_1\\ (G_1\ W_1) = \dot{\neg}(R\ g(U_1)\ U_1)\dot{\vee}\dot{\neg}\dot{\forall}G_2\\ (G_2\ y_1) = \dot{\neg}A_2\dot{\vee}\dot{\neg}B_2\\ (G_1\ W_1') = \dot{\neg}A_2\dot{\vee}\dot{\neg}B_2\\ (P\ Y) = \dot{\neg}\dot{\forall}G_3\\ (G_3\ y_2) = \dot{\neg}(R\ g(Y')\ Z')\dot{\vee}\dot{\neg}B_3\\ (P\ Z') = \dot{\neg}B_3\\ (P\ Y') = \dot{\neg}\dot{\forall}G_4\\ (G_4\ W_2) = \dot{\neg}(R\ g(U_2)\ U_2)\dot{\vee}\dot{\neg}\dot{\forall}G_5\\ (G_5\ y_3) = \dot{\neg}A_5\dot{\vee}\dot{\neg}B_5\\ (G_4\ W_2') = \dot{\neg}A_5\dot{\vee}\dot{\neg}B_5\end{array}$$

1		$\varepsilon(R\ g(U)\ U)$
2		$\neg\varepsilon(R\ X\ Y), \neg\varepsilon(R\ X\ Z), \neg\varepsilon(P\ Y), \varepsilon(P\ Z)$
3		$\neg\varepsilon(R\ X\ Y), \neg\varepsilon(R\ X\ Z), \neg\varepsilon(P\ Z), \varepsilon(P\ Y)$
4	res. $(1,2)$	$\neg\varepsilon(R\ g(Y)\ Z), \neg\varepsilon(P\ Y), \varepsilon(P\ Z)$
5	res. $(1,4)$	$\neg\varepsilon(P\ Y), \varepsilon(P\ Y)$
6	five narr. (5)	$\neg\varepsilon(A_1), \neg\varepsilon(B_1), \varepsilon(P\ Y)/c_1, c_2$
7	res. $(6,1)$	$\neg\varepsilon(B_1), \varepsilon(P\ Y)/c_1, c_2'$
8	four narr. (7)	$\varepsilon(A_2), \varepsilon(P\ Y)/c_1, c_2'', c_3$
9		$\varepsilon(B_2), \varepsilon(P\ Y)/c_1, c_2'', c_3$
10	renaming (6)	$\neg\varepsilon(A_1'), \neg\varepsilon(B_1'), \varepsilon(P\ Y)/c_1, c_4$
11	res. $(10,8)$	$\neg\varepsilon(B_1'), \varepsilon(P\ Y)/c_1, c_2'', c_3, c_4'$
12	res. $(11,9)$	$\varepsilon(P\ Y)/c_1, c_2'', c_3, c_4''$
13	five narr. (12)	$\varepsilon(A_3)/c_1, c_2'', c_3, c_4'', c_5, c_6$
14		$\varepsilon(B_3)/c_1, c_2'', c_3, c_4'', c_5, c_6$
15	renaming (4)	$\neg\varepsilon(R\ g(Y')\ Z'), \neg\varepsilon(P\ Y'), \varepsilon(P\ Z')$
16	res. $(15,13)$	$\neg\varepsilon(P\ Y'), \varepsilon(P\ Z')/c_1, c_2'', c_3, c_4'', c_5, c_6'$
17	narr. (16)	$\neg\varepsilon(P\ Y'), \neg\varepsilon(Q)/c_1, c_2'', c_3, c_4'', c_5, c_6', c_7$
18	res. $(17,14)$	$\neg\varepsilon(P\ Y')/c_1, c_2'', c_3, c_4'', c_5, c_6', c_7'$
19	five narr. (18)	$\neg\varepsilon(A_4), \neg\varepsilon(B_4)/c_1, c_2'', c_3, c_4'', c_5, c_6', c_7', c_8, c_9$
20	res. $(19,1)$	$\neg\varepsilon(B_4)/c_1, c_2'', c_3, c_4'', c_5, c_6', c_7', c_8, c_9'$
21	four narr. (20)	$\varepsilon(A_5)/c_1, c_2'', c_3, c_4'', c_5, c_6', c_7', c_8, c_9'', c_{10}$
22		$\varepsilon(B_5)/c_1, c_2'', c_3, c_4'', c_5, c_6', c_7', c_8, c_9'', c_{10}$
23	renaming (19)	$\neg\varepsilon(A_4'), \neg\varepsilon(B_4')/c_1, c_2'', c_3, c_4'', c_5, c_6', c_7', c_8, c_{11}$
24	res. $(23,21)$	$\neg\varepsilon(B_4')/c_1, c_2'', c_3, c_4'', c_5, c_6', c_7', c_8, c_9'', c_{10}, c_{11}'$
25	res. $(24,22)$	$\square/c_1, c_2'', c_3, c_4'', c_5, c_6', c_7', c_8, c_9'', c_{10}, c_{11}''$

with

$$
\begin{array}{ll}
c_1 & (P\ Y) = \dot{\neg}\dot{\forall} G_1 \\
c_2 & (G_1\ W_1) = \dot{\neg} A_1 \dot{\vee} \dot{\neg} B_1 \\
c_2' & (G_1\ W_1) = \dot{\neg}(R\ g(U_1)\ U_1) \dot{\vee} \dot{\neg} B_1 \\
c_2'' & (G_1\ W_1) = \dot{\neg}(R\ g(U_1)\ U_1) \dot{\vee} \dot{\neg}\dot{\forall} G_2 \\
c_3 & (G_2\ y_1) = \dot{\neg} A_2 \dot{\vee} \dot{\neg} B_2 \\
c_4 & (G_1\ W_1') = \dot{\neg} A_1' \dot{\vee} \dot{\neg} B_1' \\
c_4' & (G_1\ W_1') = \dot{\neg} A_2 \dot{\vee} \dot{\neg} B_1' \\
c_4'' & (G_1\ W_1') = \dot{\neg} A_2 \dot{\vee} \dot{\neg} B_2 \\
c_5 & (P\ Y) = \dot{\neg}\dot{\forall} G_3 \\
c_6 & (G_3\ y_2) = \dot{\neg} A_3 \dot{\vee} \dot{\neg} B_3 \\
c_6' & (G_3\ y_2) = \dot{\neg}(R\ g(Y')\ Z') \dot{\vee} \dot{\neg} B_3 \\
c_7 & (P\ Z') = \dot{\neg} Q \\
c_7' & (P\ Z') = \dot{\neg} B_3 \\
c_8 & (P\ Y') = \dot{\neg}\dot{\forall} G_4 \\
c_9 & (G_4\ W_2) = \dot{\neg} A_4 \dot{\vee} \dot{\neg} B_4 \\
c_9' & (G_4\ W_2) = \dot{\neg}(R\ g(U_2)\ U_2) \dot{\vee} \dot{\neg} B_4 \\
c_9'' & (G_4\ W_2) = \dot{\neg}(R\ g(U_2)\ U_2) \dot{\vee} \dot{\neg}\dot{\forall} G_5 \\
c_{10} & (G_5\ y_3) = \dot{\neg} A_5 \dot{\vee} \dot{\neg} B_5 \\
c_{11} & (G_4\ W_2') = \dot{\neg} A_4' \dot{\vee} \dot{\neg} B_4' \\
c_{11}' & (G_4\ W_2') = \dot{\neg} A_5 \dot{\vee} \dot{\neg} B_4' \\
c_{11}'' & (G_4\ W_2') = \dot{\neg} A_5 \dot{\vee} \dot{\neg} B_5
\end{array}
$$

Fig. 6. Cantor's theorem in type theory with a relation

that have the solution

$P = \lambda\dot{\neg}[\uparrow](1\ (g(C)[\uparrow]))$
$G_1 = G_2 = G_3 = G_4 = G_5 = \lambda(\dot{\neg}[\uparrow](R[\uparrow]\ g(C)[\uparrow]\ 1)\dot{\vee}[\uparrow]\dot{\neg}[\uparrow](1\ g(C)[\uparrow]))$
$Y = W_1 = U_1 = Y' = W_2 = U_2 = C$
$W_1' = y_1$
$Z' = y_2$
$W_2' = y_3$
$A_2 = (R\ g(C)\ y_1)$
$B_2 = (y_1\ g(C))$
$B_3 = (y_2\ g(C))$
$A_5 = (R\ g(C)\ y_3)$
$B_5 = (y_3\ g(C))$

where $C = \lambda x\ \dot{\forall}\lambda y\ (\dot{\neg}(R\ x\ y)\dot{\vee}\dot{\neg}(y\ x))$,
i.e. $\lambda\ \dot{\forall}[\uparrow]\lambda\ (\dot{\neg}[\uparrow^2](R[\uparrow^2]\ 2\ 1)\dot{\vee}[\uparrow^2]\dot{\neg}[\uparrow^2](1\ 2))$.

6.3 In set theory

We consider a set B and a potential surjection from this set to its power set. We express this potential surjection by a set R. The surjectivity and functionality of this set are expressed by the propositions

$$E : \forall y\ (y \in \mathcal{P}(B) \Rightarrow \exists x\ (x \in B \wedge < x, y > \in R))$$

$$F : \forall x\ \forall y\ \forall z\ (< x, y > \in R \Rightarrow < x, z > \in R \Rightarrow y = z)$$

We use also the axiom of equality

$$L : \forall z\ \forall x\ \forall y\ (x = y \Rightarrow \neg z \in x \Rightarrow \neg z \in y)$$

The proposition E reduces to the proposition

$$\forall u\ (\forall y\ (y \in u \Rightarrow y \in B)) \Rightarrow \exists x\ (x \in B \wedge < x, u > \in R))$$

Putting this proposition in clausal form yields the clauses

$$y(U) \in U, < g(U), U > \in R$$

$$\neg y(U) \in B, < g(U), U > \in R$$

$$y(U) \in U, g(U) \in B$$

$$\neg y(U) \in B, g(U) \in B$$

The two other propositions yield the clauses

$$\neg < X, Y > \in R, \neg < X, Z > \in R, Y = Z$$

$$\neg X = Y, Z \in X, \neg Z \in Y$$

The search is described on figure 7 where we simplify the constraints and substitute the solved constraints at each step. Propagating the solved constraints may lead to new reductions that require to put the proposition in clausal form again. This explains that some resolution steps yield several clauses. It returns the empty clause.

1	$y(U) \in U, < g(U), U >\in R$
2	$\neg y(U) \in B, < g(U), U >\in R$
3	$y(U) \in U, g(U) \in B$
4	$\neg y(U) \in B, g(U) \in B$
5	$\neg < X, Y >\in R, \neg < X, Z >\in R, Y = Z$
6	$\neg X = Y, Z \in X, \neg Z \in Y$
7 narr. (1)	$y(C) \in B, < g(C), C >\in R$
8	$\neg < y(C), W >\in R, \neg y(C) \in W, < g(C), C >\in R$
9 res. (7, 2)	$< g(C), C >\in R$
10 narr. (3)	$y(C) \in B, g(C) \in B$
11	$\neg < y(C), W >\in R, \neg y(C) \in W, g(C) \in B$
12 res. (10, 4)	$g(C) \in B$
13 res. (9, 5)	$\neg < g(C), Z >\in R, C = Z$
14 res. (9, 13)	$C = C$
15 res.(14, 6)	$Z \in B, \neg Z \in B, < Z, y_1(Z) >\in R$
16	$Z \in B, \neg Z \in B, Z \in y_1(Z)$
17	$\neg < Z, W_1 >\in R, \neg Z \in W_1, \neg Z \in B, < Z, y_1(Z) >\in R$
18	$\neg < Z, W_1 >\in R, \neg Z \in W_1, \neg Z \in B, Z \in y_1(Z)$
19 res. (17, 9)	$\neg g(C) \in B, < g(C), y_2 >\in R, < g(C), y_1(g(C)) >\in R$
20	$\neg g(C) \in B, g(C) \in y_2, < g(C), y_1(g(C)) >\in R$
21 res. (19, 12)	$< g(C), y_2 >\in R, < g(C), y_1(g(C)) >\in R$
22 res. (20, 12)	$g(C) \in y_2, < g(C), y_1(g(C)) >\in R$
23 res. (18, 9)	$\neg g(C) \in B, < g(C), y_3 >\in R, g(C) \in y_1(g(C))$
24	$\neg g(C) \in B, g(C) \in y_3, g(C) \in y_1(g(C))$
25 res. (23, 12)	$< g(C), y_3 >\in R, g(C) \in y_1(g(C))$
26 res. (24, 12)	$g(C) \in y_3, g(C) \in y_1(g(C))$
27 res. (17, 21)	$\neg g(C) \in y_2, \neg g(C) \in B, < g(C), y_1(g(C)) >\in R$
28 res. (27, 12)	$\neg g(C) \in y_2, < g(C), y_1(g(C)) >\in R$
29 res. (28, 22)	$< g(C), y_1(g(C)) >\in R$
30 res. (18, 25)	$\neg g(C) \in y_3, \neg g(C) \in B, g(C) \in y_1(g(C))$
31 res. (30, 12)	$\neg g(C) \in y_3, g(C) \in y_1(g(C))$
32 res. (31, 26)	$g(C) \in y_1(g(C))$
33 res. (29, 13)	$C = y_1(g(C))$
34 res. (33, 6)	$Z \in B, \neg Z \in y_1(g(C))$
35	$\neg < Z, W_2 >\in R, \neg Z \in W_2, \neg Z \in y_1(g(C))$
36 res. (35, 32)	$\neg < g(C), W_2 >\in R, \neg g(C) \in W_2$
37 res. (36, 9)	$\neg g(C) \in B, < g(C), y_4 >\in R$
38	$\neg g(C) \in B, g(C) \in y_4$
39 res. (37, 12)	$< g(C), y_4 >\in R$
40 res. (38, 12)	$g(C) \in y_4$
41 res. (36, 39)	$\neg g(C) \in y_4$
42 res. (41, 40)	$\square$

Fig. 7. Cantor's theorem in set theory

6.4 Remarks

The termination and completeness issues are not addressed by these examples because, even in set theory, Cantor's theorem has a cut free proof and the search involves only terminating propositions.

The proof in set theory is longer because several steps are dedicated to the treatment of typing literals that are repeatedly resolved with the clause (12).

In type theory with a function, only two extended narrowing steps are needed to generate the symbol $\dot{\neg}$ in the term $\lambda\ \dot{\neg}[\uparrow](f[\uparrow]\ 1\ 1)$ (i.e. $\lambda x\ \dot{\neg}(f\ x\ x)$) that expresses Cantor's set. In type theory with a relation, four extended narrowing steps are needed to generate the term $\lambda\ \dot{\forall}[\uparrow]\lambda\ (\dot{\neg}[\uparrow^2](R[\uparrow^2]\ 2\ 1)\dot{\vee}[\uparrow^2]\dot{\neg}[\uparrow^2](1\ 2))$ (i.e. $\lambda x\ \dot{\forall}\lambda y\ (\dot{\neg}(R\ x\ y)\dot{\vee}\dot{\neg}(y\ x))$) that expresses Cantor's set. The term expressing Cantor set is thus mostly constructed by the unification algorithm in the first case and mostly constructed by the extended narrowing rule in the second. In set theory, like in type theory with a relation, the term expressing Cantor's set is mostly constructed by the extended narrowing rule.

In this case, a single step is needed because we have taken the *ad hoc* rule

$$x \in C \longrightarrow x \in B \wedge \forall y\ (< x, y > \in R \Rightarrow \neg x \in y)$$

But in a reasonable formulation of set theory several steps would be needed.

Notice, at last, that in the proof in type theory with a relation, the term expressing Cantor's set is constructed several times, because the constraints are frozen while in set theory, because the constraints are solved on the fly, this term is constructed only twice and propagated. To avoid this redundancy in type theory with a relation, it would be a good idea to solve as soon as possible the constraints c_1 and c_2.

Conclusion

Using a single automated theorem proving method for type theory and for set theory permits a comparison.

Although the use of a typed (many-sorted) language can be criticized, type theory has several advantages for automated theorem proving: typing permits to avoid typing literals, it enjoys termination and cut elimination, and the possibility to form a term $(f\ a)$ expressing the image of an object by a function avoids indirect definitions.

This motivates the search of a type-free formalization of mathematics, that also enjoys termination and cut elimination and where functions are primitive.

References

1. P.B. Andrews. Resolution in type theory. *The Journal of Symbolic Logic*, 36, 3 (1971), pp. 414-432.

2. P.B. Andrews, An introduction to mathematical logic and type theory: to truth through proof, *Academic Press* (1986).
3. P.B. Andrews, D.A. Miller, E. Longini Cohen, and F. Pfenning, Automating higher-order logic, W.W. Bledsoe and D.W. Loveland (Eds.), *Automated theorem proving: after 25 years*, Contemporary Mathematics Series 29, American Mathematical Society (1984), pp. 169-192.
4. S.C. Bailin, A normalization theorem for set theory, *The Journal of Symbolic Logic*, 53, 3 (1988), pp. 673-695.
5. S.C. Bailin, A λ-unifiability test for set theory, *Journal of Automated Reasoning*, 4 (1988), pp. 269-286.
6. A. Church, A formulation of the simple theory of types, *The Journal of Symbolic Logic*, 5 (1940), pp. 56-68.
7. M. Davis, Invited commentary to [28], A.J.H. Morrell (Ed.) *Proceedings of the International Federation for Information Processing Congress*, 1968, North Holland (1969) pp. 67-68.
8. D.J. Dougherty, Higher-order unification via combinators, *Theoretical Computer Science*, 114 (1993), pp. 273-298.
9. G. Dowek, Lambda-calculus, combinators and the comprehension scheme, M. Dezani-Ciancaglini and G. Plotkin (Eds.), *Typed Lambda Calculi and Applications*, Lecture notes in computer science 902, Springer-Verlag (1995), pp. 154-170. *Rapport de Recherche* 2565, INRIA (1995).
10. G. Dowek, Proof normalization for a first-order formulation of higher-order logic, E.L. Gunter and A. Felty (Eds.), *Theorem Proving in Higher-order Logics*, Lecture notes in computer science 1275, Springer-Verlag (1997), pp. 105-119. *Rapport de Recherche* 3383, INRIA (1998).
11. G. Dowek, Th. Hardin, and C. Kirchner, Theorem proving modulo, *Rapport de Recherche* 3400, INRIA (1998).
12. G. Dowek, Th. Hardin, and C. Kirchner, HOL-$\lambda\sigma$: an intentional first-order expression of higher-order logic, to appear in *Rewriting Techniques and Applications* (1999). *Rapport de Recherche* 3556, INRIA (1998).
13. G. Dowek and B. Werner, Proof normalization modulo, *Rapport de Recherche* 3542, INRIA (1998).
14. J. Ekman, Normal proofs in set theory, *Doctoral thesis*, Chalmers University of Technology and University of Göteborg (1994).
15. M. Fay, First-order unification in an equational theory, *Fourth Workshop on Automated Deduction* (1979), pp. 161-167.
16. L. Hallnäs, On normalization of proofs in set theory, *Doctoral thesis*, University of Stockholm (1983).
17. G. Huet, Constrained resolution: a complete method for higher order logic, *Ph.D.*, Case Western Reserve University (1972).
18. G. Huet, A mechanization of type theory, *International Joint Conference on Artificial Intelligence* (1973), pp. 139-146.
19. G. Huet, A unification algorithm for typed lambda calculus, *Theoretical Computer Science*, 1,1 (1975), pp. 27–57.
20. G. Huet, Résolution d'équations dans les Langages d'Ordre 1,2, ..., ω, *Thèse d'État*, Université de Paris VII (1976).
21. J.-M. Hullot, Canonical forms and unification, W. Bibel and R. Kowalski (Eds.) *Conference on Automated Deduction*, Lecture Notes in Computer Science 87, Springer-Verlag (1980), pp. 318-334.

22. J.-P. Jouannaud and C. Kirchner, Solving equations in abstract algebras: a rule-based survey of unification, J.-L. Lassez and G. Plotkin (Eds.) *Computational logic. Essays in honor of Alan Robinson*, MIT press (1991), pp. 257–321.
23. J.W. Klop, V. van Oostrom, and F. van Raamsdonk, Combinatory reduction systems: introduction and survey, *Theoretical Computer Science*, 121 (1993), pp. 279-308.
24. D.A. Miller, Proofs in higher order logic, *Ph.D.*, Carnegie Mellon University (1983).
25. D.A. Miller, A compact representation of proofs, *Studia Logica*, 46, 4 (1987).
26. G. Plotkin, Building-in equational theories, *Machine Intelligence*, 7 (1972), pp. 73-90.
27. W.V.O. Quine, Set theory and its logic, *Belknap press* (1969).
28. J.A. Robinson. New directions in mechanical theorem proving. A.J.H. Morrell (Ed.) *Proceedings of the International Federation for Information Processing Congress*, 1968, North Holland (1969), pp. 63-67.
29. J.A. Robinson. A note on mechanizing higher order logic. *Machine Intelligence* 5, Edinburgh university press (1970), pp. 123-133.
30. M. Stickel, Automated deduction by theory resolution, *Journal of Automated Reasoning*, 4, 1 (1985), pp. 285-289.