Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 174 (2007) 127–139
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Encoding Functional Relations in Scunak

Chad E. Brown[1]

Universität des Saarlandes
Saarbrücken, Germany

Abstract

We describe how a set-theoretic foundation for mathematics can be encoded in the new system Scunak. We then discuss an encoding of the construction of functions as functional relations in untyped set theory. Using the dependent type theory of Scunak, we can define object level application and lambda abstraction operators (in the spirit of higher-order abstract syntax) mediating between functions in the (meta-level) type theory and (object-level) functional relations. The encoding has also been exported to Automath and Twelf.

Keywords: Set Theory, Dependent Type Theory, Proof Irrelevance, Formal Mathematics

1 Introduction

Untyped set theory is often considered a foundation for mathematics because most of the usual mathematical objects of interest can be constructed as sets. For instance, certain sets can be considered pairs, and certain sets of pairs can be considered functions. In textbooks, this construction is described informally, as carrying out such a construction in standard first-order formulations of set theory is tedious. In this paper, we will describe how such a construction can be carried out in a natural, but fully formal, manner by encoding the construction in a dependent type theory. (Of course, such constructions have been formalized before in other systems [7,3,6].)

The construction can be carried out using the type theories implemented in Twelf [8] or Automath [9]. However, we will show how the encoding becomes easier and arguably more natural using the system Scunak. We then export the signature to Twelf and Automath.

There are essentially two reasons why the encoding is natural in Scunak. First, Scunak includes "class types." Second, the concrete syntax for types and terms includes some syntactic sugar for set theory notation.

[1] Email: cebrown@ags.uni-sb.de

1571-0661
doi:10.1016/j.entcs.2007.01.022

Type-theoretically, class types are particular instances of Σ-types for pairs of the form $\langle x, p\rangle$ where x is an object and p is a proof of a property of the object. Scunak also includes proof irrelevance, so that the Σ-types behave in some ways as subset types rather than types of pairs. The reason for calling these "class" types is set theoretic. Assuming all mathematical objects are sets (a common assumption in axiomatic set theory), predicates correspond to classes. For each predicate ϕ, the class type for ϕ in Scunak is essentially

$$\{\langle x, p\rangle | x \text{ is an object and } p \text{ is a proof of } \phi(x)\}$$

This set corresponds to the class $\{x|\phi(x)\}$ if there is at most one proof of $\phi(x)$ for each x (i.e., if one has proof irrelevance). Without proof irrelevance, such Σ-types do not correspond to classes since elements in the class may have several representatives in the Σ-type. While class types play an important role in the construction described in this paper, proof irrelevance can be avoided. Consequently, we will for the most part avoid discussing proof irrelevance.

We refer to three systems throughout the paper: Scunak, Twelf, and Automath. Each of these refers to an implemented system which includes, at least, a type checker for some type theory. Twelf [8] includes a checker for the LF type theory [5] (as well as various other important features). Simply referring to "Automath" is ambiguous, since there have been a number of type theories in the Automath family which have been implemented more than once [4]. When we refer to "Automath" as a type theory, we are referring to AUT-68. When we refer to "Automath" as a system, we are referring to Freek Wiedijk's C implementation of a checker for the AUT-68 and AUT-QE type theories [9].

The new system we discuss in this paper is Scunak [2,1]. Scunak includes a type checker for what we will call the "Scunak type theory." Within this type theory, one can specify foundations for mathematics by giving a signature. We will demonstrate this in the paper by describing an axiomatic set theory and a construction of functions as functional relations. Of course, Scunak includes a concrete syntax (the PAM syntax) for specifying types and terms. The PAM syntax provides syntactic sugar for set theoretic constructions. For instance, notation such as `{x:A|x::B}` can be used where $\{x \in A | x \in B\}$ is intended. The parser expands this into a term in the Scunak type theory.

2 Syntax

We begin by briefly describing the Scunak type theory. We use $x, y, z, x^1, \ldots$ to denote variables and $c, d, c^1, \ldots$ to denote constants. For terms, we take untyped λ-terms with constants and pairing. The basic types are as follows:

- `obj` is the type of all mathematical objects. In set theory, objects are sets.
- `prop` is the type of all propositions.
- `pf` P is the type of all proofs of the proposition P.
- `class` ϕ, where ϕ is a property, is the type of pairs $\langle M, N\rangle$ where M is an object

and N is a proof that M satisfies the property ϕ.

For types, we take the dependent types generated starting from these basic types. In other words, we have:

$$\begin{array}{lll}\textbf{Terms} & M, N, P, \phi, \ldots & := x|c|(\lambda x.M)|(M\,N)|\langle M, N\rangle|\pi_1(M)|\pi_2(M) \\ \textbf{Types} & A, B, C, \ldots & := \texttt{obj}|\texttt{prop}|(\texttt{pf}\ P)|(\texttt{class}\ \phi)|(\Pi x : A.B)\end{array}$$

We use $A \to B$ to denote $\Pi x : A.B$ when x does not occur free in B.

We use $[M/x]$ to denote substitution of M for x. We assume familiarity with β-reduction and the following pairing reductions:

$$(\pi_1) : \pi_1(\langle M, N\rangle) \to_{\pi_1} M \qquad (\pi_2) : \pi_2(\langle M, N\rangle) \to_{\pi_2} N$$

When type-checking, we restrict to $\beta\pi_1\pi_2$-normal terms. If such a normal term M is neither of the form $(\lambda x.M_1)$ nor $\langle M_1, M_2\rangle$, we say M is an *extraction*. We use E, F, E^1, E^2, ... to denote extractions. We could optionally include η-reduction and a surjective pairing reduction reducing $\langle \pi_1(M), \pi_2(M)\rangle$ to M, but these are not needed for type checking the signature considered in this paper. (One can enable or disable such reductions in Scunak using flags.)

As usual, Σ denotes a signature $c^1 : A^1, \ldots, c^n : A^n$. Similarly, Γ denotes a context $x^1 : B^1, \ldots, x^m : B^m$. We assume (but do not discuss) validity of signatures and contexts.

In order to account for proof irrelevance, the main judgments in the Scunak type theory are $\Gamma \vdash M \sim N \uparrow A$ (checking normal terms M and N are equal at type A) and $\Gamma \vdash E \sim F \downarrow A$ (extracting a type A at which extractions E and F are equal). Rules for such judgments are given in [1]. Since we will not need proof irrelevance in this paper, we can give a simplified typing judgment and rely on structure equality of normal forms of terms. We let $M^{\downarrow}$ and $A^{\downarrow}$ denote the normal form of types and terms, respectively. Since terms are untyped, normal forms do not always exist. In the cases we consider in this paper, normal forms exist. The type judgments we consider here are the following:

- $\Gamma \vdash_\Sigma M \uparrow A$ (Check normal term M has type A.)
- $\Gamma \vdash_\Sigma E \downarrow A$ (Extract type A for extraction E.)
- $\Gamma \vdash_\Sigma A : \mathit{Type}$ (Check A is a valid type.)

The corresponding rules are given in Figures 1 and 2.

It is important to note that this is a simplification of the actual type-checking performed in Scunak. The term $(\lambda P \lambda\phi\lambda u\lambda v\lambda w.w)$ can be checked to inhabit type $(\Pi P : \texttt{prop}.\Pi\phi : (\texttt{pf}\ P \to \texttt{prop}).\Pi u : (\texttt{pf}\ P).\Pi v : (\texttt{pf}\ P).\Pi w : (\texttt{pf}\ (\phi\, u)).\texttt{pf}\ (\phi\, v))$ by making use of proof irrelevance. (In particular, the proofs u and v can be considered the same.) However, the term does not inhabit the type using the simplified form of typing presented here. While semantically proof irrelevance is vital for class types to correspond to classes, in the Scunak signatures considered so far, proof irrelevance has rarely actually been used during type checking. Even when proof irrelevance is used, its use can often be eliminated fairly easily. In the first construction of functions from sets in Scunak, proof irrelevance

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \downarrow A} \qquad \frac{c : A \in \Sigma}{\Gamma \vdash c \downarrow A} \qquad \frac{\Gamma \vdash E \downarrow (\Pi x : A.B) \quad \Gamma \vdash M \uparrow A}{\Gamma \vdash (EM) \downarrow ([M/x]B)}$$

$$\frac{\Gamma \vdash E \downarrow \texttt{class}\ \ \phi}{\Gamma \vdash \pi_2(E) \downarrow \texttt{pf}\ \ (\phi\, \pi_1(E))} \qquad \frac{\Gamma \vdash E \downarrow \texttt{class}\ \ \phi}{\Gamma \vdash \pi_1(E) \downarrow \texttt{obj}}$$

$$\frac{\Gamma \vdash E \downarrow B \quad B \in \{\texttt{obj}, \texttt{prop}\}}{\Gamma \vdash E \uparrow B} \qquad \frac{\Gamma \vdash E \downarrow \texttt{pf}\ \ M \quad M^{\downarrow} = N}{\Gamma \vdash E \uparrow \texttt{pf}\ \ N}$$

$$\frac{\Gamma, z : A \vdash [z/u]M \uparrow [z/x]B \quad z \in \mathcal{V} \text{ fresh}}{\Gamma \vdash (\lambda u M) \uparrow (\Pi x : A.B)} \qquad \frac{\Gamma, z : A \vdash (Ez) \uparrow [z/x]B \quad z \in \mathcal{V} \text{ fresh}}{\Gamma \vdash E \uparrow (\Pi x : A.B)}$$

$$\frac{\Gamma \vdash_\Sigma M_1 \uparrow \texttt{obj} \quad \Gamma \vdash_\Sigma M_2 \uparrow \texttt{pf}\ \ (\phi\, M_1)}{\Gamma \vdash_\Sigma \langle M_1, M_2 \rangle \uparrow \texttt{class}\ \ \phi}$$

$$\frac{\Gamma \vdash_\Sigma \pi_1(E) \uparrow \texttt{obj} \quad \Gamma \vdash_\Sigma \pi_2(E) \uparrow \texttt{pf}\ \ (\phi\, \pi_1(E))}{\Gamma \vdash_\Sigma E \uparrow \texttt{class}\ \ \phi}$$

Fig. 1. Rules for Typing Judgments without Proof Irrelevance

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma, z : A \vdash [z/x]B : \mathit{Type} \quad z \in \mathcal{V} \text{ fresh}}{\Gamma \vdash (\Pi x : A.B) : \mathit{Type}} \qquad \frac{}{\Gamma \vdash \texttt{obj} : \mathit{Type}}$$

$$\frac{}{\Gamma \vdash \texttt{prop} : \mathit{Type}} \qquad \frac{\Gamma \vdash M \uparrow \texttt{prop}}{\Gamma \vdash \texttt{pf}\ \ M : \mathit{Type}} \qquad \frac{\Gamma \vdash M \uparrow (\texttt{obj} \to \texttt{prop})}{\Gamma \vdash \texttt{class}\ \ M : \mathit{Type}}$$

Fig. 2. Simplified Rules for Valid Types

was used a few times, but these occurrences were eliminated by slightly modifying a few declarations.

Naturally, there are several important meta-theoretic questions one could investigate regarding the Scunak type theory. Is it possible that a type is inhabited by a nonnormal term, but inhabited by no normal term? The answer to this question is trivially "no", since only normal terms can be judged to inhabit a type given the algorithmic typing rules in Figure 2. Meta-theoretic issues such as normalization and subject reduction become interesting once one considers a typing judgment for arbitrary terms. One can then consider whether the algorithmic typing judgment is complete with respect to the more general judgment. One can also consider semantics for types and terms. We leave such issues for future work. At the moment the emphasis of the research is on investigating the naturality of encoding formal mathematics in the Scunak type theory.

A Scunak signature can be translated into a Twelf or Automath signature. In both Twelf and Automath, one begins by declaring three basic type families corresponding to `obj`, `prop` and `pf`. When translating to Twelf or Automath, any occurrences of class types are removed by Currying. So long as the Scunak signature can be type-checked using the simplified typing system above (i.e., proof irrelevance is not needed), the resulting Twelf and Automath files should be well-typed. In the signature described below, we have managed to remove all essential uses of proof irrelevance so that the corresponding Twelf and Automath files do type check. (Actually, one must explicitly add `%abbrev` to some Twelf abbreviations by hand, but this is a separate issue.)

3 Specifying a Set Theory

One can give a signature of constants and abbreviations for Scunak in PAM files. The PAM syntax allows a mixture of set theoretic and type theoretic notations. (PAM stands for "pseudo-Automath" since some of the notation is similar to Automath. However, the PAM syntax is also significantly different from Automath.) To demonstrate the PAM syntax, we describe a PAM file for a form of set theory starting from certain axioms and ending with a definition of functions as functional relations. We begin by describing the constants in the signature which correspond to the axiomatic kernel of the set theory. Similar encodings of a variety of foundational systems for mathematics in Automath are discussed in [10].

Throughout a PAM file, one can specify local parameters. For example,

```
[M:prop][N:prop][y:obj][z:obj][A:set][B:set][C:set]
```

Intuitively, this declaration of parameters means: "Let M and N be propositions, y and z be objects, and A, B and C be sets." (Note that `obj` and `set` are synonyms, standing for the same basic type `obj`.)

The declaration

```
(not M):prop.
```

introduces a new constant `not` of type `prop` $\to$ `prop` into the signature. (Note the use of the parameter M of type `prop` as an argument.)

In order to obtain classical logic, we can declare an excluded middle proof by cases rule as follows:

```
[case1:|- M -> |- N]
[case2:|- (not M) -> |- N]
(xmcases M N case1 case2):|- N.
```

The parameters `case1` and `case2` correspond to the two premises of the rule. Note that `|- N` is the PAM syntax for the type (`pf` N). The type of `xmcases` is

$$\Pi M : \texttt{prop}.\Pi N : \texttt{prop}.(\texttt{pf}\ \ M \to \texttt{pf}\ \ N) \to (\texttt{pf}\ \ (\texttt{not}\, M) \to \texttt{pf}\ \ N) \to \texttt{pf}\ \ N$$

We also declare the usual elimination rule for negation.

```
(notE M N):|- M -> |- (not M) -> |- N.
```

The usual introduction rule for negation, as well as the proof by contradiction rule, can be derived using `xmcases` and `notE`.

Negation and the two rules above translate into the following Twelf code

```
not : prop -> prop.
```

```
xmcases : {M:prop} {N:prop} (pf M -> pf N)
                    -> (pf (not M) -> pf N) -> pf N.
notE : {M:prop} {N:prop} pf M -> pf (not M) -> pf N.
```

as well as corresponding Automath code. Since class types have not yet been used, the Scunak, Twelf, and Automath versions are very similar.

One may expect to see more propositional connectives (such as conjunction or implication) in the signature. However, once we include the set theory constructors and axioms, we can actually define these connectives. We will show such definitions later.

The basic relations in set theory are equality and membership.

```
(eq y z):prop.
(in A z):prop.
```

In PAM syntax, one can write `(y==z)` for `(eq y z)` and `(z::A)` for `(in A z)`. Note that `(in A z)` intuitively represents the proposition $z \in A$. The reason the arguments are reversed is so that the η-short form `(in A)`, an extraction of type `obj` $\to$ `prop`, represents the "class" of all members of `A`.

An equality elimination rule corresponding to replacing equals by equals is included in the signature. We omit this here.

The rule for set extensionality is declared as follows.

```
[AsubB:{x:obj}{u:|- (x::A)}|- (x::B)]
[BsubA:{x:obj}{u:|- (x::B)}|- (x::A)]
(setext A B AsubB BsubA):|- (A==B).
```

The type of the parameter `BsubA` is $\Pi x : \mathtt{obj}.\Pi u : \mathtt{pf}\ (\mathtt{in}\ B\, x).\mathtt{pf}\ (\mathtt{in}\ A\, x)$. Intuitively, this corresponds to a premise stating that every element of B is an element of A. That is, B is a subset of A. However, note that this represents the assertion that B is a subset of A at the *type* level, not at the level of propositions. We will reuse the parameter `BsubA` when declaring the rules for powerset.

At this point, we can begin describing the basic set constructors and the rules (or axioms) corresponding to each such constructor.

There is an empty set. We encode this axiom simply by declaring a constant `emptyset` of type `obj`.

```
emptyset:obj.
```

In PAM syntax, one can write `{}` for `emptyset`. If some y is in the empty set, then every proposition M holds.

```
[yinempty:|- (y::{})]
(emptysetE y yinempty M):|- M.
```

We can adjoin y to the set A to obtain the set $y; A$ (or, $\{y\} \cup A$).

```
(setadjoin y A):obj.
```

In PAM syntax, `(y;A)` represents `(setadjoin y A)`. There is special PAM syntax for finite enumerated sets which expands into `emptyset` and `setadjoin`. One can use `{x1,...,xn}` (intuitively, the finite set $\{x_1, \ldots, x_n\}$) to represent the term `(setadjoin x1 ... (setadjoin xn emptyset))`. In particular, `{y}` and `{y,z}` correspond to the terms `(setadjoin y emptyset)` and `(setadjoin y (setadjoin z emptyset))`, respectively. We omit the three rules for introducing and eliminating `setadjoin`.

The power set of A is a set. There are two rules for introducing and eliminating the powerset. (Note the reuse of the parameter `BsubA` declared above.)

```
(powerset A):obj.
(powersetI A B BsubA):|- (B::(powerset A)).
(powersetE A B z):|- (B::(powerset A)) -> |- (z::B) -> |- (z::A).
```

The union of A (intuitively, $\bigcup A$) is a set. There are two corresponding rules, omitted here.

```
(setunion A):obj.
```

Finally, we come to the most interesting axiom: separation. We can state this as follows. For any property $\psi(x)$ of elements $x \in A$, there is a set $\{x \in A|\psi(x)\}$.

```
[psi:A -> prop]
(dsetconstr A psi):obj.
```

We have declared the parameter `psi` to have type `A -> prop`. However, technically, `A` is a term, not a type. In PAM syntax one is allowed to use an extraction as a type, so long as the extraction has type `obj` or `obj` $\to$ `prop`. In this case, `A` has type `obj`. So, Scunak assumes the intention is for `A` to be the class type `class` $(\mathtt{in}\,A)$.[2] Technically, the type of `psi` is $(\mathtt{class}\ (\mathtt{in}\,A)) \to \mathtt{prop}$ and the type of `dsetconstr` is $\Pi A : \mathtt{obj}.((\mathtt{class}\ (\mathtt{in}\,A)) \to \mathtt{prop}) \to \mathtt{obj}$.

In PAM syntax, we write `{x:A|M}` for `(dsetconstr A (\x.M))`, where a backslash is PAM syntax for a λ binder.

Note that `dsetconstr` makes explicit use of a class type. Consequently, in the translations to Twelf and Automath, ψ becomes a function of two arguments: an object x_1 and a proof x_2 that x_1 is in A. In Twelf, we have

```
dsetconstr : {A:obj} ({x1:obj} pf (in A x1) -> prop) -> obj.
```

We omit the proof rules for `dsetconstr`.

It is important that in the set construction above, $\psi(x)$ can make use of the fact that $x \in A$ (as opposed to x being simply an object). This allows one to specify sets such as $\{x \in (\Re \setminus \{0\})|\frac{x^2-1}{x} = 0\}$ where one must know $x \neq 0$ in order to construct the term representing $\frac{x^2-1}{x}$.

These axioms are sufficient to describe all hereditarily finite sets. If one adds an axiom of infinity, one essentially obtains a form of Mac Lane set theory (Zermelo set theory with bounded quantifiers).

[2] This is a concrete example justifying reversing the usual order of arguments of `in`.

4 From Set Theory Axioms to Binary Relations

Starting from the axioms of set theory described above, one can define the usual propositional connectives as well as bounded quantification. Also, one can construct pairs and define binary relations as certain sets of pairs. This provides the infrastructure for defining functions (at the object level). We describe this infrastructure below.

First, we can define `true` and `false` as $\emptyset \in \{\emptyset\}$ and $\emptyset \in \emptyset$, respectively.

```
true:prop=({}::{{}}).
false:prop=({}::{}).
```

The important properties of `true` and `false` hold. Namely, there are terms inhabiting `pf true` and $\Pi P : \texttt{prop}.\texttt{pf false} \rightarrow \texttt{pf}\ P$.

For any proposition M, $\{x \in \{\emptyset\}|M\}$ is $\{\emptyset\}$ if M is true and $\emptyset$ if M is false. Using this set, we can embed the type of propositions into the type of objects.

```
(prop2set M):obj={x:{{}}|M}.
```

Using `prop2set`, we can define disjunction, implication and conjunction. The types corresponding to the usual natural deduction rules for these connectives are inhabited.

```
(or M N):prop=({{}}::{prop2set M,prop2set N}).
(imp M N):prop=((not M) | N).
(and M N):prop=(not (M => (not N))).
```

In PAM syntax, we can write `(M | N)`, `(M => N)`, and `(M & N)` for `(or M N)`, `(imp M N)`, and `(and M N)`, respectively.

If A is a set and $\psi(x)$ is a property of elements of A, then $\{x \in A|\psi(x)\} = A$ iff $\psi(x)$ holds for all $x \in A$. Similarly, $\{x \in A|\psi(x)\} \neq \emptyset$ iff $\psi(x)$ holds for some $x \in A$. We use these facts to define bounded quantifiers.

```
(dall A psi):prop=({x:A|psi x}==A).
(dex A psi):prop=(not ({x:A|psi x}=={})).
```

In PAM syntax, we write `(forall x:A . M)` and `(exists x:A . M)` as syntactic sugar for `(dall A (\x.M))` and `(dex A (\x.M))`, respectively.

In fact, `dall` and `dex` are bounded, *dependent* quantifiers. We can use the fact that x is in the set A in the construction of the proposition $x \in A$. Thus, we can sensibly represent a proposition such as $\exists x \in (\mathfrak{R} \setminus \{0\}).\frac{x^2-1}{x} = 0$.

Using bounded quantification, we can define subset.

```
(subset A B):prop=(forall x:A . (x::B)).
```

In PAM syntax, we can write `(A <= B)` for `(subset A B)`.

Binary union $A \cup B$ is defined as $\bigcup\{A, B\}$.

```
(binunion A B)=(setunion {A,B}).
```

In PAM syntax, we can write `(A \cup B)` for `(binunion A B)`.

A set A is a singleton if there is some x such that $A = \{x\}$. Since we only have bounded quantification, we must give a set in which that x must live. That is, we do not have a term corresponding to the proposition $\exists x.(A = \{x\})$. Instead we must use an appropriate set B and represent the proposition as $\exists x \in B.(A = \{x\})$. In this case, an appropriate choice of B is obvious: A.

```
(singleton A):prop=(exists x:A . (A=={x})).
```

Since `singleton` has type `obj` $\to$ `prop`, `class singleton` is a valid class type. In the PAM syntax, we can simply use the extraction `singleton` as a type:

```
[S:singleton]
```

Note that if S be a member of this class, then $\pi_1(S)$ has type `obj` and $\pi_2(S)$ has type `pf (singleton` $\pi_1(S)$`)`. In PAM syntax, one never explicitly writes π_1 and π_2 operators. If `S` is used where a term of type `obj` is expected, Scunak reconstructs the term π_1(`S`). If `S` is used where a term of type `pf (singleton` π_1(`S`)`)` is expected, Scunak reconstructs π_2(`S`). In particular, we write the proposition $(\bigcup S) \in S$ as `((setunion S)::S)` in PAM syntax. The reconstructed term is (`in` π_1(`S`) (`setunion` π_1(`S`))). We can declare a *claim* (i.e., a signature element for which a definition *will* be declared) called `theprop` of this proof type.

```
(theprop S):|- ((setunion S)::S)?
```

There is a term inhabiting this type, which we omit here. Once one gives the term as the definition (i.e., proof) of `theprop`, then `theprop` is an abbreviation and no longer a claim.

Using `theprop`, we can define a dependently typed description operator `the` as follows:

```
(the S):(in S)=<(setunion S),theprop S>.
```

Once the type and term are reconstructed, `the` has type

$$\Pi S : (\texttt{class singleton}).\texttt{class (in } \pi_1(S))$$

and is defined by the term $(\lambda S.\langle(\texttt{setunion } \pi_1(\texttt{S})), (\texttt{theprop S})\rangle)$. With the typing rules in Figure 1 and the given types of `setunion` and `theprop`, one can easily verify that the term indeed inhabits the type. Intuitively, given a singleton set S, (`the` S) is the unique member of S.

We can define a quantifier for unique existence using the singleton predicate..

```
(ex1 A psi):prop=(singleton {x:A|psi x}).
```

In PAM syntax, we write `(exists1 x:A . M)` for `(ex1 A (\x.M))`.

A set A is a Kuratowski pair if there exist u and v such that $A = \{\{u\}, \{u, v\}\}$. To define this notion using bounded quantification, we make use of $\bigcup A$ as a bound. One can prove that if any such u and v exist, they must inhabit $\bigcup A$.

```
(iskpair A):prop=(exists u:(setunion A) .
                  (exists v:(setunion A) . (A=={{u},{u,v}}))).
```

Given any objects y and z, $\{\{y\}, \{y, z\}\}$ is a Kuratowski pair. We can prove this and form an abbreviation `kpairiskpair`. Using such an abbreviation, we can define an operation `kpair` which takes two objects y and z and returns a member of the class type of Kuratowski pairs.

```
(kpair y z):iskpair=<{{y},{y,z}},kpairiskpair y z>.
```

In PAM syntax, we write `<<y,z>>` for the Kuratowski pair of `y` and `z`.

Using Kuratowski pairs, we can define the Cartesian product $A \times B$ of two sets A and B as follows:

```
(cartprod A B):obj
 ={x:powerset (powerset (A \cup B))|
    (exists u:A . (exists v:B . (x==<<u,v>>)))}.
```

In PAM syntax we write `(A \times B)` for `(cartprod A B)`.

We have already used the notation `{x:A|psi x}` for denoting $\{x \in A|\psi(x)\}$ in PAM syntax. When working with functions, we will need to consider sets of pairs. Informally, we can write $\{(u, v) \in A \times B|\phi(u, v)\}$. In order to support a corresponding PAM notation, we define a dependent set of pairs constructor.

```
[phi:A -> B -> prop]
(dpsetconstr A B phi):obj
 ={x:(A \times B)|
    (exists u:A . (exists v:B . ((phi u v) & (x==<<u,v>>))))}.
```

In PAM syntax, we write `{<<u,v>>:A \times B|M}` as syntactic sugar equivalent to `(dpsetconstr A B (\u v.M))`. (A single backslash in PAM notation binds a list of variables.)

Finally, we define the notion of a binary relation on two sets A and B in the usual way.

```
[R:obj]
(breln A B R):prop=(R<=(A \times B)).
```

This gives all the infrastructure necessary to define set-theoretic functions.

5 Representing Functions as Objects

Let A, B, and R be sets. We say R is a function from A to B if R is a binary relation on A and B and forall $x \in A$ there is a unique $y \in B$ such that the pair of x and y is in R. In PAM syntax, we can make this abbreviation as follows.

```
[A:set][B:set][R:obj]
(func A B R):prop
 =((breln A B R)&(forall x:A . (exists1 y:B . (<<x,y>>::R)))).
```

As before, Scunak reconstructs the π_1 operations. Note that `(<<x,y>>::R)` is PAM syntax for the term (`in R` π_1(`kpair` π_1(`x`) π_1(`y`))).

Since `(func A B)` has type `obj` $\to$ `prop`, `class (func A B)` is a valid class type. Let f have this type and let x have type A.

```
[f:(func A B)]
[x:A]
```

Using the definition of `func`, we can prove the set represented in PAM notation as `{y:B|<<x,y>>::f}` is a singleton. In the signature, `funcImageSingleton` is an abbreviation corresponding to this fact. Hence, the pair (in PAM syntax)

```
<{y:B|<<x,y>>::f},(funcImageSingleton A B f x)>
```

is of class type `class singleton`. Applying the description operator `the`, we obtain a member of `{y:B|<<x,y>>::f}`. One can prove the first component of `(the <{y:B|<<x,y>>::f},(funcImageSingleton A B f x)>)` is in `B`. In the signature, `apProp` abbreviates such a proof. Given this information, we can define an object level application as follows:

```
(ap A B f x):B
 =<(the <{y:B|<<x,y>>::f},(funcImageSingleton A B f x)>),
   (apProp A B f x)>.
```

The type of `ap` is

$$\Pi A : \texttt{obj}. \Pi B : \texttt{obj}. (\texttt{class}\ (\texttt{func}\, A\, B)) \to (\texttt{class}\ (\texttt{in}\ A)) \to \texttt{class}\ (\texttt{in}\ B)$$

It is perhaps instructive to compare this to the Twelf version of `ap` obtain by translating from Scunak. Since `ap` returns a member of the class type `class` (`in` B), there are two corresponding Twelf abbreviations. (Due to a use of `%abbrev`, the description operator `the` is expanded in terms of `setunion` in the Twelf version.)

```
%abbrev
ap : {A:obj} {B:obj} {f:obj}
           pf (func A B f) -> ({x:obj} pf (in A x) -> obj)
   = [A:obj] [B:obj] [f:obj] [fp:pf (func A B f)]
     [x:obj] [xp:pf (in A x)]
 (setunion
  (dsetconstr B ([y:obj] [yp:pf (in B y)] in f (kpair x y)))).

ap_pf :
   {A:obj} {B:obj} {f:obj} pf (func A B f)
    -> ({x:obj} pf (in A x)
        -> pf (in B (setunion (dsetconstr B
                               ([y:obj] [yp:pf (in B y)]
                                in f (kpair x y))))))
   = [A:obj] [B:obj] [f:obj] [fp:pf (func A B f)]
     [x:obj] [xp:pf (in A x)] apProp A B f fp x xp.
```

Note that in Twelf, `ap` is a function of six arguments instead of four. In particular, the object `f` is separated from the proof `fp` that `f` is a function from `A` to `B`. Likewise, the object `x` is separated from the proof `xp` that `x` is a member of `A`. Intuitively, the Twelf abbreviation `ap` returns the object corresponding to $f(x)$ and the Twelf abbreviation `ap_pf` returns the proof that $f(x)$ is in B.

Similarly, we can define an object-level λ-abstraction operator. Intuitively, this reifies a meta-level function g from A to B to be an object-level function from A to B. Let g have type $(\texttt{class}\ (\texttt{in}\ A)) \to (\texttt{class}\ (\texttt{in}\ B))$. In PAM syntax, we write `[g:A -> B]`. We can prove the set of pairs represented in PAM syntax by `{<<x,y>>:A \times B|((g x)==y)}` is a function from A to B. We abbreviate this proof as `lamProp`. Using this, we can define the abstraction operator `lam` as follows.

```
(lam A B g):(func A B)
 =<{<<x,y>>:A \times B|((g x)==y)},(lamProp A B g)>.
```

The type of `lam` is

$$\Pi A : \texttt{obj}.\, \Pi B : \texttt{obj}.(\texttt{class}\ (\texttt{in}\, A) \to \texttt{class}\ (\texttt{in}\, B)) \to \texttt{class}\ (\texttt{func}\, A\, B)$$

Note that the types of `ap` and `lam` have the form one expects when coding simply typed λ-calculus using higher-order abstract syntax. In particular, `ap` takes an object-level function in `class (func A B)` to a meta-level function `class A` $\to$ `class B` and `lam` takes such a meta-level function to such an object-level function. However, the intention is quite different. We are not encoding *syntax* of simply typed λ-terms, but the standard set theoretic *semantics* of simply typed λ-terms. Consequently, we can prove properties which hold in such standard models. For example, we can prove functional extensionality and soundness of β- and η-conversion.

Functional extensionality states that two functions $f, k : A \to B$ are equal if they return the same value on all $x \in A$. We can declare functional extensionality as a claim `funcext` in PAM syntax.

```
[k:(func A B)]
[eqfkx:{x:A}|- ((ap A B f x)==(ap A B k x))]
(funcext A B f k eqfkx):|- (f==k)?
```

In the PAM file, the proof (i.e., definition) is given following the declaration of the claim. We omit this proof term here.

Finally, we can prove the object-level versions of β-equality and η-equality. We omit the proof terms and only show the declarations of the claims.

```
(beta1 A B g x):|- ((ap A B (lam A B g) x)==(g x))?
(eta1 A B f):|- ((lam A B (ap A B f))==f)?
```

6 Comparing the Signatures

The construction of functions starting from the given axioms of set theory can be encoded in Scunak by giving a signature of 23 constants and 112 abbreviations. By Currying, one obtains corresponding Twelf and Automath signatures. Each of these signatures contains 3 declarations for the type families, 23 constants and 116 abbreviations. In particular, 4 of the Scunak abbreviations (`the`, `kpair`, `ap`, and `lam`) return a class type and therefore correspond to 8 abbreviations in Twelf and Automath. In Twelf, 11 abbreviations must be declared using `%abbrev` since there is no strict occurrence of some argument. Each of the three systems can type check the signature in less than a second.

7 Conclusion

Scunak provides a convenient way to specify a set theory and represent mathematics within the set theory. Two of the reasons for the naturality of mathematics represented in Scunak are class types and the PAM syntax. Class types allow one to treat arbitrary predicates (set-theoretic classes) as subtypes of the type of (untyped) mathematical objects. The PAM syntax allows one to give types and terms in a reasonably natural way.

References

[1] Brown, C. E., *Combining Type Theory and Untyped Set Theory*, in: *IJCAR 2006*, Seattle, Washington, 2006, to appear.

[2] Brown, C. E., *Verifying and Invalidating Textbook Proofs using Scunak*, in: *Mathematical Knowledge Management, MKM 2006*, Wokingham, England, 2006, to appear.

[3] Byliński, C., *Functions and their basic properties*, Journal of Formalized Mathematics **1** (1989), http://mizar.org/JFM/Vol1/funct_1.html.

[4] de Bruijn, N. G., *A survey of the project AUTOMATH*, in: J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, Academic Press, 1980 pp. 579–606.

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

[6] Megill, N., *Metamath Home Page*, http://au.metamath.org/index.html.

[7] Paulson, L. C., *Set Theory for Verification: I. From Foundations to Functions*, Journal of Automated Reasoning **11** (1993), pp. 353–389.

[8] Pfenning, F. and C. Schürmann, *System Description: Twelf–A Meta-Logical Framework for Deductive Systems*, in: H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence **1632** (1999), pp. 202–206.

[9] Wiedijk, F., *A new implementation of Automath*, J. Autom. Reasoning **29** (2002), pp. 365–387.

[10] Wiedijk, F., *Is ZF a hack? Comparing the complexity of some (formalist interpretations of) foundational systems for mathematics*, Journal of Applied Logic **4** (2006), to appear.