Type Inference for ZFH

Type Inference for ZFH

Steven Obua, Jacques Fleuriot, Phil Scott, and David Aspinall

School of Informatics, Edinburgh University
10 Crichton Street, EH8 9AB Edinburgh, Scotland, UK
www.proofpeer.net

Abstract. *ZFH* stands for Zermelo-Fraenkel set theory implemented in higher-order logic. It is a descendant of Agerholm's and Gordon's HOL-ST but does not allow the use of type variables nor the definition of new types. We first motivate why we are using ZFH for *ProofPeer*, the collaborative theorem proving system we are building. We then focus on the type inference algorithm we have developed for ZFH. In ZFH's syntax, function application, written as juxtaposition, is overloaded to be either set-theoretic or higher-order. Our algorithm extends Hindley-Milner type inference to cope with this particular overloading of function application. We describe the algorithm, prove its correctness, and discuss why prior general approaches to type inference in the presence of coercions or overloading do not cover our particular case.

1 Introduction

The *ProofPeer* project [1, 2] is our attempt to combine interactive theorem proving (ITP) and the modern web, making ITP technology more accessible than it has been. We will first explain why we have chosen ZFH as the logic of ProofPeer, and then introduce the problem this paper solves.

1.1 Why ZFH?

Despite a few prominent counter examples [3, 4] it is particularly astonishing how few mathematicians are aware of or even use ITP systems. We believe that one reason for this is that traditionally the development and application of ITP technology has been driven by computer scientists, not mathematicians. Major successful ITP systems like Isabelle and Coq are based on variants of type theory, while most mathematicians feel more familiar with set theory. Simple mathematical standards like point set topology cannot be formalized in either system without the result feeling alien to most mathematicians.

We have therefore decided that the logic used in the ProofPeer system should be based on Zermelo-Fraenkel set theory which is more or less familiar to all mathematicians. At the same time we want to build on the considerable technical advances that contemporary ITP systems have achieved. Therefore we embed set theory within simply-typed classical higher-order logic by introducing a special type $\mathscr{U}$ which forms the universe of Zermelo-Fraenkel sets, additional

constants like the element-of operator $\in$: $\mathscr{U} \to \mathscr{U} \to \mathbb{P}$, and additional axioms describing the properties of these new constants. The symbol $\mathbb{P}$ denotes the type of propositions / booleans, and for any two types α and β we can form the type of higher-order functions $\alpha \to \beta$. For a full list of all new constants and axioms see theory root [9] in the ProofPeer system.

Because technically, all we have done is add an additional type together with a few new constants and axioms, all of the machinery present in systems like HOL-4 or Isabelle/HOL can be ported to work in our system. For example, Isabelle/HOL's facilities for defining partial and nested recursive functions [8] could be translated to ProofPeer.

This approach was first advocated by Agerholm and Gordon [5]. They called the resulting logic HOL-ST. A related approach is pursued by Isabelle/ZF which embeds set theory within its intuitionistic higher-order meta logic [7]. The Isabelle/ZF approach seems more involved than our approach: it begins at base with *intuitionistic* higher-order logic, over which first-order classical logic is introduced, which in turn, is used to formalise set theory. We instead skip the middle step and base set theory directly on *classical* higher-order logic, obtaining a more powerful logic by simpler means. This is just how HOL-ST works as well, but that opens up a new dilemma: HOL-ST is so powerful that often it is not clear how concepts should best be formalised. Take for example the natural numbers: should they be formalised as a type, or should they be formalised as a set, i.e. as an element of $\mathscr{U}$? Or take lists: should they be formalised as a type α list together with polymorphic operations like $\mathsf{cons} : \alpha \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list}$, or should they be formalised as a constant $\mathsf{list} : \mathscr{U} \to \mathscr{U}$ such that $\mathsf{list}\ \alpha$ denotes the set of lists over elements of α? Note how in the latter case we can extend our discussion to the *class* of all (heterogeneous) lists by defining

$$\mathsf{isList}\ l = \exists \alpha.\ l \in \mathsf{list}\ \alpha$$

The type of cons would now be $\mathscr{U} \to \mathscr{U} \to \mathscr{U}$ and for reasonable definitions of list we could prove theorems like

$$\forall l.\ \mathsf{isList}\ l \to \forall x.\ \mathsf{isList}(\mathsf{cons}\ x\ l)$$
$$\forall l\ \alpha.\ l \in \mathsf{list}\ \alpha \to \forall x \in \alpha.\ \mathsf{cons}\ x\ l \in \mathsf{list}\ \alpha$$

We want people to perceive ProofPeer as a system based on set theory; the only reason we also employ simply-typed higher-order logic is because of its technical convenience and simplicity. Therefore for us there is an easy and coherent way out of the dilemma that HOL-ST has: we forbid the introduction of new types besides the ones we already described, and we furthermore do not use type variables as part of our internal term representation. The only polymorphic constants in our logic are equality ($=$), universal quantification ($\forall$) and existential quantification ($\exists$), and we do not provide any means for defining additional ones.

Abstaining from polymorphic terms in favour of monomorphic ones has a further advantage noticed already by Gordon [6, Section 3]: We can treat theories as simple (albeit large) theorems. The axioms of the theory become the antecedents

of the theorem, and constants declared in the theory can be treated as universally quantified variables. This doesn't work in polymorphic simply-typed higher order logic because polymorphic constants can appear with different types in the theory but variables must appear always with the same type in the theorem.

We choose the name *ZFH* for the logical system we obtain by embedding set theory into classical higher-order logic in the way outlined above. ZFH represents the same logic as HOL-ST minus type variables and minus a mechanism for defining custom types. In particular this means that ZFH and HOL-ST are equiconsistent, and that both HOL and ZFC can be formalized and proven to be consistent within ZFH.

1.2 Set-theoretic vs. Higher-order Function Application

There are two kinds of function application in ZFH:

- application of a higher-order function $f : \alpha \to \beta$ to its argument $x : \alpha$, and
- application of a set-theoretic function $f : \mathscr{U}$ to its argument $x : \mathscr{U}$.

In ZFH, set-theoretic functions are governed by two properties:

$$\forall X f.\ \mathsf{fun}\ X f = \{(x, f\ x) \mid x \in X\}$$
$$\forall X f.\ \forall x \in X.\ \mathsf{apply}\ (\mathsf{fun}\ X\ f)\ x = f\ x$$

Here $\mathsf{fun} : \mathscr{U} \to (\mathscr{U} \to \mathscr{U}) \to \mathscr{U}$ takes a domain $X : \mathscr{U}$ and a higher-order function $f : \mathscr{U} \to \mathscr{U}$ as its arguments and produces the corresponding set-theoretic function on that domain. Set-theoretic functions created thus can then be applied via $\mathsf{apply} : \mathscr{U} \to \mathscr{U} \to \mathscr{U}$.

In the actual ProofPeer theory [9], the second property is written like this:

$$\forall X f.\ \forall x \in X.\ \mathsf{fun}\ X\ f\ x = f\ x$$

Instead of explicitly mentioning apply we write application of a set-theoretic function in exactly the same way as application of a higher-order function! This is possible because in the above $\mathsf{fun}\ X\ f$ is a set, which leads type inference to conclude that set-theoretic function application must be meant, not higher-order function application.

In general, the situation is not so clear-cut. Consider the following term:

$$\forall x.\ \exists f.\ f\ x = x$$

Informally the above says that every x is the fixpoint of some function f. But which types should we assign to f and x? There are infinitely many valid ones:

1. $f : \mathscr{U}$ and $x : \mathscr{U}$
2. $f : \mathbb{P} \to \mathbb{P}$ and $x : \mathbb{P}$
3. $f : \mathscr{U} \to \mathscr{U}$ and $x : \mathscr{U}$
4. $f : (\mathscr{U} \to \mathscr{U}) \to (\mathscr{U} \to \mathscr{U})$ and $x : \mathscr{U} \to \mathscr{U}$
5. $f : (\mathbb{P} \to \mathscr{U}) \to (\mathbb{P} \to \mathscr{U})$ and $x : \mathbb{P} \to \mathscr{U}$

... and so on

Even if we had type variables at our disposal to formulate the typing (which we don't) there would still be two equally valid typings to choose from:

1. $f : \mathscr{U}$ and $x : \mathscr{U}$
2. $f : \alpha \to \alpha$ and $x : \alpha$

Which one should we pick?

In the next section we will present a type inference algorithm for ZFH with the following properties:

- If there is a valid typing at all, the algorithm will find one, and will otherwise fail. In particular, all function applications will be resolved to be either set-theoretic or higher-order.
- Preference is given to the type $\mathscr{U}$ over all other types, and to set-theoretic function application over higher-order function application.

Note that the second property is a desirable one in our case, as this again emphasises the set theory focus of ProofPeer.

In our above example the algorithm yields then the typing $f : \mathscr{U}$ and $x : \mathscr{U}$.

2 The Type Inference Algorithm

We first introduce the types and terms our algorithm operates on. Then we introduce the type equations which guide the algorithm, and recall how to solve type equations. After highlighting the basic difficulties of the problem we state the algorithm. Finally we prove that the algorithm terminates, that it is sound, and in what sense it is complete.

2.1 Types and Terms

Although we do not allow type variables as part of proper ZFH terms, we do allow them for type inference purposes. In particular a *pretype* τ is either the universal type $\mathscr{U}$, the propositional/boolean type $\mathbb{P}$, a function type $\tau_1 \to \tau_2$, or a type variable α:

$$\tau \ ::= \ \mathscr{U} \mid \mathbb{P} \mid \tau_1 \to \tau_2 \mid \alpha.$$

A *type* is a pretype which does not contain any type variables. A *preterm* t is either a constant c, a polymorphic constant $p[\tau]$, an explicit typing $t : \tau$, a higher-order function $x : \tau_1 \mapsto t : \tau_2$, a variable x, a higher-order function application $t_1 \diamond_{\mathsf{H}} t_2 : \tau$, a set-theoretic function application $t_1 \diamond_{\mathsf{ZF}} t_2 : \tau$, or a function application $t_1 \diamond_? t_2 : \tau$ where it is unspecified if it is of higher-order or set-theoretic kind:

$$t \ ::= c \mid p[\tau] \mid t : \tau \mid x : \tau_1 \mapsto t : \tau_2 \mid x \mid t_1 \diamond_{\mathsf{H}} t_2 : \tau \mid t_1 \diamond_{\mathsf{ZF}} t_2 : \tau \mid t_1 \diamond_? t_2 : \tau.$$

A *term* is a preterm which does not contain any type variables, nor any function applications of unspecified kind.

Example 1. Our introductory example $\forall x.\ \exists f.\ fx = x$ corresponds to the preterm:

$$\forall[\alpha_1]\ \diamond_{\mathsf{H}}\ (x : \alpha_2 \mapsto (\exists[\alpha_3]\ \diamond_{\mathsf{H}}\ (f : \alpha_4 \mapsto ((= [\alpha_5]\ \diamond_{\mathsf{H}}\ (f\ \diamond_?\ x : \alpha_6) : \alpha_7)\ \diamond_{\mathsf{H}}\ x : \alpha_8) : \alpha_9) : \alpha_{10}) : \alpha_{11}) : \alpha_{12}.$$

Note that everywhere our preterm format requires a type, we simply used a fresh type variable.

2.2 Type Equations

A *substitution* σ associates every type variable α with a pretype σ_α. Applying a substitution to a pretype τ means replacing every type variable in τ by its associated pretype (fig. 1), and applying a substitution to a preterm t means applying the substitution to every pretype in t (fig. 2).

Fig. 1. Applying a substitution σ to a pretype

$$\begin{aligned}
\sigma(\mathscr{U}) &= \mathscr{U} \\
\sigma(\mathbb{P}) &= \mathbb{P} \\
\sigma(\tau_1 \to \tau_2) &= \sigma(\tau_1) \to \sigma(\tau_2) \\
\sigma(\alpha) &= \sigma_\alpha
\end{aligned}$$

Fig. 2. Applying a substitution σ to a preterm

$$\begin{aligned}
\sigma(c) &= c \\
\sigma(p[\tau]) &= p[\sigma(\tau)] \\
\sigma(t : \tau) &= \sigma(t) : \sigma(\tau) \\
\sigma(x : \tau_1 \mapsto t : \tau_2) &= x : \sigma(\tau_1) \mapsto \sigma(t) : \sigma(\tau_2) \\
\sigma(x) &= x \\
\sigma(t_1 \diamond_{\mathsf{H}} t_2 : \tau) &= \sigma(t_1) \diamond_{\mathsf{H}} \sigma(t_2) : \sigma(\tau) \\
\sigma(t_1 \diamond_{\mathsf{ZF}} t_2 : \tau) &= \sigma(t_1) \diamond_{\mathsf{ZF}} \sigma(t_2) : \sigma(\tau) \\
\sigma(t_1 \diamond_? t_2 : \tau) &= \sigma(t_1) \diamond_? \sigma(t_2) : \sigma(\tau)
\end{aligned}$$

With each constant c a fixed type $\mathcal{C}(c)$ is associated, e.g. $\mathcal{C}(\mathsf{apply}) = \mathscr{U} \to \mathscr{U} \to \mathscr{U}$. Assuming also a partial map $\mathcal{V}$ from variables to pretypes we can associate

with each preterm t its type $\Gamma_{\mathcal{C},\mathcal{V}}(t)$ and a set of equations between pretypes $\mathcal{E}_{\mathcal{C},\mathcal{V}}(t)$ as shown in Fig. 3. In the following we will assume an implicitly given $\mathcal{C}$ and define

$$\mathcal{E}(t) = \mathcal{E}_{\mathcal{C},\emptyset}(t)$$

where $\emptyset$ in this context denotes the empty map.

Fig. 3. Definition of $\Gamma_{\mathcal{C},\mathcal{V}}$ and $\mathcal{E}_{\mathcal{C},\mathcal{V}}$

$$\begin{aligned}
\Gamma_{\mathcal{C},\mathcal{V}}(c) &= \mathcal{C}(c)\\
\mathcal{E}_{\mathcal{C},\mathcal{V}}(c) &= \emptyset\\
\Gamma_{\mathcal{C},\mathcal{V}}(p[\tau]) &= \begin{cases}(\tau \to \mathbb{P}) \to \mathbb{P} & \text{if } p \in \{\forall, \exists\}\\ \tau \to \tau \to \mathbb{P} & \text{if } p \in \{=\}\end{cases}\\
\mathcal{E}_{\mathcal{C},\mathcal{V}}(p[\tau]) &= \emptyset\\
\Gamma_{\mathcal{C},\mathcal{V}}(t : \tau) &= \tau\\
\mathcal{E}_{\mathcal{C},\mathcal{V}}(t : \tau) &= \mathcal{E}_{\mathcal{C},\mathcal{V}}(t) \cup \{\Gamma_{\mathcal{C},\mathcal{V}}(t) \equiv \tau\}\\
\Gamma_{\mathcal{C},\mathcal{V}}(x : \tau_1 \mapsto t : \tau_2) &= \tau_1 \to \tau_2\\
\mathcal{E}_{\mathcal{C},\mathcal{V}}(x : \tau_1 \mapsto t : \tau_2) &= \mathcal{E}_{\mathcal{C},\mathcal{W}}(t) \cup \{\Gamma_{\mathcal{C},\mathcal{W}}(t) \equiv \tau_2\} \text{ where } \mathcal{W} = \mathcal{V}[x := \tau_1]\\
\Gamma_{\mathcal{C},\mathcal{V}}(x) &= \begin{cases}\mathcal{V}(x) & \text{if } \mathcal{V} \text{ is defined at } x\\ \mathscr{U} & \text{otherwise}\end{cases}\\
\mathcal{E}_{\mathcal{C},\mathcal{V}}(x) &= \begin{cases}\emptyset & \text{if } \mathcal{V} \text{ is defined at } x\\ \{\mathscr{U} \equiv \mathbb{P}\} & \text{otherwise}\end{cases}\\
\Gamma_{\mathcal{C},\mathcal{V}}(t_1 \diamond_{\mathsf{H}} t_2 : \tau) &= \tau\\
\mathcal{E}_{\mathcal{C},\mathcal{V}}(t_1 \diamond_{\mathsf{H}} t_2 : \tau) &= \mathcal{E}_{\mathcal{C},\mathcal{V}}(t_1) \cup \mathcal{E}_{\mathcal{C},\mathcal{V}}(t_2) \cup \{\Gamma_{\mathcal{C},\mathcal{V}}(t_1) \equiv \Gamma_{\mathcal{C},\mathcal{V}}(t_2) \to \tau\}\\
\Gamma_{\mathcal{C},\mathcal{V}}(t_1 \diamond_{\mathsf{ZF}} t_2 : \tau) &= \tau\\
\mathcal{E}_{\mathcal{C},\mathcal{V}}(t_1 \diamond_{\mathsf{ZF}} t_2 : \tau) &= \mathcal{E}_{\mathcal{C},\mathcal{V}}(t_1) \cup \mathcal{E}_{\mathcal{C},\mathcal{V}}(t_2) \cup \{\Gamma_{\mathcal{C},\mathcal{V}}(t_1) \equiv \mathscr{U}, \Gamma_{\mathcal{C},\mathcal{V}}(t_2) \equiv \mathscr{U}, \tau \equiv \mathscr{U}\}\\
\Gamma_{\mathcal{C},\mathcal{V}}(t_1 \diamond_{?} t_2 : \tau) &= \tau\\
\mathcal{E}_{\mathcal{C},\mathcal{V}}(t_1 \diamond_{?} t_2 : \tau) &= \mathcal{E}_{\mathcal{C},\mathcal{V}}(t_1) \cup \mathcal{E}_{\mathcal{C},\mathcal{V}}(t_2)
\end{aligned}$$

A substitution σ is a *unifier* of a set $\mathcal{E}$ of equations of pretypes iff for all equations $l \equiv r \in \mathcal{E}$ the left hand side and the right hand side of the equation become identical after substitution, i.e. $\sigma(l) = \sigma(r)$ holds. We call $\mathcal{E}$ *solvable* if it has a unifier. Defining $\sigma(\mathcal{E}) = \{\sigma(l) \equiv \sigma(r) \mid l \equiv r \in \mathcal{E}\}$ allows the following rephrasing: σ is a unifier of $\mathcal{E}$ iff $\sigma(\mathcal{E})$ is a set of identities.

Substitutions can be composed. The composition $\delta \circ \sigma$ of a substitution σ with a substitution δ is defined via

$$(\delta \circ \sigma)_\alpha = \delta(\sigma_\alpha).$$

A unifier σ_1 is called more general than a unifier σ_2, in symbols $\sigma_1 \geq \sigma_2$, iff there is a substitution δ such that $\sigma_2 = \delta \circ \sigma_1$.

Lemma 1. *If $\mathcal{E}$ is solvable then it has an idempotent most general unifier $\mathsf{mgu}_{\mathcal{E}}$, i.e. the following two properties hold for $\mathsf{mgu}_{\mathcal{E}}$:*

1. *$\mathsf{mgu}_{\mathcal{E}} \geq \sigma$ for any unifier σ of $\mathcal{E}$, and*
2. *$\mathsf{mgu}_{\mathcal{E}} \circ \mathsf{mgu}_{\mathcal{E}} = \mathsf{mgu}_{\mathcal{E}}$.*

Proof. See [10, section 4.5]. □

If $\mathcal{E}(t)$ is solvable for a given preterm t, then we define

$$\mathcal{S}(t) = \mathsf{mgu}_{\mathcal{E}(t)}(t).$$

Note that $\mathcal{S}(t)$ is unique up to a renaming of type variables. Computation of $\mathcal{S}(t)$ is known as *Hindley-Milner type inference* [11].

Example 2. Given the preterm t from Example 1, the type equations $\mathcal{E}(t)$ are:

$$\begin{aligned}
(\alpha_1 \to \mathbb{P}) \to \mathbb{P} &\equiv (\alpha_2 \to \alpha_{11}) \to \alpha_{12} \\
\alpha_{10} &\equiv \alpha_{11} \\
(\alpha_3 \to \mathbb{P}) \to \mathbb{P} &\equiv (\alpha_4 \to \alpha_9) \to \alpha_{10} \\
\alpha_8 &\equiv \alpha_9 \\
\alpha_7 &\equiv \alpha_2 \to \alpha_8 \\
\alpha_5 \to \alpha_5 \to \mathbb{P} &\equiv \alpha_6 \to \alpha_7
\end{aligned}$$

A most general unifier for these equations of pretypes is given by

$$\mathsf{mgu}_{\mathcal{E}(t)}(\alpha_i) = \begin{cases} \alpha & \text{if } i \in \{1,2,5,6\} \\ \beta & \text{if } i \in \{3,4\} \\ \alpha \to \mathbb{P} & \text{if } i = 7 \\ \mathbb{P} & \text{if } i \in \{8,9,10,11,12\} \end{cases}$$

and therefore

$$\begin{aligned}
\mathcal{S}(t) \;=\; & \forall[\alpha]\, \diamond_{\mathsf{H}}\, (x : \alpha \mapsto (\exists[\beta]\, \diamond_{\mathsf{H}} \\
& (f : \beta \mapsto ((= [\alpha]\, \diamond_{\mathsf{H}}\, (f\, \diamond_{?}\, x : \alpha) : \alpha \to \mathbb{P})\, \diamond_{\mathsf{H}}\, x : \mathbb{P}) : \mathbb{P}) : \mathbb{P}) : \mathbb{P}) : \mathbb{P}.
\end{aligned}$$

2.3 A First Attempt

An obvious first attempt to solve our type inference problem for a given preterm t would be to single out all n occurrences of $\diamond_?$ in t and to form all 2^n possibilities t_i by replacing $\diamond_?$ with either $\diamond_{\mathsf{H}}$ or $\diamond_{\mathsf{ZF}}$.

If none of the sets of type equations $\mathcal{E}(t_i)$ is solvable, type inference fails. Otherwise let t_j denote those t_i for which $\mathcal{E}(t_i)$ is solvable. This gives us up to 2^n almost-solutions s_j where

$$s_j = \mathcal{S}(t_j).$$

Because the s_j possibly contain type variables, but proper ZFH terms may not contain type variables, we need to somehow eliminate all type variables from the s_j. One rather arbitrary way of doing so would be to replace all type variables by $\mathscr{U}$, i.e. to form

$$r_j = \mathcal{U}(s_j)$$

where $\mathcal{U}$ is the substitution which replaces all type variables by $\mathscr{U}$:

$$\mathcal{U}_\alpha = \mathscr{U} \text{ for all type variables } \alpha.$$

This leaves us finally with up to 2^n possible solutions r_j to our type inference problem. Computing all of these solutions is not practical for obvious performance reasons; furthermore, even if we *did* compute all of them, it is not clear which one among them we should pick as the result of the type inference.

2.4 The Algorithm

In our above attempt at a type inference algorithm we computed $\mathcal{S}(t_i)$ only for preterms t_i which did not contain any occurrences of $\diamond_?$. This was an arbitrary choice we made and it did not pay off.

Instead, given a preterm t which may still contain occurrences of $\diamond_?$, let us directly compute $t_0 = \mathcal{S}(t)$ if $\mathcal{E}(t)$ is solvable. If t contained any occurrences of $\diamond_?$, then so will t_0, *but we now might have more type information available to decide whether an occurrence of* $\diamond_?$ *should really be replaced by* $\diamond_{\mathsf{H}}$ *or* $\diamond_{\mathsf{ZF}}$*!*

To exploit type information present in a preterm t we define a function $\mathcal{D}(t)$ which is able to decide in certain situations whether an occurrence of $\diamond_?$ in t should be converted into $\diamond_{\mathsf{H}}$ or into $\diamond_{\mathsf{ZF}}$ (fig. 4). Analogously to the definition of $\mathcal{E}(t)$ in terms of $\mathcal{E}_{\mathcal{C},\mathcal{V}}(t)$ we define $\mathcal{D}(t)$ in terms of $\mathcal{D}_{\mathcal{C},\mathcal{V}}(t)$. The main work in $\mathcal{D}$

Fig. 4. Definition of $\mathcal{D}$

$$\begin{aligned}
\mathcal{D}(t) &= \mathcal{D}_{\mathcal{C},\emptyset}(t)\\
\mathcal{D}_{\mathcal{C},\mathcal{V}}(c) &= c\\
\mathcal{D}_{\mathcal{C},\mathcal{V}}(p[\tau]) &= p[\tau]\\
\mathcal{D}_{\mathcal{C},\mathcal{V}}(t:\tau) &= \mathcal{D}_{\mathcal{C},\mathcal{V}}(t):\tau\\
\mathcal{D}_{\mathcal{C},\mathcal{V}}(x:\tau_1 \mapsto b:\tau_2) &= x:\tau_1 \mapsto \mathcal{D}_{\mathcal{C},\mathcal{W}}(b):\tau_2 \text{ where } \mathcal{W} = \mathcal{V}[x := \tau_1]\\
\mathcal{D}_{\mathcal{C},\mathcal{V}}(x) &= x\\
\mathcal{D}_{\mathcal{C},\mathcal{V}}(t_1 \diamond_{\mathsf{H}} t_2 : \tau) &= \mathcal{D}_{\mathcal{C},\mathcal{V}}(t_1) \diamond_{\mathsf{H}} \mathcal{D}_{\mathcal{C},\mathcal{V}}(t_2) : \tau\\
\mathcal{D}_{\mathcal{C},\mathcal{V}}(t_1 \diamond_{\mathsf{ZF}} t_2 : \tau) &= \mathcal{D}_{\mathcal{C},\mathcal{V}}(t_1) \diamond_{\mathsf{ZF}} \mathcal{D}_{\mathcal{C},\mathcal{V}}(t_2) : \tau\\
\mathcal{D}_{\mathcal{C},\mathcal{V}}(t_1 \diamond_? t_2 : \tau) &= \mathcal{D}_{\mathcal{C},\mathcal{V}}(t_1) \diamond (\Gamma_{\mathcal{C},\mathcal{V}}(t_1), \Gamma_{\mathcal{C},\mathcal{V}}(t_2), \tau)\ \mathcal{D}_{\mathcal{C},\mathcal{V}}(t_2) : \tau
\end{aligned}$$

is done by the function

$$\diamond(\tau_1, \tau_2, \tau_3) \in \{\diamond_{\mathsf{H}}, \diamond_{\mathsf{ZF}}, \diamond_{?}\}$$

which takes three pretypes τ_1, τ_2, τ_3 as arguments and tries to determine which kind of function application fx must be when the type of f is known to be τ_1, the type of x is known to be τ_2 and the type of fx is known to be τ_3. If any of the τ_i cannot be the type $\mathscr{U}$, in symbols $\neg^{\mathscr{U}}(\tau_i)$, then we know that fx cannot be set-theoretic function application and therefore can only be (if any at all) higher-order function application. On the other hand, if the type of τ_1 cannot be a function type, in symbols $\neg^{\rightarrow}(\tau_1)$, then fx cannot be higher-order function application and can therefore be only (if any at all) set-theoretic function application (fig. 5).

Fig. 5. Definition of $\diamond(\tau_1, \tau_2, \tau_3)$

$$\neg^{\mathscr{U}}(\tau) = \begin{cases} \mathsf{true} & \text{if } \tau = \mathbb{P} \\ \mathsf{true} & \text{if } \tau = \omega_1 \to \omega_2 \text{ for some pretypes } \omega_1 \text{ and } \omega_2 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\neg^{\rightarrow}(\tau) = \begin{cases} \mathsf{true} & \text{if } \tau = \mathbb{P} \\ \mathsf{true} & \text{if } \tau = \mathscr{U} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\diamond(\tau_1, \tau_2, \tau_3) = \begin{cases} \diamond_{\mathsf{H}} & \text{if } \neg^{\mathscr{U}}(\tau_1) \text{ or } \neg^{\mathscr{U}}(\tau_2) \text{ or } \neg^{\mathscr{U}}(\tau_3) \\ \diamond_{\mathsf{ZF}} & \text{else if } \neg^{\rightarrow}(\tau_1) \\ \diamond_{?} & \text{otherwise} \end{cases}$$

We have now gathered all the pieces to formulate our type inference algorithm as shown in Fig. 6.

Example 3. Continuing Example 2 we compute now $\mathsf{TypeInfer}(t)$. Having already computed $s = \mathcal{S}(t)$ we now need to compute $\mathcal{D}(s)$. There is only one occurrence of $\diamond_{?}$ in s and the corresponding invocation of $\diamond$ yields

$$\diamond(\beta, \alpha, \alpha) = \diamond_{?}$$

and thus $\mathcal{D}(s) = s$. This means that no recursive call to $\mathsf{TypeInfer}$ is necessary and therefore

$$\begin{aligned} \mathsf{TypeInfer}(t) = \mathcal{D}(\mathcal{U}(s)) = \forall[\mathscr{U}]\, \diamond_{\mathsf{H}}\, (x : \mathscr{U} \mapsto (\exists[\mathscr{U}]\, \diamond_{\mathsf{H}} \\ (f : \mathscr{U} \mapsto ((= [\mathscr{U}]\, \diamond_{\mathsf{H}}\, (f\, \diamond_{\mathsf{ZF}}\, x : \mathscr{U}) : \mathscr{U} \to \mathbb{P})\, \diamond_{\mathsf{H}}\, x : \mathbb{P}) : \mathbb{P}) : \mathbb{P}) : \mathbb{P}) : \mathbb{P}. \end{aligned}$$

Fig. 6. The Type Inference Algorithm

```
TypeInfer(t) =
    if E(t) is not solvable then
        fail
    else
        s = S(t)
        d = D(s)
        if s = d then
            D(U(d))
        else
            TypeInfer(d)
        end
    end
```

2.5 Termination

Let us first show that our algorithm actually terminates. There are only finitely many occurrences of $\diamond_?$ in a preterm t, let us denote the number of such occurrences by $N(t)$. For two preterms s and t let us write $s \sqsubseteq t$ if s arises from t by replacing some (or none) of the occurrences of $\diamond_?$ in t by either $\diamond_{\mathsf{H}}$ or $\diamond_{\mathsf{ZF}}$. Obviously $s \sqsubseteq t$ together with $s \neq t$ implies $N(s) < N(t)$.

Lemma 2. TypeInfer*(t) terminates for every preterm t.*

Proof. Given some preterm s, $\mathcal{D}(s) \sqsubseteq s$ holds. Therefore $s \neq \mathcal{D}(s)$ implies

$$N(\mathcal{D}(s)) < N(s).$$

We also know that $N(t) = N(\mathcal{S}(t))$ because $\mathcal{S}$ only possibly instantiates type variables and leaves occurrences of $\diamond_?$ unchanged. Together this means that for each recursive call to TypeInfer its argument strictly decreases as measured by N and therefore the algorithm must terminate. □

2.6 Soundness and Completeness

Given two preterms t and t' we say that t' is an instance of t, in symbols

$$t' \leq t$$

iff there is a substitution σ such that $t' \sqsubseteq \sigma(t)$.

What does it mean for our type inference algorithm to be sound? Given a preterm t as input it should output a preterm t' such that

1. t' is a term,
2. $t' \leq t$, and
3. $\mathcal{E}(t')$ is solvable.

If there is no such t' the algorithm should fail. If there are several possible candidates for t' it would also be good to have a simple and sensible criterion for which of the candidates the algorithm will pick. Our algorithm fulfills such a criterion: it will pick the unique candidate t' which is minimal with respect to the relation $\preceq$ which is first defined on types (fig. 7) and then lifted to terms (fig. 8). The reflexive, transitive and antisymmetric relation $\preceq$ expresses formally what we referred to earlier as "*$\mathscr{U}$ is preferred over any other type, and set-theoretic function application is preferred over higher-order function application*".

Fig. 7. Definition of $\preceq$ for Types

$$\frac{}{\mathbb{P} \preceq \mathbb{P}} \qquad \frac{\tau \text{ is a type}}{\mathscr{U} \preceq \tau} \qquad \frac{\tau_1 \preceq \omega_1 \text{ and } \tau_2 \preceq \omega_2}{\tau_1 \to \tau_2 \preceq \omega_1 \to \omega_2}$$

Fig. 8. Definition of $\preceq$ for Terms

$$\frac{}{c \preceq c} \qquad \frac{}{x \preceq x} \qquad \frac{\tau_1 \preceq \tau_2}{p[\tau_1] \preceq p[\tau_2]} \qquad \frac{t_1 \preceq t_2 \text{ and } \tau_1 \preceq \tau_2}{t_1 : \tau_1 \preceq t_2 : \tau_2}$$

$$\frac{\tau_1 \preceq \tau_2 \text{ and } t_1 \preceq t_2 \text{ and } \omega_1 \preceq \omega_2}{x : \tau_1 \mapsto t_1 : \omega_1 \preceq x : \tau_2 \mapsto t_2 : \omega_2}$$

$$\frac{t_1 \preceq s_1 \text{ and } t_2 \preceq s_2 \text{ and } \tau_1 \preceq \tau_2}{t_1 \diamond_{\mathsf{H}} t_2 : \tau_1 \preceq s_1 \diamond_{\mathsf{H}} s_2 : \tau_2}$$

$$\frac{t_1 \preceq s_1 \text{ and } t_2 \preceq s_2 \text{ and } \tau_1 \preceq \tau_2 \text{ and } \diamond \in \{\diamond_{\mathsf{H}}, \diamond_{\mathsf{ZF}}\}}{t_1 \diamond_{\mathsf{ZF}} t_2 : \tau_1 \preceq s_1 \diamond s_2 : \tau_2}$$

Lemma 3. *Let σ be a substitution and t a preterm. Then $\mathcal{E}(\sigma(t)) = \sigma(\mathcal{E}(t))$.*

Proof. Immediate from the definitions. □

Lemma 4. *Let t be a preterm such that $\mathcal{E}(t)$ is solvable. Then $\mathcal{E}(\mathcal{S}(t))$ is a set of identities.*

Proof. $\mathcal{E}(\mathcal{S}(t)) = \mathcal{E}(\mathsf{mgu}_{\mathcal{E}(t)}(t)) = \mathsf{mgu}_{\mathcal{E}(t)}(\mathcal{E}(t))$. □

Lemma 5. *Let t be a preterm such that $\mathcal{E}(t)$ is solvable and $\mathcal{S}(t) = t$. Then $\mathsf{mgu}_{\mathcal{E}(t)} = \mathsf{id}$ and $\mathcal{E}(t)$ is a set of identities.*

Proof. $\mathsf{id}(\mathcal{E}(t)) = \mathcal{E}(t) = \mathcal{E}(\mathcal{S}(t))$ □

Lemma 6. *Let s and t be preterms such that $s \sqsubseteq t$. Then $\mathcal{E}(t) \subseteq \mathcal{E}(s)$. In particular, if $\mathcal{E}(s)$ is solvable then so is $\mathcal{E}(t)$.*

Proof. Immediate from the definitions. □

Lemma 7. *If t is a preterm without any type variables then $\mathcal{D}(t)$ is a term.*

Proof. If τ_1 does not contain any type variables then either $\neg^{\mathscr{U}}(\tau_1)$ or $\neg^{\rightarrow}(\tau_1)$ is true, and therefore $\diamond(\tau_1, \tau_2, \tau_3) \in \{\diamond_{\mathsf{H}}, \diamond_{\mathsf{ZF}}\}$. □

Lemma 8. *If t is a preterm without any type variables, and t' is a term such that $\mathcal{E}(t')$ is solvable and $t' \sqsubseteq t$ then $t' = \mathcal{D}(t)$.*

Proof. The terms t' and $\mathcal{D}(t)$ could only possibly differ in places where t has an occurrence of $\diamond_?$. In those places, choosing differently from $\mathcal{D}$ would make the resulting equations unsolvable; however, $\mathcal{E}(t')$ is solvable. □

Lemma 9. *For any preterm t and any substitution σ*

$$\mathcal{D}(\sigma(t)) \sqsubseteq \sigma(\mathcal{D}(t)).$$

Proof. This follows from the fact that $\diamond(\tau_1, \tau_2, \tau_3) \in \{\diamond_{\mathsf{H}}, \diamond_{\mathsf{ZF}}\}$ implies

$$\diamond(\sigma(\tau_1), \sigma(\tau_2), \sigma(\tau_3)) = \diamond(\tau_1, \tau_2, \tau_3).$$

□

Lemma 10. *Let t be a preterm and t' a term such that $t' \leq t$ and $\mathcal{E}(t')$ is solvable. Then $\mathcal{E}(t)$ is solvable and both $t' \leq \mathcal{S}(t)$ and $t' \leq \mathcal{D}(t)$ hold.*

Proof. Because $t' \leq t$ there exist σ and t'' such that $t'' = \sigma(t)$ and $t' \sqsubseteq t''$. Because $\mathcal{E}(t')$ is solvable so is $\mathcal{E}(t'')$. Because t' is a term, neither t' nor t'' contain any type variables and thus $\mathcal{S}(t'') = t''$ which implies that $\mathcal{E}(t'') = \mathcal{E}(\sigma(t)) = \sigma(\mathcal{E}(t))$ are all sets of identities, and therefore σ is a unifier of $\mathcal{E}(t)$. This means there is a substitution δ such that $\sigma = \delta \circ \mathsf{mgu}_{\mathcal{E}(t)}$ which implies $t'' = \sigma(t) = \delta(\mathcal{S}(t))$. Thus $t' \leq \mathcal{S}(t)$. Furthermore,

$$t' = \mathcal{D}(t'') = \mathcal{D}(\sigma(t)) \sqsubseteq \sigma(\mathcal{D}(t)),$$

and thus $t' \leq \mathcal{D}(t)$. □

Lemma 11. TypeInfer *is sound. It is also complete in the sense that it will compute the unique $\preceq$-minimal solution if there is any solution at all.*

Proof. Given a preterm t, TypeInfer will check if $\mathcal{E}(t)$ is solvable.

If it is not, it will fail; this is correct, because then there can be no solution t' with $t' \leq t$ and $\mathcal{E}(t')$ solvable because otherwise $\mathcal{E}(t)$ would be solvable as well because of Lemma 10.

If on the other hand $\mathcal{E}(t)$ is solvable it will either recursively call itself with argument d where $d = \mathcal{D}(\mathcal{S}(t))$ or perform a final calculation and return the

result. In the case of a recursive call, we know because of Lemma 10 that every solution t' of t is also a solution of d.

So let us look at the final calculation now. We know that $d = s$ holds where $s = \mathcal{S}(t)$. In other words, d is a fixpoint of $\mathcal{D}$ which means that

$$\diamond(\tau_1, \tau_2, \tau_3) = \diamond_?$$

holds for all invocations of $\diamond$ during the computation of $\mathcal{D}(d)$ which implies that all of τ_1, τ_2 and τ_3 are either equal to $\mathscr{U}$ or equal to a type variable. The substitution $\mathcal{U}$ will therefore make all τ_i in those invocations equal to $\mathscr{U}$ and thus the effect of applying $\mathcal{D}$ to $\mathcal{U}(d)$ is to switch all occurrences of $\diamond_?$ to $\diamond_{\mathsf{ZF}}$. In particular, $\mathcal{E}(\mathcal{D}(\mathcal{U}(d)))$ is solvable because $\mathcal{E}(d)$ is a set of identities and

$$\mathcal{E}(\mathcal{D}(\mathcal{U}(d))) = \mathcal{U}(\mathcal{E}(d)) \cup \{\mathscr{U} \equiv \mathscr{U}\}.$$

That means that t_0 is a solution where $t_0 = \mathcal{D}(\mathcal{U}(d))$. Furthermore t_0 is minimal with respect to $\preceq$ because for any solution t' we know $t' \leq d$ and because for any term a and any preterm b such that $a \leq b$ it follows that $\mathcal{D}^{\mathsf{ZF}}(\mathcal{U}(b)) \preceq a$ where $\mathcal{D}^{\mathsf{ZF}}$ replaces all occurrences of $\diamond_?$ in its argument by $\diamond_{\mathsf{ZF}}$. Because of the antisymmetry of $\preceq$, minimality implies uniqueness. □

2.7 Examples

We present three more examples of applying TypeInfer. We will use abbreviated notations for preterms in the following.

Example 4. Let t be the preterm $\forall x : \alpha.\ \exists f : \beta.\ f \diamond_? x : \gamma$. Then

$$\mathcal{S}(t) = \forall x : \alpha.\ \exists f : \beta.\ f \diamond_? x : \mathbb{P}.$$

Because of $\diamond(\beta, \alpha, \mathbb{P}) = \diamond_{\mathsf{H}}$ we know

$$t' = \mathcal{D}(\mathcal{S}(t)) = \forall x : \alpha.\ \exists f : \beta.\ f \diamond_{\mathsf{H}} x : \mathbb{P} \neq \mathcal{S}(t)$$

Computing $\mathsf{TypeInfer}(t')$ yields first $\mathcal{S}(t') = \forall x : \alpha.\ \exists f : \alpha \to \mathbb{P}.\ f \diamond_{\mathsf{H}} x$ and then

$$\mathsf{TypeInfer}(t) = \mathsf{TypeInfer}(t') = \mathcal{U}(\mathcal{S}(t')) = \forall x : \mathscr{U}.\ \exists f : \mathscr{U} \to \mathbb{P}.\ f \diamond_{\mathsf{H}} x.$$

Example 5. Let t be $a : \alpha \mapsto b : \beta \mapsto c : \gamma \mapsto d : \delta \mapsto a \diamond_? b \diamond_? c \diamond_? d$. Then

$$t' = \mathcal{D}(\mathcal{S}(t)) = a : \alpha \mapsto b : \beta \mapsto c : \gamma \mapsto d : \delta \mapsto a \diamond_? b \diamond_? c \diamond_? d$$
$$\mathsf{TypeInfer}(t) = \mathcal{D}(\mathcal{U}(t')) = a : \mathscr{U} \mapsto b : \mathscr{U} \mapsto c : \mathscr{U} \mapsto d : \mathscr{U} \mapsto a \diamond_{\mathsf{ZF}} b \diamond_{\mathsf{ZF}} c \diamond_{\mathsf{ZF}} d$$

Example 6. Let us modify the previous example and infer the type of

$$a : \alpha \mapsto b : \beta \mapsto c : \gamma \mapsto d : \delta \mapsto a \diamond_? b \diamond_? c \diamond_? d \wedge d.$$

This time the algorithm needs three recursive calls and yields finally

$$a : \mathscr{U} \to \mathscr{U} \to \mathbb{P} \to \mathbb{P} \mapsto b : \mathscr{U} \mapsto c : \mathscr{U} \mapsto d : \mathbb{P} \mapsto a \diamond_{\mathsf{H}} b \diamond_{\mathsf{H}} c \diamond_{\mathsf{H}} d \wedge d$$

This example can be generalized to produce for any n an example with n occurrences of $\diamond_?$ such that TypeInfer needs n recursive calls.

3 Related Work

In HOL-ST [5], set-theoretic and higher-order function application have different syntax; in particular, higher-order function application is written fx and set-theoretic function application is denoted by $f \diamond x$. Because HOL-ST has type variables and capabilities for defining new types, the type $\mathscr{U}$ is just one type besides many others; our type inference algorithm does not yield a desirable result in such a setting. Of course, as HOL-ST is a strict superset of ZFH, one could work in it as one works in ZFH; our type inference algorithm can be directly translated to HOL-ST to support such a scenario.

Isabelle/ZF [7] also uses two different notations, fx for higher-order and $f`x$ for set-theoretic function application. Although Isabelle/ZF is embedded in polymorphic intuitionistic higher-order logic it is used in an essentially monomorphic way using an identical type system to ZFH. Isabelle has a flexible mechanism for syntax extension by adding context-free grammar rules so it should be possible to introduce syntax to write set-theoretic function application via juxtaposition as well. Type information is used in Isabelle to disambiguate between several possible parse trees. Using this built-in mechanism would lead to a situation similar to what we described in Section 2.3: whenever there are multiple possible typings parsing would fail. But in principle it should be possible to write a system-level Isabelle extension which implements our type inference algorithm for Isabelle/ZF.

Our operator for set-theoretic function application $\mathsf{apply} : \mathscr{U} \to \mathscr{U} \to \mathscr{U}$ could be viewed as a coercion from $\mathscr{U}$ to $\mathscr{U} \to \mathscr{U}$. There has been previous work with regard to the general problem of extending Hindley-Milner type inference in the presence of coercions. In [12] coercions between types which only differ in their base types but not in their type constructors are considered; because $\mathscr{U}$ does not contain the type constructor $\to$ but $\mathscr{U} \to \mathscr{U}$ does, their work is not applicable to our case. In [13] more general coercions are considered but their algorithm has the property that no coercions are inserted if Hindley-Milner type inference alone already yields a valid typing; this is not what we would like in our setting as this property means that their algorithm would choose the typing $f : \alpha \to \alpha$ and $x : \alpha$ over the typing $f : \mathscr{U}$ and $x : \mathscr{U}$ in our introductory example. And then there would still be the question of how that polymorphic type should be converted into a monomorphic one.

Another way of looking at our scenario is from an overloading point of view where the generic operator $\diamond_?$ of type $\alpha \to \beta \to \gamma$ has two different instances $\diamond_{\mathsf{ZF}} : \mathscr{U} \to \mathscr{U} \to \mathscr{U}$ and $\diamond_{\mathsf{H}} : (\alpha \to \beta) \to \alpha \to \beta$. But typical algorithms which extend Hindley-Milner to take overloading into account like in [14] compute a principal type of which all other possible valid typings are instances. This is not what our algorithm does; instead we minimize a preference relation $\preceq$ which is different from the is-an-instance-of relation a principal type maximizes.

4 Conclusion

We have implemented TypeInfer as part of the implementation of ProofScript, the proof language of ProofPeer. Combining the strengths of set theory with the strengths of higher-order logic has always had a certain appeal to ITP researchers. We believe that the answer has been staring into our faces for quite some time now in the form of ZFH; all we had to do to arrive at ZFH was to take HOL-ST and take away powers which HOL practitioneers take for granted but which are of little use in the context of set theory. The existence of TypeInfer which allows us to fuse the notations for higher-order function application and set-theory function application into a single one *because of the absence of those powers* supports our belief.

References

1. ProofPeer, `http://www.proofpeer.net`
2. Steven Obua, Jacques Fleuriot, Phil Scott, David Aspinall: ProofPeer: Collaborative Theorem Proving, `http://arxiv.org/abs/1404.6186`
3. Thomas Hales et al.: A formal proof of the Kepler conjecture, `http://arxiv.org/abs/1501.02155`
4. Homotopy Type Theory, `http://homotopytypetheory.org/`
5. Sten Agerholm, Mike Gordon: Experiments with ZF Set Theory in HOL and Isabelle. In: Higher Order Logic Theorem Proving and Its Applications, 8th International Workshop, Aspen Grove, USA. LNCS 971, Springer (1995)
6. Mike Gordon: Set Theory, Higher Order Logic or Both? TPHOLs 1996, Springer
7. Lawrence C. Paulson: Set Theory for Verification: I. From Foundations to Functions. Journal of Automated Reasoning, Volume 11, Issue 3, Springer (1993)
8. Alexander Krauss: Partial and Nested Recursive Function Definitions in Higher-order Logic. Journal of Automated Reasoning, Volume 44, Issue 4, Springer (2010)
9. ProofPeer Root Theory, `http://proofpeer.net/repository?root.thy`
10. Franz Baader, Tobias Nipkow: Term Rewriting and All That. Cambridge University Press (1999)
11. Robin Milner: A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences 17, 348-375 (1978)
12. Dmitriy Traytel, Stefan Berghofer, Tobias Nipkow: Extending Hindley-Milner Type Inference with Coercive Structural Subtyping. APLAS 2011, LNCS 7078, Springer
13. Zhaohui Luo: Coercions in a polymorphic type system. Mathematical Structures in Computer Science, Volume 18, Special Issue 04, Cambridge Journals 2008
14. Martin Odersky, Philip Wadler, Martin Wehr: A Second Look at Overloading. Proceedings of the seventh international conference on Functional programming languages and computer architecture, ACM 1995