ProofScript: Proof Scripting for the Masses

Citation for published version:
Obua, S, Scott, P & Fleuriot, J 2016, ProofScript: Proof Scripting for the Masses. in Theoretical Aspects of Computing - ICTAC 2016: 13th International Colloquium, Taipei, Taiwan, ROC, October 24–31, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9965, Springer International Publishing, pp. 333-348, 13th International Colloquium on Theoretical Aspects of Computing, Taipei, Taiwan, Province of China, 24/10/16. https://doi.org/10.1007/978-3-319-46750-4_19

Digital Object Identifier (DOI):
10.1007/978-3-319-46750-4_19

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Theoretical Aspects of Computing - ICTAC 2016

ProofScript: Proof Scripting for the Masses

Steven Obua, Phil Scott, and Jacques Fleuriot

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

Abstract. The goal of the *ProofPeer* project is to make *collaborative theorem proving* a reality. An important part of our plan to make this happen is *ProofScript*, a language designed to be the main user interface of ProofPeer. Of foremost importance in the design of ProofScript is its fit within a collaborative theorem proving environment. By this we mean that it needs to fit into an environment where peers who are not necessarily part of the current theorem proving and programming language communities work independently from but collaboratively with each other to produce formal definitions and proofs. All aspects of ProofScript are shaped by this design principle. In this paper we will discuss ProofScript's most important aspect of being an integrated language both for interactive proof and for proof scripting.

1 Introduction

Interactive theorem proving (ITP) has come a long way since its inception in the seventies. We have argued elsewhere [2] that *collaborative theorem proving* (CTP) represents its natural evolution, where we defined CTP as the social machine of ITP. The goal of the *ProofPeer* [1] project is to make CTP a reality both by developing CTP fundamentals and by building a practical CTP system.

An important component of ProofPeer is the language that peers use to formulate theorems and proofs, which we call *ProofScript*. In many respects our role model and arguably the state of the art for a structured proof language is Isabelle/Isar [6]. The Isabelle/Isar system consists of a complicated stack in which the programming language Standard ML powers the Isabelle kernel and its programmatic extensions. Isar is the most important of these extensions. That means that in order to add automation and scripting to Isar, one needs to be familiar with the low-level ML fundamentals on which Isar itself is built. In acknowledgment of this, limited capabilities for proof automation called *Eisbach* have recently been added to Isar so that proof methods can be formulated within the Isar language itself [9]. Apart from the fact that this only makes possible proof automation of limited scope, we think that the general situation has become even more complex by this addition, as depicted in Fig. 1.

Some say that complexity crushes the human mind, others say that one should design things as simple as possible, but not simpler. Not wanting to ignore any of these truths, with ProofScript we are trying to condense the capabilities,

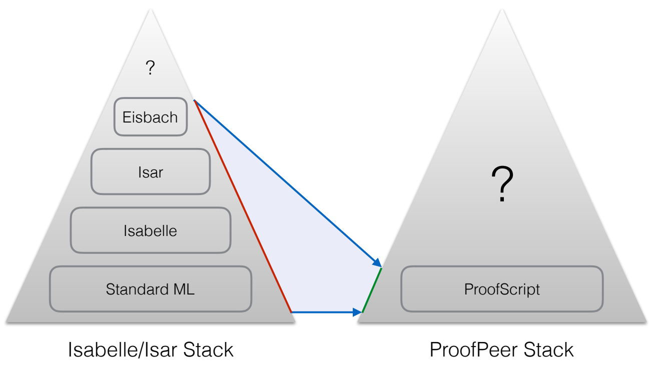

Fig. 1. Surface complexity of Isabelle/Isar vs. ProofPeer

ideas, and experience embodied by the Isabelle/Isar stack into a single language which serves us as a fresh starting point for exploring collaborative theorem proving. This is not unsimilar to how ITP started out, when ML was designed as a language to explore the design space of ITP [10] before extension pyramids like Isabelle/Isar were built on top of it.

Our goal is to minimize the *surface complexity* the user of our system perceives, thus opening up the system to a wide audience of mathematicians, engineers, basically anyone in need of designing correct virtual or physical artifacts. We do not expect proofpeers to be familiar with type theory or to possess other knowledge usually common only in the programming language community. ProofScript provides us with a unified layer of abstraction which we hope is easy to learn and become productive in. Picking up ProofScript should not be harder than, for example, learning Javascript. Having this abstraction layer also stabilizes ProofPeer as an environment, as changes to how theories are stored, how they are interpreted/compiled, how their execution is distributed in our cloud computing environment etc. can to a large extent happen under the hood.

In this paper we describe the current state of the ProofScript language. Although capability wise we are nowhere near yet what the whole Isabelle/Isar stack can do, we have made important steps towards this goal and we think that the promise of our approach is now apparent.

We will first give an overview of the programming language aspects of ProofScript in Section 2. After describing the logical foundations of ProofScript in Section 3, we describe in Section 4 how we integrated them with the programming language into an approach we call *structured proof scripting*. We conclude in Section 5.

For further information about ProofScript, and to experiment with our current implementation, please consult the ProofScript documentation [3].

2 The Programming Language

In this section we give an overview of the ProofScript programming language, with a focus on *why* we designed the language as we did.

2.1 Theories and Namespaces

ProofScript is written in units called *theories*. All theories except the `root` theory [8] have one or more parent theories which they extend, thus forming a directed, acyclic and connected theory extension graph. Before a theory is executed, all of its parents must have been executed. The result of executing a theory is basically a binding of names to computed values, which is usually persisted after execution.

Each theory has its own unique *namespace*. Like Java packages, namespaces are organized as a directory-like structure. We use them to limit both name conflicts in theories and user access privileges to those theories. The namespace tree is orthogonal to the extension graph, and thus one way for peers to collaborate with each other is by extending each other's theories. Within a theory, namespace *aliases* can be defined.

2.2 Types and Patterns

ProofScript is dynamically typed. This means types are checked during the execution of a theory, and not statically, i.e. at some time between the writing and the execution of the theory. This choice is somewhat unorthodox, given that we have drawn earlier parallels with Standard ML, which is famous for the invention of static type checking via the Hindley-Milner type system. The main reason for our choice is that a static type system that approximates the flexibility of dynamical typing would need to include advanced concepts like functors, type classes, and so on. This would presume a depth of programming knowledge which is simply not realistic for our intended target audience of proofpeers. To further strengthen our argument, languages like Python and Julia have shown that dynamic typing is well-received in the scientific community.

Another more subtle reason is that ProofScript's logic is based on set theory, which in some sense is closer to dynamic typing than static typing. Of course there is no technical reason why there should be any affinity between scripting language and logic, but from an anthropological point of view there seems to be an advantage in fostering these affinities.

Unlike set theory though, being a vehicle for theorem proving, ProofScript has strong types. In particular this means that values of type `Theorem` can only be created by truth preserving operations sanctioned by the logical kernel. Similarly, any other value obeys the invariants established by its type.

There are four built-in logic related types: `Context`, `Theorem`, `Term` and `Type`. These are covered in Section 3. Furthermore, there are currently eight built-in non-logical types, shown in Figure 2.

Type	Example Value
`Nil`	**nil**
`Integer`	`-18446744073709551617`
`String`	`"ProofScript"`
`Boolean`	`false`
`Tuple`	`(7, "seven", ("two", 2))`
`Set`	`{2, 3, 5, 7, "prime"}`
`Map`	`{7 → "seven", 3 → "three", "four" → 4}`
`Function`	`x ↦ x * x`

Fig. 2. Built-in non-logical types of ProofScript

ProofScript has pattern matching. The matching for logical terms is treated in Section 4. Examples for other patterns are shown in Figure 3. As shown in the examples, patterns can be used to perform simple type checks.

Pattern Example	Explanation
`x : Integer`	matches any integer
`(x, _, y)` **if** `x == y`	matches any triple that has equal first and third elements
`x <+ _ +> y` **if** `x == y`	matches any tuple with at least two elements where the first and the last element are equal
`None`	matches a value equal to the custom constructor `None`
`Some x`	matches any value created by applying the custom constructor `Some`
`f x`	matches any value which is a constructor application
`_ : Option`	matches any value that has the custom type `Option`

Fig. 3. Examples of non-logical patterns

It is possible in ProofScript to define *custom datatypes*. An example is shown in Figure 4 (left hand side) where the custom type `Option` is defined, together

```
datatype Option
  None
  Some x
```

```
datatype List
  Nil
  Cons (head, tail : List)
```

Fig. 4. Custom types `Option` and `List`

with two constructors `None` and `Some`. `Some` takes an argument and `None` does not. An example for an expression yielding a value of type `Option` is `Some (1, None)`. Examples for using pattern matching with this custom type are shown in Figure 3. Note that type and constructor names must start with an uppercase letter. Mostly this is just a convention we like to enforce, but for constructor names it also makes it easier to distinguish in patterns between constructors and variables.

The argument of a datatype constructor can be constrained by an arbitrary pattern. A datatype modelling heterogeneous lists could therefore be defined as shown in Figure 4 (right hand side). While expressions like `Cons(1, Cons(2, Cons (3, Nil)))` would evaluate successfully to a value of type `List`, an expression like `Cons(1, 2)` would lead to a runtime error.

2.3 Purely Functional Structured Programming

ProofScript is a purely functional programming language with strict evaluation. By this we mean that functions are first-class, there are no side-effects, and arguments are evaluated *before* they are passed to a function. This makes it possible for peers to write concise, expressive, modular, yet predictable code.

The absence of side-effects, besides its other obvious advantages, also immensely simplifies the semantics and economy of how theories are managed and shared. Imagine theory `A` being extended by n theories $\mathtt{T}_1, \ldots, \mathtt{T}_n$. In ProofPeer, after executing a theory, its resulting state is persisted and reused, immediately or at a later time. In our example, that would mean that after executing all of the aforementioned theories, $n + 1$ theory states would have been persisted. If instead `A` were to contain a function which depended on and mutated some state, then calling this function from $\mathtt{T}_i$ could lead to a changed state of `A` which then would need separate persisting. Assuming none of the $\mathtt{T}_i$ extend each other, after executing all theories, up to $2n+1$ theory states would have been persisted. Now assume that the $\mathtt{T}_i$ contained mutable state themselves and depended on each other, say $\mathtt{T}_j$ also extends $\mathtt{T}_i$ for all $i < j$. In the worst possible case, that would lead to $\frac{(n+1)(n+2)}{2}$ persisted theory states. Our intention is to develop technologies for collaborative theorem proving which scale, so avoiding such quadratic growth by enforcing the absence of side-effects is an obvious thing to do.

Despite its advantages, purely functional programming is not mainstream. Imperative programming is a much more popular style to program in. A major reason for this is that structured programming, which is one of the pillars on which imperative programming rests, is just much more readable to most people than a program written by composing higher-order functions.

We want the advantages of purely functional programming, but we also want to appeal to the mainstream. Fortunately, purely functional programming and structured programming are not at odds with each other at all, but can easily be combined in an approach called *purely functional structured programming* [11]. We have adopted this approach for ProofScript, and therefore it is possible to program in ProofScript *both* by composing functions *and* by writing block-structured code, including **for**-loops, **while**-loops, etc.

As a simple example, consider computing the greatest common divisor of two integers in ProofScript as shown in Figure 5. On the left hand side, typical functional code for computing the `gcd` is presented. The right hand side displays an alternative formulation of `gcd` using structured programming. Both styles fit within the paradigm of purely functional structured programming, and both versions of `gcd` are side-effect free.

```
def gcd (a, b) =
  if a < 0 then gcd (-a, b)
  else
    if b < 0 then gcd (a, -b)
    else
      if b == 0 then a
      else gcd (b, a mod b)
```

```
def gcd (a, b) =
  if a < 0 then a = -a
  if b < 0 then b = -b
  while b > 0 do
    (a, b) = (b, a mod b)
  return a
```

Fig. 5. Greatest common divisor in ProofScript

2.4 Layout-sensitive Syntax

Rules that endow indentation and layout with meaning have been adopted by a diverse range of computer languages, from programming languages like Python, Haskell or Scala to markup languages like Markdown. It seems that having explicit rules for layout is especially helpful for novices [12] by removing clutter in the form of curly braces, semicolons, and the like, making errors more obvious by making intended, indented and actual structure synonymous.

That is why we have created ProofScript from the start as a language where indentation and layout is meaningful. ProofScript's syntax is defined by a context-free grammar with added explicit annotations which constrain the possible shapes of parsed text. The technical details of this have been developed over the past three years and are actually still evolving. It was our hope from the beginning that semantic layout would not only make it easier directly for the users themselves to read and write code, but also that the error recovery mechanisms of the parser would profit from it – this is important for the interactive experience we are striving for. As it turns out, this is indeed the case, and has lead us to the discovery of a general way of combining lexing and parsing which we call *local lexing* [4] and which will be described in a forthcoming separate paper.

The main use of indentation in ProofScript is to delineate the *block structure* of code, making it an ideal companion of our paradigm of purely functional structured programming. An example is the use of indentation to resolve the *dangling else* conflict, as might already be apparent from Figure 5.

It is often argued that while semantic layout might be beneficial for novices, in a professional setting meaningful layout is detrimental to productivity. A simple example is that usually text editors can be configured to equate a tab character with a fixed number of space characters, often this number is 2 or 4. Differing editor configurations can thus lead to differing meanings of the same program code, which is clearly undesirable. Our solution to this particular problem is to simply disallow the tab character in legal ProofScript. Additionally, we hope to avoid this problem entirely by letting peers interact with ProofPeer in a web environment under our control.

Another example is that of the naive use of Landin's *offside rule* [13] to associate indentation with meaning, which leads to code that might easily break or change its meaning when common automatic refactorings like changing the name of an identifier are applied to it. The offside rule states that *"The southeast quadrant that just contains the phrase's first symbol must contain the entire phrase [...]."* If we apply this rule to the hypothetical code snippets in Figure 6, then it would make sense for the left hand side to evaluate to `[(11, 10000)]` and for the right hand side to yield `[(10000, [101])]` (the **do*** control flow statement evaluates all expressions in its argument block and returns them as a tuple). But the right hand side is just a refactoring of the left hand side where we replaced `f` with `somefunction`! To combat this problem, neither of the two code snippets shown in Figure 6 are valid ProofScript. We achieve this by only using layout rules which only compare distances made up entirely of spaces, as opposed to comparing the lengths of texts made up of arbitrary characters. For example, we cannot compare the length (in pixels or centimeters) of the text "**def** `f x =` `do*`" which consists of 13 characters with the length of 13 spaces. This also solves related problems, such as the fact that a different variable-width font could possibly change the meaning of a program. The legal ProofScript versions of the code shown in Figure 6 are presented in Figure 7. Clearly, their meaning is stable under the refactorings `f` ↦ `somefunction` and `somefunction` ↦ `f`, respectively.

```
do*
  val x = 10
  def f x = do* x + 1
                x = x * x
                x * x
  f x
```

```
do*
  val x = 10
  def somefunction x = do* x + 1
                x = x * x
                x * x
  somefunction x
```

Fig. 6. Instabilities in Landin's offside rule under refactorings

```
do*
  val x = 10
  def f x =
    do*
      x + 1
      x = x * x
      x * x
  f x
```

```
do*
  val x = 10
  def somefunction x = do* x + 1
  x = x * x
  x * x
  somefunction x
```

Fig. 7. Corrected versions of the invalid code in Figure 6

3 Logical Foundations

Most people who use math do not work in a formal setting. They have been taught a naive form of set theory, and usually they know that more rigorous and (hopefully) paradox-free versions exist which are very similar to naive set theory, like Zermelo Fraenkel set theory (ZF).

To accommodate all of these people, ProofScript's logic is based on Zermelo Fraenkel set theory. Isabelle/ZF [14,15] has pioneered how to embed Zermelo Fraenkel set theory with choice (ZFC) in *intuitionistic* higher-order logic. Our approach differs from the Isabelle/ZF approach in that we embed ZF in *classical* higher-order logic. We call this logic *ZFH* [17] (the H stands both for higher-order logic and Hilbert choice). The reason for this is that we want to be able to draw on the wealth of experience and tools which have been developed in the realm of classical higher-order logic (HOL).

ZFH is almost identical to the logic of HOL-ST [7] and Isabelle/HOLZF [16], which embed ZFC within HOL via a special type representing the universe of ZF sets that comes with the constants and axioms which make up ZF (note that the axiom of choice is implied by the properties of the Hilbert choice operator in HOL). A dilemma that comes up in both HOL-ST and Isabelle/HOLZF is that it is often not clear how to choose between ZF and HOL. Natural numbers for example could be formalised as a ZF set, but they could also be defined as a type in HOL. We resolve this dilemma in ZFH by not having any facilities for defining new HOL types, and by disallowing type variables in terms. This restriction makes it clear that HOL is to be used as the "meta logic", whereas ZF is the logic where all the real work gets done. Because ZFH is a restricted version of HOL-ST and Isabelle/HOLZF, it is consistent if they are.

In the following we will explain the various parts that constitute ZFH and how ProofScript's logical kernel manages them.

3.1 Types and Terms

A *type* τ is either the universal type of ZF sets $\mathscr{U}$, the propositional/boolean type $\mathbb{P}$, or a function type $\tau_1 \to \tau_2$:

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

A *term* t is either a constant c, a polymorphic constant $p[\tau]$, a higher-order function $x : \tau_1 \mapsto t$, a bound variable x or a higher-order application $t_1\, t_2$:

$$t \ ::= c \mid p[\tau] \mid x : \tau \mapsto t \mid x \mid t_1\, t_2.$$

We have chosen the notation $x : \tau \mapsto t$ over the notation $\lambda x : \tau.\ t$ because the former is more familiar to a wider audience than the latter.

There are only three polymorphic constants p: equality $=$, universal quantification $\forall$ and existential quantification $\exists$. All other constants are monomorphic constants c; this means in particular that all user-defined constants are monomorphic.

Terms on their own do not have any type because we do not know the types of the monomorphic constants c which appear in a term. The type of a term can only be determined relative to a *context* $\mathcal{C}$. Contexts are the topic of Section 3.3. For our purposes here we can simply view a context $\mathcal{C}$ as a partial function from constants c to types $\mathcal{C}(c)$. We can then define the type $\Gamma_{\mathcal{C}}(t)$ of a term t relative to a context $\mathcal{C}$ as shown in Figure 8. $\Gamma_{\mathcal{C}}$ is a partial function, and we call t *valid*

$$\begin{aligned}
\Gamma_{\mathcal{C}}(t) &= \Gamma_{\mathcal{C},\emptyset} \\
\Gamma_{\mathcal{C},\mathcal{V}}(c) &= \mathcal{C}(c) \quad \text{if } \mathcal{C} \text{ is defined at } c \\
\Gamma_{\mathcal{C},\mathcal{V}}(x) &= \mathcal{V}(x) \quad \text{if } \mathcal{V} \text{ is defined at } x \\
\Gamma_{\mathcal{C},\mathcal{V}}(p[\tau]) &= \begin{cases} (\tau \to \mathbb{P}) \to \mathbb{P} & \text{if } p \in \{\forall, \exists\} \\ \tau \to \tau \to \mathbb{P} & \text{if } p \in \{=\} \end{cases} \\
\Gamma_{\mathcal{C},\mathcal{V}}(x : \tau \mapsto t) &= \tau \to \rho \quad \text{if } \Gamma_{\mathcal{C},\mathcal{V}[x:=\tau]} = \rho \\
\Gamma_{\mathcal{C},\mathcal{V}}(t_1\, t_2) &= \rho \quad \text{if } \Gamma_{\mathcal{C},\mathcal{V}}(t_1) = \tau \to \rho \text{ and } \Gamma_{\mathcal{C},\mathcal{V}}(t_2) = \tau
\end{aligned}$$

Fig. 8. The type $\Gamma_{\mathcal{C}}(t)$ of a term t relative to a context $\mathcal{C}$

(in $\mathcal{C}$) if $\Gamma_{\mathcal{C}}$ is defined at t, i.e. if $\Gamma_{\mathcal{C}}(t) = \tau$ for some type τ. Note that a valid term has *no free variables*: in valid ZFH terms, all variables appearing in the term must be bound to the argument of an enclosing higher-order function.

Because a term without a context is usually useless, the kernel only operates on *certified terms*, which are basically pairs $(\mathcal{C}, t)$ of a context $\mathcal{C}$ and a term t such that t is valid in $\mathcal{C}$.

3.2 Type Inference

Of course when actually writing down concrete terms in ProofScript, most of the time there is no need to explicitly provide the type τ in the terms $p[\tau]$ and $x : \tau \mapsto t$ as ProofScript performs fully automatic type inference in the spirit of Hindley-Milner. This can be done by allowing type variables for the purposes of the internal type inference algorithm only. There is a caveat though: given that there are no type variables in ZFH terms, what do we do if the result of the type inference still contains type variables?

A simple solution to this problem is to replace all type variables simply by $\mathscr{U}$, the universe of ZF sets. This makes sense as our focus is on set theory anyway, so among the infinitely many possible instantiations this is the most likely one.

This is *almost* the solution we chose; our actual solution is slightly more involved but allows the overloading of syntactic function application with both higher-order function application and set theoretic function application. This is described in detail in [17]. The resulting type inference can be described as being basically Hindley-Milner, but preferring set-theoretic function application over higher-order function application and the type $\mathscr{U}$ over all other types.

3.3 Contexts and Theorems

We have already pointed out that in ProofScript terms only make sense relative to a context $\mathcal{C}$. The context is responsible for maintaining a record of which constants have been defined or introduced so far, and which axioms have been assumed. Contexts in ProofScript are not derived constructs but axiomatic and built-in, and the kernel is responsible for their creation and maintenance. The specification for the concrete syntax of terms is also associated with contexts but not maintained by the kernel.

A context can be thought of as unifying two concepts which in other HOL systems are separate: that of the global logical state of the kernel (or theory context in a system that supports theories), and that of the local logical context.

At the start of a theory, a new context is created based on the contexts of all theories that the theory extends. From then on, there are basically four kernel operations to construct new contexts from existing ones: Introduce, Assume, Define and Choose. Each of them creates a new context of the kind indicated by the operation, and each of these new contexts maintains a backpointer to the context it was created from, its *parent context*. During the execution of a ProofScript theory, a tree of contexts is created, the root being the context created at the start of the theory.

Theorems are basically certified terms $(\mathcal{C}, t)$ such that $\Gamma_{\mathcal{C}}(t) = \mathbb{P}$ and such that t has been proven to represent a true proposition in context $\mathcal{C}$. There are three different ways to create theorems: 1) Theorems can be created via built-in functions. Examples are `reflexive`, which returns theorems of the form $(\mathcal{C}, t = t)$, or the function `instantiate`, which allows the instantiation of (some of) the universally quantified variables of an existing theorem. 2) Theorems can be created by lifting them between contexts. This is described later. 3) Theorems are created as byproducts of constructing new contexts. This is described in the following where we discuss the four basic kernel operations for constructing new contexts.

All of the following kernel operations are being applied to an existing context $\mathcal{C}$. The newly created context $\mathcal{D}$ has $\mathcal{C}$ as its parent; the context $\mathcal{D}$ also stores which operation using what parameters created it. Here are the operations:

Introduce(n, τ) takes a name n and a type τ and creates a new context $\mathcal{D}$ from $\mathcal{C}$ with an additional constant d with name n and of type τ in $\mathcal{D}$. The name n is not allowed to belong to any constant c in $\mathcal{C}$.

Assume(t) takes a certified term t of type $\mathbb{P}$ and autolifts it into context $\mathcal{C}$ (see Section 3.5), resulting in the certified term $(\mathcal{C}, t')$. It then creates a new context $\mathcal{D}$ from $\mathcal{C}$ and returns the theorem $(\mathcal{D}, t')$.

Define(n, t) takes a name n and a certified term t. The name n is not allowed to belong to any constant in $\mathcal{C}$. It first autolifts t into context $\mathcal{C}$, resulting in the certified term $(\mathcal{C}, t')$. After creating a new context $\mathcal{D}$ from $\mathcal{C}$ with an additional constant d with name n and of type $\Gamma_{\mathcal{C}}(t')$, it returns the theorem $(\mathcal{D}, d = t')$.

Choose(n, t) takes a name n and a theorem t. The name n is not allowed to belong to any constant in $\mathcal{C}$, and the theorem must have the form

$$(\mathcal{C}, \forall x_1 : \tau_1. \ldots \forall x_k : \tau_k. \exists y : \tau. t') \quad \text{for some } k \geq 0.$$

We assume that all x_i are different from each other (otherwise we would enforce this via automatic α-conversion).
The operation first creates the context $\mathcal{D}$ from $\mathcal{C}$ with an additional constant d with name n and of type $\tau_1 \to \ldots \to \tau_k \to \tau$ and then returns the theorem

$$(\mathcal{D}, \forall x_1 : \tau_1. \ldots \forall x_k : \tau_k. t'[d\, x_1 \ldots x_k/y]).$$

3.4 Theories and Namespaces

When a theory has been executed, one of the leaves of its context tree becomes the *completed context* of that theory. In principle, it would be possible to select any of the leaves *after* the creation of the complete context tree, but instead we have chosen that during the creation of the context tree of a theory it is always clear which one of the current leaves of the tree is the one which will lead to the completed context. The sequence of these designated leaves forms the *main context thread* of the theory, which at the end of execution will be the same as the path from the context at the start of the theory to the final completed context. The kernel manages the thread by maintaining for each executing theory a pointer to the current leaf of the main context thread; each time a new context is created whose parent is on the main context thread (the only exception is SpawnThread, introduced below), it asks the kernel to put it on the main context thread as well. The kernel will oblige if the pointer it maintains points to the parent of the new context, and will adjust it to point now to the new context instead; otherwise the creation of the new context will fail. Figure 9 illustrates the main context thread. Note that there are two additional kinds of contexts for managing the main context thread:

Complete() demands that context $\mathcal{C}$ is on the main context thread and creates a new context $\mathcal{D}$ on it which has all unqualified constants removed from it and which is not allowed to have any children.

SpawnThread() demands that context $\mathcal{C}$ is on the main context thread and creates a new context $\mathcal{D}$ which is *not* on the main context thread.

The names of logical constants may include a namespace qualification. Only constants which are introduced on the main context thread may be qualified; the namespace they are qualified with is of course the namespace of the theory the main context thread belongs to. Unqualified constants are treated as *private* constants which cannot be referred to by extending theories. This is reflected in how the context at the start of the theory is formed: it contains all and only the constants of the completed contexts of the theories it extends – and these constants are all qualified.

Note that it is not allowed to have an Assume context on the main context thread of any theory except theory `root` which introduces all axioms of ZFH. This means that all theories are conservative extensions of the `root` theory.

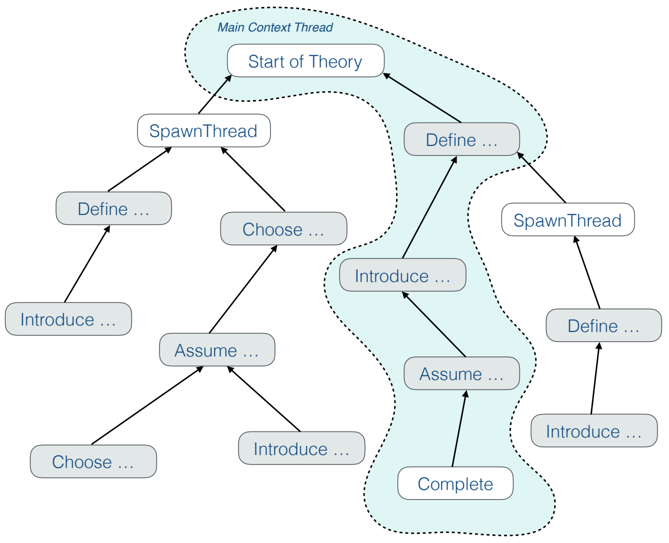

Fig. 9. Context tree and main context thread

3.5 Lifting between Contexts

We will consider in this section how the kernel lifts theorems and certified terms between contexts.

Let us first introduce some shorthand notation. For two theories A and B, let us write $B \rhd A$ if B directly or transitively extends A. Furthermore, for any context $\mathcal{C}$ let us write $\mathsf{theory}(\mathcal{C})$ for the theory that $\mathcal{C}$ belongs to. Finally, for two contexts $\mathcal{C}$ and $\mathcal{D}$ we write $\mathcal{D} \succ \mathcal{C}$ if $\mathsf{theory}(\mathcal{C}) = \mathsf{theory}(\mathcal{D})$ and $\mathcal{C}$ is a direct or transitive parent of $\mathcal{D}$.

Consider a theorem or certified term $(\mathcal{D}, t)$ which we wish to lift into context $\mathcal{C}$. In practice, there are only two relevant situations, and the kernel only supports these: we have either $\mathsf{theory}(\mathcal{C}) \rhd \mathsf{theory}(\mathcal{D})$ or $\mathsf{theory}(\mathcal{C}) = \mathsf{theory}(\mathcal{D})$.

We will first consider the situation where $\mathsf{theory}(\mathcal{C}) = \mathsf{theory}(\mathcal{D})$. If $\mathcal{C} = \mathcal{D}$ or $\mathcal{C} \succ \mathcal{D}$ then there is nothing to do and the result of the lift is simply the theorem / certified term $(\mathcal{C}, t)$ – an exception arises when $\mathcal{C}$ is the completed context, in which case t may only contain *qualified* constants. We will defer the case $\mathcal{D} \succ \mathcal{C}$ until later. If none of these cases hold, we find the unique context $\mathcal{C}'$ in the context tree of the theory such that both $\mathcal{D} \succ \mathcal{C}'$ and $\mathcal{C} \succ \mathcal{C}'$, and such that for any other $\mathcal{C}''$ with $\mathcal{D} \succ \mathcal{C}''$ and $\mathcal{C} \succ \mathcal{C}''$ we have $\mathcal{C}' \succ \mathcal{C}''$. We can now perform the lift by first lifting $(\mathcal{D}, t)$ to $(\mathcal{C}', t')$ and then lifting $(\mathcal{C}', t')$ to $(\mathcal{C}, t')$.

Secondly, assume we find ourselves in the situation $\mathsf{theory}(\mathcal{C}) \rhd \mathsf{theory}(\mathcal{D})$. We then find the completed context $\mathcal{D}'$ of $\mathsf{theory}(\mathcal{D})$ and lift $(\mathcal{D}, t)$ to $(\mathcal{D}', t')$. If t' would contain unqualified constants, which are considered private in $\mathsf{theory}(\mathcal{D})$, the lifting to $\mathcal{D}'$ fails. Otherwise the full lifting yields $(\mathcal{C}, t')$.

Note that for lifting to work as described it is crucial that a name introduced via a context is different from all names in the parent context so that no "constant capture" can occur when lifting downwards the context tree.

Lifting Upwards We will now consider the deferred case $\mathcal{D} \succ \mathcal{C}$. In particular we consider only the special case where $\mathcal{C}$ is the direct parent of $\mathcal{D}$ – the general case is derived from this by lifting along a successive sequence of parents leading from $\mathcal{D}$ to $\mathcal{C}$. We furthermore distinguish by the kind of the context $\mathcal{D}$ and whether we are lifting a theorem or a certified term, and whether the lifting is done in a *canonical* or a *structure preserving* way. Even without the cases for the context kinds SpawnThread and Complete (which are trivial) this leaves us with a total of 16 cases. Therefore we treat here only the cases for Introduce.

So assume that we want to lift a theorem $(\mathcal{D}, t)$ from an $\mathsf{Introduce}(n, \tau)$ context to its parent in a structure preserving way to obtain a theorem $(\mathcal{C}, t')$. Let c be the constant with name n that has been introduced by the context. Then we set $t' = \forall x : \tau. \, t[x/c]$ where x is a fresh variable not occuring as a bound variable anywhere in t. If instead we lift the canonical way, we arrive at the same t' if c actually appears in t; otherwise we just set $t' = t$.

Lifting a certified term $(\mathcal{D}, t)$ works similarly. When lifting in a structure preserving way we set $t' = x : \tau \mapsto t[x/c]$, when lifting canonically we simply set $t' = t$ if c does not occur in t and $t' = x : \tau \mapsto t[x/c]$ otherwise.

Auto Lifting We have previously used the phrase of *autolifting* a certified term $(\mathcal{D}, t)$ to context $\mathcal{C}$. By this we simply mean that we lift $(\mathcal{D}, t)$ to $(\mathcal{C}, t')$ the canonical way; but in addition the autolift only succeeds if t and t' are identical.

Correctness Introducing contexts as a first-class concept into the kernel and providing axiomatic lifting between contexts is an interesting and we think promising new approach to building HOL systems. Contexts correspond closely to how mathematicians actually reason and it seems to be clear that they are reducible to ordinary logic; therefore we believe that our implementation of them is correct. Obviously it would be better to have a formal proof for this which we do not have yet. But then again, there are only very few theorem proving systems around where the kernel has actually been proven to be correct.

4 Structured Proof Scripting

Structured proof and purely functional structured programming are such a good fit, one might almost think that one was made for the other. Both are side-effect free, and both use a block structure notation which can be nested. Contexts as introduced in Section 3.3 make it easy to marry the two as we will outline in this section.

To turn purely functional structured programming into structured proof scripting, we augment the program state with two additional components, the *current* context, and the *literal* context. Literal context and current context are the same, except during the execution of a function call: then the literal context is the same as the literal context *at the point of the function definition.* The reason for this is that we need both a dynamically scoped context, which is the current context, and a lexically scoped one, which is the literal context.

There are several built-in statements which manipulate the current context. One of them is the **let**-statement which introduces either an Introduce or a Define context, depending on its argument which is a *term literal*. For example,

```
let 'x'
```

takes the current context, applies Introduce(`x`, $\mathscr{U}$) to it, and makes the result the new current context, whereas

```
let x_def: 'x = d'
```

makes Define(`x`, `'d'`) the new current context $\mathcal{D}$ and binds `x_def` to the theorem ($\mathcal{D}$, `'x = d'`).

Another logical statement is the **theorem**-statement. Assume the statement

```
theorem t: '∀ p. p → p'
  let 'p : ℙ'
  assume prop: 'p'
  prop
```

```
val tm = '∀ p. p → p'
def prf () =
  let p: 'p : ℙ'
  assume prop: p
  prop
theorem t: tm
  prf ()
```

Fig. 10. A simple theorem

shown in Figure 10 (left hand side) is issued in the current context $\mathcal{C}$. What is happening here is that the block starting at **let** is evaluated to yield the theorem ($\mathcal{D}$, `'p'`) where $\mathcal{D}$ results from $\mathcal{C}$ by first applying to it SpawnThread (if $\mathcal{C}$ is on the main context thread) and then Define(p, $\mathbb{P}$) and Assume(`'p'`). The theorem ($\mathcal{D}$, `'p'`) is lifted back into context $\mathcal{C}$ both in a canonical and in a structure preserving way, and the results are compared with ($\mathcal{C}$, `'∀ p. p → p'`); if at least one of them is equal (modulo $\alpha/\beta/\eta$-conversion), and in this case both are, the lifted theorem is bound to the identifier `t` (after a possible $\alpha/\beta/\eta$-conversion to match the theorem statement).

Logical statements and non-logical statements can be freely mixed as shown in the example in Figure 10 (right hand side), which proves the same theorem as the previous example. Note how the current context is dynamically passed along during the function call `prf()`.

In general it is not a good idea to introduce a logical constant with a fixed name `p` within a function because that name might already have been taken within the context that the function is called with. A more general way to write `prf` which avoids name clashes is shown on the left hand side of Figure 11. The expression `fresh "p"` returns a constant name similar to `p` which has not been used yet within the current context. This new name is then inserted into the

```
def prf () =                          def prf () =
  val p : String = fresh "p"            let '‹val p› : ℙ'
  let '‹p› : ℙ'                         assume prop: '‹p›'
  val p : Term = '‹p›'                  prop
  assume prop: '‹p›'
  prop
```

Fig. 11. Avoiding name clashes with `fresh`

term literal argument of **let** via the *quote* ‹p›. As a syntactic convenience, this can be written simply as displayed on the right hand side of Figure 11.

Quotes are useful not only in the above situation, but whenever one wishes to insert a string or a term into a term literal. Furthermore, quotes are used within *term literal patterns* to designate the holes inside the term literal pattern which are expected to be filled by the pattern match. For example here is how one would define a function which takes apart an implication into premise and conclusion and which fails if the argument term is not an implication:

```
def dest_imp '‹p› → ‹c›' = (p, c)
```

The allowed term patterns are those of the higher-order pattern fragment identified by Dale Miller [18,19].

To conclude this outline of structured proof scripting, let us return to the issue of current and literal context. Our examples so far have shown how the *current* context is used and evolved. But what is the *literal* context good for? The answer is that the literal context is used for parsing term literals. Imagine for example you had implemented a theory `Geometry` in which you define the logical constant `sin`. You also provide a helper function which determines if a term represents the `sin` function as shown in Figure 12. Now imagine that

```
theory Geometry                     theory A extends Geometry
let 'sin = ...'                     let 'sin = ...'
def                                 show is_sin 'sin'
  is_sin 'sin' = true
  is_sin _ = false
```

Fig. 12. Theories `Geometry` and `A`

another theory `A` also shown in Figure 12 extends theory `Geometry` and defines its own `sin` constant. What would you expect the **show**-statement to print, **true** or **false**? It will print **false**, as the term literal `'sin'` in theory `Geometry` refers to the logical constant `\Geometry\sin`, and in theory `A` refers to `\A\sin` instead. To implement this behaviour, term literals are parsed by the literal context, and not by the current context. Note however that quotes within term literals are evaluated still within the current context, not the literal one.

5 Conclusion

We have presented ProofScript, a language for structured proof scripting. We believe that this language can become a firm and simple foundation of ProofPeer, our system in the making for collaborative theorem proving. ProofScript today still needs to grow as a language and as an environment. Many features are still being developed, among them: extensible syntax, theories with assumptions, execution of/code generation for functions defined in logic, automation, and an actual interactive user interface. Yet, it is already a functioning theorem proving system with promise – we have bootstrapped the system to a point where it can convert certificates from an automated theorem prover for first-order logic into valid ProofScript proofs, which is described in a forthcoming paper [5].

References

1. ProofPeer, `http://www.proofpeer.net`.
2. S. Obua, J. Fleuriot, P. Scott, D. Aspinall: ProofPeer: Collaborative Theorem Proving. arXiv:1404.6186, 2013.
3. ProofScript, `http://proofpeer.net/topics/proofscript`.
4. S. Obua, P. Scott, J. Fleuriot: Local Lexing, `http://proofpeer.net/papers/locallexing`.
5. P. Scott, S. Obua, J. Fleuriot: Bootstrapping LCF Declarative Proofs, `http://proofpeer.net/papers/bootstrapping`.
6. M. Wenzel: Isar — A Generic Interpretative Approach to Readable Formal Proof Documents. doi:10.1007/3-540-48256-3_12, 1999.
7. S. Agerholm, M. Gordon: Experiments with ZF Set Theory in HOL and Isabelle. doi:10.1007/3-540-60275-5_55, 1995.
8. ProofPeer Root Theory, `http://proofpeer.net/repository?root.thy`.
9. D. Matichuk, M. Wenzel, T. Murray: An Isabelle Proof Method Language. doi:10.1007/978-3-319-08970-6_25, 2014.
10. M. Gordon, A. Milner, C. Wadsworth: Edinburgh LCF. doi:10.1007/3-540-09724-4, 1979.
11. S. Obua: Purely Functional Structured Programming. arXiv:1007.3023, 2010.
12. C. Okasaki: In praise of mandatory indentation for novice programmers. `http://okasaki.blogspot.co.uk/2008/02/in-praise-of-mandatory-indentation-for.html`, February 2008.
13. P. Landin: The Next 700 Programming Languages. doi:10.1145/365230.365257, 1966.
14. L. Paulson: Set Theory for Verification: I. From Foundations to Functions. doi:10.1007/BF00881873, 1993.
15. L. Paulson: Set Theory for Verification: II. Induction and Recursion. doi:10.1007/BF00881916, 1995.
16. S. Obua: Partizan Games in Isabelle/HOLZF. doi:10.1007/11921240_19, 2006.
17. S. Obua, J. Fleuriot, P. Scott, D. Aspinall: Type Inference for ZFH. doi:10.1007/978-3-319-20615-8_6, 2015.
18. D. Miller: A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. doi:10.1007/BFb0038698, 1991.
19. T. Nipkow: Functional Unification of Higher-Order Patterns. doi:10.1109/LICS.1993.287599, 1993.