A Domain-Specific Language for Name Modifiers

Kuen-Bang Hou (Favonia)[1]

University of Minnesota, Minneapolis, Minnesota, U.S.A
kbh@umn.edu

Abstract

Name management is an important factor of usability, but is often an overlooked aspect of the design of programming languages. The issue becomes even more pressing in the case of proof assistants, for a proof frequently depends on different definitions or properties of conflicting names. While names are often not part of the core type theory that a proof assistant is based upon, effective management of them is one decisive factor of practical usability. I therefore designed a compositional and extensible domain-specific language for manipulating hierarchical names. The language has been implemented as a standalone OCaml library and is used in the proof assistants cooltt and algaett.

1 Introduction

Software engineering practice encourages dividing a large program into multiple units, enabling separate or incremental compilation. The same principle applies to any serious mechanized proof using dependent type theory, as type-checking often takes significant time. The efficiency of compilation or type-checking is not the only reason—the division also serve as an opportunity to organize definitions and lemmas according to their conceptual closeness.

In order to support the use of multiple units, most programming languages provide “import” or “include” mechanisms for a unit to access content in another. Those mechanisms bring in declarations from another unit so that local code can access them as if they were defined locally. However, such an “import” feature introduces a new problem—the imported content might shadow existing content in the local scope. As a result, many languages allow programmers to rename, select, or hide parts of imported content. A common strategy is to place new content in a namespace, which can be understood as a special case of group renaming.

The interplay between these mechanisms—renaming, selection, hiding, namespaces, and others—is unfortunately unclear in most designs. For example, consider the Agda statement

```
open A using (x) renaming (x to y) hiding (y)
```

Would Agda make what’s named `x` available as both `x` and `y`, or only as `y`, or perhaps unavailable? The above statement would actually be rejected because of the ambiguity. Arguably, how renaming, selection, and hiding could work together is unclear from the concrete syntax.

The core issue is that *name modifiers are effectful.* Most designs chose to limit the syntax to avoid confusing interactions. The other approach, which I believe is more fruitful, is to design a proper domain-specific language with powerful combinators such as sequencing, union, and scoping. Such a language gives programmers the full power to manipulate names.

Another motivation to redesign how names work is to facilitate patching an API for compatibility. Often, a minor update of a library only introduces new bindings, and a client can start using the new API while supporting old ones by implementing those bindings during the transition. For example, suppose `is_prefix` was recently introduced in the `string` namespace in the standard library. The client should be able to inject its own implementation of `is_prefix` into `string` to support older versions of the standard library while updating its codebase.

The desire to patch libraries motivates an *implicit* treatment of namespaces, which means a namespace is only a group of names that happen to share the same prefix. The ML-style module system, on the other hand, limits the ability to change components of a module to make sure it always has a valid signature. There has been work on liberating ML-style modules [3], but the untyped nature of namespaces means we are not subject to these constraints and can derive a simpler design. Moreover, in the context of proof assistants, one should recognize top-level definitions might not have an (internalized) type anyway. It is perhaps easier and better to support both untyped namespaces and typed modules (or records in dependent type theory).

2 The Design

The effectfulness of name modifiers and the need to patch libraries suggest that all operations should work on names sharing a prefix (that is, a subtree) instead of individual names. For example, selection of `a.b` should be understood as selecting all names with the prefix `a.b`. Other important considerations include detection of typos, decoupling from export control, expressiveness, extensibility with custom hooks, and conciseness of the core language. Balancing out these factors led to a language with the following six operators:

- **Emptiness-Checking:** err if there are no matching bindings; this is to detect typos.
- **Scoping:** apply a modifier to names with a given prefix.
- **Renaming:** change names with a given prefix to names with the new prefix.
- **Sequencing:** apply a list of modifiers in order.
- **Union:** take the union of the results of a list of modifiers.
- **Hook:** apply a custom hook.

As a demonstration of expressiveness, the modifier that hides names with the prefix `a.b` can be implemented as a combination of **Scoping** (to focus on the names with the prefix `a.b`), **Emptiness-Checking** (to detect typos), and then **Union** of an empty list (to drops all bindings). Therefore, we do not need a modifier dedicated to hiding in the core language.

3 Discussions

The language has been implemented as a standalone OCaml library named Yuujinchou [1]. It was used in the experimental proof assistants cooltt and algaett. Racket is one of the few languages that provide an equally powerful language for modifying names, and the language has specialized operators for different phases in Racket [2]. However, its support of hierarchical names seems to be limited because its renaming and prefixing modifiers (`rename-in` and `prefix-in`) cannot directly place bindings under a namespace. Other than Racket, most programming languages and proof assistants give little power to its users, and perhaps our simple yet compositional domain-specific language could inspire.

References

[1] Kuen-Bang Hou (Favonia). Yuujinchou: Name modifiers. http://www.github.com/RedPRL/yuujinchou.

[2] Matthew Flatt and PLT. Importing and exporting: require and provide. https://docs.racket-lang.org/reference/require.html.

[3] Norman Ramsey, Kathleen Fisher, and Paul Govereau. An expressive language of signatures. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, page 27–40, New York, NY, USA, 2005. Association for Computing Machinery.