論 文 の 内 容 の 要 旨

論文題目　A study of protocol-checking and memory-management techniques for assisting library development
(ライブラリ開発を支援するためのプロトコル検査および
メモリ管理技術の研究)

氏　名　山﨑　徹郎

Today, we can access various kinds of libraries. By using an appropriate library, we can implement a complex application with a little effort. However, there is no library suitable for all applications.Applications in different problem domains need different libraries since not many program pieces are shared between them. Developing a specific library can assist only in limited application developments. To assist in a wider range of application developments, we aim to reduce the cost of library developments. Basically, developing a library is not a simple task. The main difficulty in library development comes from the abstract nature of libraries. Unfortunately, this complexity is inevitable. We aim to reduce the library development cost by assisting developers in extra tasks accompanied by library developments. There already exists many research and tools which assist in library development. In this thesis, we focus on two approaches to assist in library developments and propose we propose three additional techniques. The first approach is about library protocol checking. Libraries often provide many usages to adopt various kinds of applications. However, from a viewpoint of a library developer, the larger number of usages a library provides, the more difficult it is to control all the behavior. To prevent an overlooked unexpected behavior from introducing bugs, library developers can check the usages and prevent illegal usages from execution. A set of rules about how a library can be used is called a library protocol. We propose an embedding technique that makes type checkers emulate LR parsers. This technique can be used to check protocols of libraries that provide fluent interfaces. We experimented on how long our method took to check protocols and revealed that the time complexity

is quadratic, although it is expected to be linear. The second approach is about Foreign Function Interfaces (FFIs). An FFI can reduce the library development cost since it enables engineers to use foreign libraries without translating their programs. However, an FFI can cause memory leaks when both programming languages support garbage collection. When a cyclic reference goes through both languages, the cycle will not be collected since neither collector can determine whether the cycle is garbage or not. Note that our interest is in reusing foreign libraries. Thus we do not want to modify the language that the foreign library is written in. We propose a backup garbage collection algorithm that can collect cross-language cyclic references without customizing both of the collectors. Our algorithm copies reachability between objects from a language to the other language and reproduces it as references between objects. Once the copy finishes, the collector in the other language can detect all garbage objects correctly, and the cross-language garbage cycle will be broken. We experimented on how long our algorithm takes to collect cyclic garbage. We also propose a garbage collection algorithm for self-reflective garbage collectors. Experimenting with a garbage collection algorithm is time-consuming. Existing collectors are complex to customize, and it often saves our efforts to develop a whole new programming language. Customizing a garbage collector could be easier if the customization was self-reflective. However, it is not easy to design a self-reflective feature. We design a simple self-reflective interface to customize a copying garbage collector (named `copy-time callback`) to discuss what problem will occur in self-reflective collector customization. `Copy-time callback` allows registering a callback function which is called each time the collector copies an object. We discovered a problem that the callback function creates objects during garbage collection, and it can consume a huge amount of memory. Our garbage collection algorithm places objects created during collection in a special region and invokes minor garbage collection to compress them. And our algorithm will move the objects managed in the special region into the normal region in which other objects are managed when the collection finishes. We experiment on how small a memory region our algorithm can execute our microbenchmark and confirmed that our algorithm could save the memory.