Contract systems from first principles

ACM Reference Format:
. 2024. Contract systems from first principles. 1, 1 (August 2024), 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

2 BACKGROUND

3 SIMPLIFYING SEQUENTIAL CONTRACTS

3.1 From CFCP to the untyped λ calculus

CFCP [1] is an extension to the untyped λ-calculus for expressing contracts and their monitoring constructs. Its syntax is depicted in fig. 1 The language consists of two contract types (flat and higher-order) κ which are separate from the term language. The term language e then contains a *monitoring construct* which relates the contract language to the term language and to a set of *blame labels*. Thus, the main purpose of the monitoring construct is to attach blame labels to the appropriate values in the program and to *dispatch* on the type of contract to achieve *flat* or *higher-order* contract monitoring respectively.

The semantics of the monitoring constructs is typically expressed as a reduction to a term in the untyped λ-calculus and does not require terms or values outside of it. This typical reduction rule in CFCP for the contract monitoring construct is depicted below.

$$mon^{j,k}(\mathit{flat}\ e_1, e_2) \longrightarrow \mathsf{if}\ (e_1\ e_2)\ e_2\ \mathsf{blame}\ j$$
$$mon^{j,k}(\kappa_1 \mapsto \kappa_2,\ f) \longrightarrow \lambda x.mon^{j,k}(\kappa_2, f(mon^{k,j}(\kappa_1, x)))$$

The result of reducing a flat contract monitor is e_2 if the predicate embedded in the flat contract holds for e_2, and a blame error otherwise. Reducing a higher-order contract monitor results in a λ-expression so that it can be used as a monitored version of f. Its semantics is straightforward. First, it checks the domain-contract κ_1 on its argument, and then checks its range-contract κ_2 on the result of applying function f to this value. Swapping of blame labels for checking the domain contract is crucial for higher-order functions. This is because the roles of supplier and user are swapped. Whereas originally, the user was responsible for the arguments of the function, the supplier is now responsible for providing arguments to the function passed as an argument.

It is important to realize that the contract language is separate from the term language such that the contract specification and monitoring construct is entirely static and not influenced by the dataflow of the program. This renders a transformation to the untyped λ-calculus trivial by recursively applying the reduction rules depicted above until a flat contract is reached and the program is free of monitoring constructs and contract specifications.

Author's address:

ACM XXXX-XXXX/2024/8-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$
\begin{aligned}
e &::= \lambda x.e \mid e\ e \mid b \mid \mathit{mon}(\kappa, e) \mid \text{if } e\ e\ e \\
\kappa &::= \mathit{flat}\ e \mid \kappa_1 \mapsto \kappa_2 \\
b &::= \mathit{bool} \mid \mathit{int}
\end{aligned}
$$

Fig. 1. Syntax for CFCP

3.2 Higher-order contract combinators

The CFCP approach to contract specification is rather limiting. A developer cannot build abstractions around its contracts to easily implement and re-use more complex contracts. Racket [] provides a contract system that addresses these needs. It does so by including the contract language in the term language, therefore making all abstraction and programming facilities from the term language available for contract specification. In Racket, this results in a library of *contract combinators* that are implemented as ordinary functions taking contracts as input and producing contracts as output. For instance, an `or/c` contract takes a number of contracts as an input and returns a contract that is valid for values satisfying one or more input contracts. To facilitate contract combinators, the syntax is adapted as depicted below (changes highlighted in grey).

$$
\begin{aligned}
e &::= \lambda x.e \mid e\ e \mid b \mid \mathit{mon}(\boxed{e}, e) \mid \text{if } e\ e\ e \mid \boxed{\kappa} \\
\kappa &::= \mathit{flat}\ e \mid \kappa_1 \mapsto \kappa_2 \\
b &::= \mathit{bool} \mid \mathit{int}
\end{aligned}
$$

Now that contracts are also values in the term language, and contracts in monitor constructs are no longer static, a transformation to the untyped λ-calculus becomes more involved. This is because the contract monitor construct has to perform run-time dispatching on the contract value. One solution is to represent contracts as functions which encode their own monitoring semantics and reify blame labels as values. For example, a flat contract can be represented as the function $\lambda j.\lambda k.\lambda x.\text{if } (e_1\ x)\ x\ \mathit{blame}j$, similar for higher-order contracts.

3.3 Towards Blame Assignment through Provenance Tracking

REFERENCES

[1] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 48–59. https://doi.org/10.1145/581478.581484