# Temporal Dependent Contracts for Higher-Order Functions

Yuki Satake　Hiroshi Unno

We present a novel contract system that can enforce both temporal and value-dependent properties of higher-order functions. To specify such contracts, we introduce a multi-modal logic over traces of events such as function calls and returns. We show that our logic subsumes Disney *et al.*'s declarative contract language and dependent refinement types, which have respectively been used to express temporal and value-dependent properties in prior contract systems. We implemented a prototype dynamic contract checker for the OCaml functional language based on the proposed method.

## Summary

Contract systems for higher-order functional languages have been studied extensively [1–3, 6]. We present a novel contract system that can enforce both temporal and value-dependent properties of higher-order functions which were impossible by the prior systems.

Previous contract systems [3, 6] have used dependent refinement types [4, 5] to specify value-dependent properties. For example, the refinement type $(x : \{x : \texttt{int} \mid x \geq 0\}) \to \{y : \texttt{int} \mid y > x\}$ of the factorial function `fact` specifies that

- if an integer $x$ is passed to `fact`, $x$ must be non-negative, and
- if an integer $y$ is returned by `fact`, $y$ must be strictly greater than $x$.

By contrast, Disney *et al.* has proposed a declarative contract language for specifying temporal properties of higher-order programs [1]. For example, consider the following temporal contract obtained (and slightly simplified) from [1]:

```
sort : int list ->
       (cmp : int -> int -> bool) ->
       int list
where not ... call(sort,_) !ret(sort,_)*
          call(sort,_)
```

The regular expression in the `where` clause specifies that the function `sort` can only be called after all the previous calls to `sort` have returned. Consequently, `sort` is never called from the function passed as `cmp` and, in a multi-threaded setting, `sort` is not re-entrant.

Our goal is to provide a unified contract language that subsumes both of the above two approaches. To this end, we introduce a multi-modal fixed point logic over traces of events, generated by an executing program, such as function calls (denoted by $\mathbf{call}(f, x)$ where $f$ is the callee function and $x$ is the actual argument) and returns (denoted by $\mathbf{ret}(f, y)$ where $y$ is the return value). For example, the above value-dependent contract of `fact` represented as the refinement type is encoded as

高階関数のための時相的依存契約



佐竹 佑規　海野 広志, 筑波大学, University of Tsukuba.

the following two formulas in our logic:

$$\nu X.\ \Box_N X \wedge (\forall x.\ \mathbf{call}(\mathtt{fact}, x) \Rightarrow x \geq 0)$$

$$\nu X.\ \Box_N X \wedge (\forall x.\ \mathbf{call}(\mathtt{fact}, x) \Rightarrow \Box_R(\forall y.\ \mathbf{ret}(\mathtt{fact}, y) \Rightarrow y > x))$$

Here, the atomic formula $\mathbf{call}(\mathtt{fact}, x)$ (resp. $\mathbf{ret}(\mathtt{fact}, y)$) means that the currently focused event in the trace is of the form $\mathbf{call}(\mathtt{fact}, x)$ (resp. $\mathbf{ret}(\mathtt{fact}, y)$). The modal operator $\Box_N$ is used to move the focus to the Next event in the trace. On the other hand, the operator $\Box_R$ is used to move the focus to the Return event corresponding to the currently focused call event.

Furthermore, the proposed multi-modal logic is expressive enough to encode the temporal contract of `sort` as follows:

$$\nu X.\ \Box_N X \wedge \left( \begin{array}{l} \mathbf{call}(\mathtt{sort}, \_) \Rightarrow \\ \nu Y.\ \Box_N \left( \begin{array}{l} \mathbf{ret}(\mathtt{sort}, \_) \vee \\ \neg\mathbf{call}(\mathtt{sort}, \_) \wedge Y \end{array} \right) \end{array} \right)$$

We also propose a method for dynamically checking temporal dependent contracts represented as formulas of the multi-modal logic. We implemented a prototype dynamic contract checker for the OCaml functional language based on the proposed method.

## 参 考 文 献


[1] Disney, T., Flanagan, C., and McCarthy, J.: Temporal higher-order contracts, *ICFP '11*, ACM, 2011, pp. 176–188.

[2] Findler, R. B. and Felleisen, M.: Contracts for Higher-order Functions, *ICFP '02*, ACM, 2002, pp. 48–59.

[3] Flanagan, C.: Hybrid type checking, *POPL '06*, ACM, 2006, pp. 245–256.

[4] Freeman, T. and Pfenning, F.: Refinement types for ML, *PLDI '91*, ACM, 1991, pp. 268–277.

[5] Xi, H. and Pfenning, F.: Dependent types in practical programming, *POPL '99*, ACM, 1999, pp. 214–227.

[6] Xu, D. N., Peyton Jones, S. L., and Claessen, K.: Static contract checking for Haskell, *POPL '09*, ACM, 2009, pp. 41–52.