Manifest Contracts for OCaml

Yuki Nishida
Kyoto University
nishida@fos.kuis.kyoto-u.ac.jp

Atsushi Igarashi
Kyoto University
igarashi@kuis.kyoto-u.ac.jp

Synopsis

We report our work in progress on an extension of the OCaml compiler with a manifest contract system, where contracts are integrated into a type system as "refinement types". Although we aim at hybrid contract checking in future, the current implementation checks all contracts at run time by inserting coercions where they are necessary. We discuss simple use cases of our system, impact on type inference, and a few details of the modified compiler.

1 Background

Software contracts (or just contracts), advocated originally by Meyer as the "design by contract" principle for Eiffel [5], have been actively studied as a promising way to construct robust software. Contracts are usually written in the programming language itself and checked dynamically. Findler and Felleisen [1] extend contracts to a functional language by introducing higher-order contracts.

Greenberg, Pierce, and Weirich [3] coined the term *manifest contracts* to refer to contract systems where contracts are integrated into a type system. A manifest contract system is usually equipped with *refinement* types of the form $\{x : T \mid e\}$, which intuitively denotes the subset of the underlying type T where the Boolean expression e evaluates to `true`. For example, $\{x : \texttt{int} \mid x > 0\}$ stands for a type of positive integers. By using a refinement type, a division function (where the first argument is the dividend and the second argument is the divisor) could be given the type $\texttt{int} \to \{x : \texttt{int} \mid x \neq 0\} \to \texttt{int}$ in order to express that the divisor must not be zero. One of the motivations for manifest contracts is hybrid type checking [2, 4], where contract checking (checking whether contracts hold or not) is done by combination of static subtyping checks and dynamic casts.

Our goal is to implement a manifest contract system for OCaml to get more programmers to use software contracts. In previous work [6], we implemented coercions for dynamic contract checks as Camlp4 macro. However, a programmer had to insert coercions manually since the compiler was ignorant of refinement types. Although no static contract verification is supported yet, our new compiler-based implementation integrates refinement types into the type system in order to achieve better support to manifest contracts. Namely, a programmer can use refinement types anywhere types can appear and, according to refinement types in a program, the compiler inserts coercions where they are needed.

2 Example of Use

In this section, we show how our system works through examples. The code in this section can be actually handled with our system.

In our system, a refinement type is described as `{@ p of e @}`, where `p` is a pattern and `e` is an arbitrary `boolean` expression, though we are not concerned with the behavior of a program which contains a refinement type in which `e` has side effects. Note that, in our system, the underlying type of a refinement type does not have to be specified explicitly, though it can be part of a pattern as `(p : T)`.

Dynamic checking is done by coercion into an appropriate refinement type. For example, in the following code, we check whether the `1` is positive or not.

```
# (1 :> {@ x of x > 0 @});;
- : {@ x of x > 0 @} = 1
```

When dynamic checking fails, the code is blamed by `Assert_failure`. The following code is blamed since `0` is not positive.

```
# (0 :> {@ x of x > 0 @});;
Exception: Assert_failure ("_none_", 1, -1).
```

(The current implementation does not yet spot where things go wrong, though.)

Refinement types are fully integrated as OCaml types and they can be used everywhere a type is expected, and types are inferred as the following example shows.

```
# let f (x:{@ x of x > 0 @}) = x;;
val f : {@ x of x > 0 @} -> {@ x of x > 0 @} = <fun>
# let g x = f x;;
val g : {@ x of x > 0 @} -> {@ x of x > 0 @} = <fun>
```

During type checking, a coercion is automatically inserted if types are not identical but *compatible*—identical after removing refinements.

```
# let div x (y:{@ x of x <> 0 @}) = x / y;;
val div : int -> {@ x of x <> 0 @} -> int = <fun>
# div 4 2;;
- : int = 2
# div 4 0;;
Exception: Assert_failure ("_none_", 1, -1).
```

Here, the typechecker allows `0` of type `int` to be passed to where `{@ x of x > 0 @}` is expected since the two types are compatible.

A programmer can see where coercions are inserted by `ocamlc -drcaml`. For example, the code above is translated by coercion insertion as follows:

```
let div x ((_ as y) : {@ x of x <> 0 @}) = x / y
div 4 (2 :> {@ x of x <> 0 @})
div 4 (0 :> {@ x of x <> 0 @})
```

3 Implementation

In this section, we discuss an impact of refinement types on type inference, how coercions are inserted and implemented, and some OCaml-specific details.

The essential typing rule of our system is the following.

$$\frac{\Gamma \vdash t : S \qquad S \sim T}{\Gamma \vdash (t :> T) : T} \qquad \text{(T-Coerce)}$$

The rule says "a term t can be coerced into an arbitrary type T with run time checking if T is compatible with the type of t". Intuitively, two types are compatible if two types are identical after removing the refinements. For example, $\{x : \texttt{int} \mid x > 0\}$ and `int` are compatible. The relation $\sim$ is formally defined as the reflexive symmetric compatible relation closed under the rule:

$$\frac{S \sim T}{\{x : S \mid e\} \sim T} \qquad \text{(C-Ref)}$$

As demonstrated in the last section, our compiler inserts coercions automatically to avoid programmers' burden to insert many coercions manually. The basic idea for coercion insertion follows Flanagan's λ^H [2]: We insert a coercion where the type of an expression is not identical but compatible with the expected type.

Coercion insertion is implemented as part of the type inference phase. A coercion is inserted into an actual argument of a function application when unification between the argument type and the parameter type of the function fails. The types in the inserted coercion are checked with compatibility as shown above.

Then, the compiler translates a coercion into dynamic checking code to implement the operational semantics of coercions, also following λ^H. A coercion to a refinement type $(e :> \{x : T \mid e'\})$ is compiled to a call to `assert` to check if e' evaluates to true; one to a function type $(e :> S \to T)$ will be compiled to $\texttt{let } v = x \texttt{ in fun } x \to (e\ (x :> S') :> T)$, where S' is the domain type of e.

One problem specific to OCaml arises from how abstract syntax trees are represented. In the OCaml implementation, the datatype to represent abstract syntax trees with type information and one to represent type information are defined in separate files, that is, separate modules. Due to refinement types, however, we would like to use the former datatype in the latter datatype, making these datatypes mutually recursive. To solve the problem, we use the recursive module extension of OCaml. (It would require a lot of modifications to combine these data types with usual recursive type definitions in a single module.) For this, we only move the datatype definitions from the files `types.ml`, `typedtree.ml`, and so on, which mutually refer to each other into one file (assume `recmod.ml`) as follows

```
module rec Types : sig
(* the part of the code of types.mli *)
end = struct
(* the part of the code of types.ml *)
end and Typedtree : ...
```

and we simply re-export the defined modules. For example, the modified `types.ml` has the following code.

```
include Recmod.Types
```

4 Future Work

In our system, all contracts are checked dynamically. We plan to reduce the overhead of dynamic contract checking by using static or hybrid contract verification techniques [2, 7] to remove coercions. We have been focusing on the core part of ML. Investigating interactions between manifest contracts and other features of OCaml is interesting future work.

Acknowledgements. We would like thank Jacques Garrigue for discussions on the OCaml compiler implementation and anonymous reviewers for useful comments. This work was supported in part by Grant-in-Aid for Scientific Research (B) No. 25280024 from MEXT of Japan.

References

[1] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proc. of ACM ICFP, Pittsburgh, PA*, pages 48–59, 2002.

[2] C. Flanagan. Hybrid type checking. In *Proc. of ACM POPL*, pages 245–256, 2006.

[3] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *Proc. of ACM POPL*, pages 353–364, 2010.

[4] K. Knowles, A. Tomb, J. Gronski, S. N. Freund, and C. Flanagan. SAGE: unified hybrid checking for first-class types, general refinement types, and dynamic (extended report). Technical report, UCSC, 2007.

[5] B. Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[6] T. Sekiyama, Y. Nishida, and A. Igarashi. Manifest contracts for datatypes. In *Proc. of ACM POPL*, pages 195–207, 2015.

[7] N. Vazou, E. L. Seidel, and R. Jhala. Liquidhaskell: experience with refinement types in the real world. In *Proc. of ACM SIGPLAN symposium on Haskell*, pages 39–51, 2014.