PACT: CONTRACTS FOR HIGHER-ORDER FUNCTIONS

Ryan Downing, Shu-Huai Lin, Andrew Mao, Nikhil Pateel

University of Maryland
Department of Computer Science

ABSTRACT

We implement contracts for higher-order functions for Outlaw, a simplified Racket-like programming language and compiler. Our implementation builds custom AST nodes for each function contract, which are recursively expanded into conditional checks with blame before the tree is compiled to assembly. Higher order functions are wrapped with conditional checks from the corresponding contract. We profile Pact[1] on a set of scripts, and find that contracts add a small amount of memory that scales linearly with the number of parameters and contract depth

1. INTRODUCTION

Behavioral software contracts are a useful programming language feature for debugging and ensuring intended program behavior. Contracts establish a boundary between two parties, such as two functions or modules. Whenever a value crosses this boundary, the contract monitoring system performs contract checks, making sure the partners abide by the established contract. In addition, the contract checker will assign blame to the responsible party, which helps programmers locate the source of the problem.

Developing contracts for higher-order functions is challenging for two reasons. First, it is impossible to statically resolve predicates on functions. Second, higher-order functions complicate blame assignment.

We propose to implement higher-order contracts with correct blame, based on [1]. Our language and compiler, called Pact, will support correct blame, writing custom predicates, and predicate logic. Our project roadmap is as follows:

1. We implement first-order contracts with support for user-defined predicates.
2. We implement higher order contracts.
3. We implement blame for higher-order contracts.
4. We profile Pact on a set of scripts, and measure the memory usage before and after adding contracts. We find that contracts add additional memory that scales linearly with the number of parameters and contract depth.

[1] https://github.com/shuai626/pact

2. BACKGROUND

2.1. Contracts in Detail

We use the examples of [1] to introduce function contracts. First we have a first-order function contract applied to f.

```
define/contract f : int[>9] -> int[0,99]
val rec f = lambda x. ···
```

A first-order function contract consists of a pair of predicates, where a predicate is a function of a single argument that returns a boolean (examples in Racket are integer? and string?). The contract has one predicate for the domain of f and one for the range.

The contract states that the argument to f must be an int greater than 9 and that f produces an int between 0 and 99. To enforce this contract, a contract compiler inserts code to check that x is in the proper range when f is called and that f's result is in the proper range when f returns. If x is not in the proper range, f's caller is blamed for a contractual violation. Symmetrically, if f's result is not in the proper range, the blame falls on f itself.

This mechanism for checking contracts does not generalize to contracts with higher-order functions, also known as higher-order contracts. Consider this contract:

```
g : (int[> 9] -> int[0,99]) -> int[0,99]
val rec g = lambda proc. ···
```

The contract's domain states that g accepts int-¿int functions and must apply them to ints larger than 9. In turn, these functions must produce ints between 0 and 99. The contract's range requires g to produce ints between 0 and 99.

There is no algorithm to statically determine whether proc matches its contract, and it is not even possible to dynamically check the contract when g is applied. This is because the function argument may not be called inside g, but it may be called elsewhere.

2.2. Higher Order Contracts and Blame

The solution that [1] proposes for checking higher-order contracts is to postpone contract enforcement until some function receives a first-order value as an argument or produces a first-order value as a result.

In the example of [Fig 2], the contract checker delays the enforcement of g's contract until g's argument is actually applied and returns.

In general, a higher-order contract checker must be able to track contracts during evaluation from the point where the contract is established to the discovery of the contract violation, potentially much later in the evaluation.

The rule for blame assignment is based on the number of arrows that appear to the right of each base contract, odd is the caller's fault, and even is the callee's. As such it follows that for a first-order contract, the input contracts blame the function's caller, where the return contract blames the function itself. For a higher order contract, this relationship is reversed at each successive layer.

Consider the example from the introduction again, but with a little more detail. Imagine that the body of g is a call to f with 0:

```
;; g : (integer → integer) → integer
(define/contract g
((greater-than-nine? 7 ->
between-zero-and-ninety-nine?)
7 ->
between-zero-and-ninety-nine?)
(lambda (f) (f 0)))
```

At the point when g invokes f, the greater-than-nine? portion of g's contract fails. According to the even-odd rule, this must be g's fault. In fact, g does supply the bad value, so g must be blamed. Imagine a variation of the above example where g applies f to 10 instead of 0. Further, imagine that f returns -10. This is a violation of the result portion of g's argument's contract and, following the even-odd rule, the fault lies with g's caller. Accordingly, the contract enforcement mechanism must track the even and odd positions of a contract to determine the guilty party for contract violations.

3. IMPLEMENTATION

The following section details how we implement the Pact compiler. We fork and modify the "Outlaw" language and compiler from the CMSC430 class [2]. In addition to the traditional steps of parsing and compiling, Pact runs an additional intermediate pass, called remove-contracts, which expands the AST nodes corresponding to contracts, into expressions that execute the contract checking logic.

[2]https://github.com/cmsc430/www/tree/main/langs/outlaw

The syntax for our contracts implementation is the same as Racket contracts. In our grammar, a contract definition (DefnContract) is defined as list of contract expressions, which corresponds to the parameters of the target function, and a final output contract for the return value. A contract expression is recursively defined as either a predicate or a pair of contract expressions.

```
(struct DefnContract
        (defn contract-list ret-contract))
```

Outlaw provides a standard library of functions that includes predicates such as integer? string?, and so on. Pact also supports user-defined predicates.

3.1. Contract Expansion

During the remove-contracts phase, the contract nodes are expanded into expressions that execute contract checking at runtime. We process the list of contract expressions. If the contract-expression is a predicate, it is expanded into an if statement that checks the contract condition, else throws an error with proper blame assignment. If the contract-expression is nested, a high-order contract, then the contract is recursively expanded using a let statement to reassign the respective variable to a lambda with the contracts injected.

To form the lambda, we generate symbols to represent the arguments of the contract, and then form the body expression by beginning with an application to the original variable with the generated lambda arguments, and expanding the in and out contracts around the application, as is done with the base expression. The implementation logic for the lambda expansion is shown in lines 20-32 of figure 2.

As a final detail, the contract on the return value is expanded by creating an intermediary variable through a let expression which assigns the function's return value to the variable, which is then checked by the contract.

3.2. Blame

To implement correct blame assignment, a boolean flag representing the even/odd parity is passed into expand-contract, and is toggled on each recursive call. The parity of the flag determines who gets blamed, as discussed in section 2.2.

4. EXPERIMENTS

We benchmark our compiler on a small set of scripts, with and without contracts.

1. `bake.rkt`: a simple function
2. `higher_order.rkt`: a script with higher order contracts
3. `recursive.rkt`: a script with a recursive function

Program	Compiled size (bytes)	Compiled size with contracts (bytes)	Size difference (%)
bake	381992	382832	2.2
higher_order	382360	384736	6.2
recursive	382408	383256	2.2

Table 1. Size difference of programs compiled with and without contracts.

We benchmark our compiler on a small set of scripts: For each script, we measure the size of the compiled assembly file, with and without a contract.

We vary the number of function arguments and the nesting depth of `higher_order.rkt`, and plot the memory. We find that contracts add a small amount of memory that scales linearly with the number of parameters and contract depth; adding a zeroth-order parameter adds about 780 bytes, and increasing the depth of a function contract with zero arguments adds about 755 bytes.

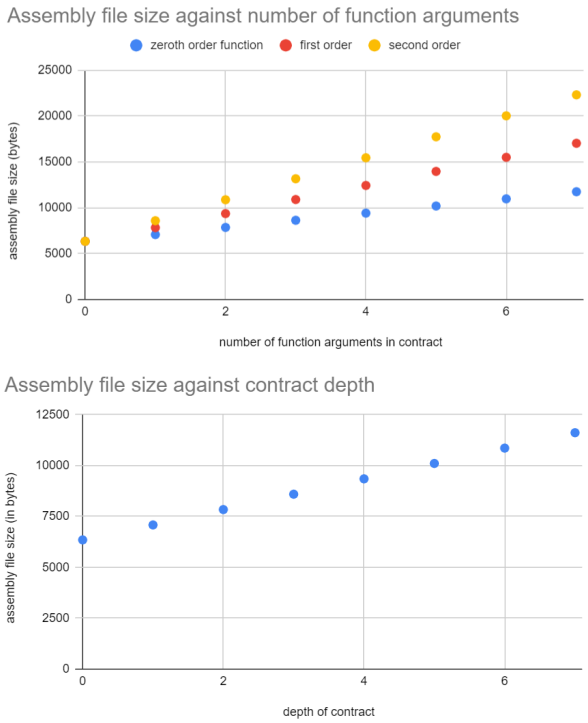

Fig. 1. Memory usage of scripts as we add function parameters, and depth.

5. RELATED WORK

Contracts for Higher-Order Functions [1] Introduces contracts that support higher-order functions, by postponing contract enforcement until the point where the responsible functions are called. The authors establish basic properties of the model (type soundness, etc.). The authors introduce "dependent contracts", or a contract that relates a function's result to its argument, and motivate first class contracts.

6. CONCLUSION

We implement Pact, a language and compiler based on Outlaw, that supports contracts, contracts for higher-order functions, and correct blame.

Our implementation adds an intermediate step on the frontend that recursively expands a contract node into conditional checks. We profile Pact on a set of scripts, and find that contracts add a small amount of memory that scales linearly with the number of parameters and contract depth.

Next steps for this project include adding logical operators such as and/c and not/c, adding support for function contracts at export time (analogous to the provide and contract-out keywords in Racket), more detailed error messages, and implementing a gradual type system using contracts.

```
; id id [Listof id] Expr [Listof Predicate] Predicate -> Defn
(define (expand-defcontract f g xs expr ins out)
  (let* ([truths (build-list (length ins) (lambda (x) #t))]
         [in-expr (foldl expand-contract expr xs ins truths)]
        [out-expr (expand-outcontract in-expr #f out)])
    (Defn f (Lam g xs out-expr))))

; Expr Predicate -> Expr
(define (expand-outcontract expr blameyou out)
  (let ([label (gensym "out")])
    (Let (list label) (list expr) (expand-contract label out blameyou (Var label)))))

; id Predicate Expr -> Expr
(define (expand-contract x c blameyou expr)
  (match c
    [(? list? c) (expand-higher-contract expr x blameyou c)]
    [(? symbol? c) (If (App (Var c) (list (Var x))) expr (errorast (if blameyou "you" "me")))]))

; Expr id (Listof Predicate) -> Expr
(define (expand-higher-contract expr x blameyou cs)
  (match (extract-last cs)
    [(cons ins out)
     (let ([syms (n-syms (length ins) "hcon")])
       (let ([blames (build-list (length syms) (lambda (x) (not blameyou)))])
       (Let (list x)
            (list (Lam (gensym "lam")
                       syms
                       (expand-outcontract
                        (foldl expand-contract (App (Var x) (map Var syms)) syms ins blames)
                        blameyou
                        (car out))))
            expr)))]))
```

Fig. 2. the core logic of remove-contracts

7. REFERENCES

[1] Robert Bruce Findler and Matthias Felleisen, "Contracts for higher-order functions," in *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, 2002, pp. 48–59.