Relationally-Parametric Polymorphic Contracts

Arjun Guha
Brown University
arjun@cs.brown.edu

Jacob Matthews
University of Chicago
jacobm@cs.uchicago.edu

Robert Bruce Findler
University of Chicago
robby@cs.uchicago.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Abstract

The analogy between types and contracts raises the question of how many features of static type systems can be expressed as dynamic contracts. An important feature missing in prior work on contracts is parametricity, as represented by the polymorphic types in languages like Standard ML.

We present a contract counterpart to parametricity. We explore multiple designs for such a system and present one that is simple and incurs minimal execution overhead. We show how to extend the notion of contract blame to our definition. We present a form of inference that can often save programmers from having to explicitly instantiate many parametric contracts. Finally, we present several examples that illustrate how this system mimics the feel and properties of parametric polymorphism in typed languages.

Categories and Subject Descriptors D.2.4 [*Software / Program Verification*]: Programming by contract

General Terms Languages, Reliability

Keywords Contracts, polymorphism, parametricity

1. Motivation and Introduction

Because most dynamic languages lack static type systems, dynamically-enforced contracts [17] play an invaluable role in both documenting and checking the obedience of pre- and post-conditions. Even in languages with static type systems, they enable the statement of richer conditions than the type system can capture. Indeed, as of 2007-01-26, contracts are the most requested addition to Java. Assertions are, likewise, particularly popular in C programs. In fact, 60% of the C and C++ entries to the 2005 ICFP programming contest [5] used assertions, even though the software was produced for only a single run.

Designing a good contract system is non-trivial. For instance, without care, contract systems can easily be unsound in the presence of language features such as subtyping [8]. Furthermore, in higher-order languages, it can often be subtle to properly ascribe blame to the faulty component [6]. Despite these difficulties, contracts have grown into significant specification systems for non-trivial languages, including C [26], C++ [23], C# [16], Haskell [12], Java [14], Perl [2], Python [22], Scheme [24], and Smalltalk [1].

Where type systems provide static proofs, contracts provide a corresponding notion of dynamic falsification. The growing importance of contracts begs the question of how far we can push the correspondence between contracts and types. Prior work has already shown the correspondence for base types, subtyping and higher-order functions [6, 8]. There is, however, no clear contract counterpart for parameterized types.

Parameterized Types Programmers who use Standard ML [19] are already familiar with parameterized types, but so are users of other languages. For example, Java's Generics permit parameterized types:

```
public interface List<E>  {
  void add(E x);
  Iterator<E> iterator();
}
```

Similarly, C++ templates are widely used in the C++ STL to define generic data structures.

While their presence in other programming languages is a useful indicator of their value, all these uses are in languages that already have static type systems. Does an analog of parametrized types make sense in a latently-typed, dynamic language?

DLS'07, October 22, 2007, Montréal, Québec, Canada.
Copyright © 2007 ACM 978-1-59593-868-8/07/0010…$5.00

Parameterization for JavaScript Consider the implementation of Flapjax [9], a new programming language for implementing contemporary Web applications. It is built as a large library written in JavaScript, and can therefore be deployed directly in existing browsers. A central tenet of Flapjax is the notion of *behaviors* [4]. These are values that vary over time; when a behavior changes, all values that depend on it update automatically, providing a form of higher-order dataflow programming. This enables the concise and declarative expression of sophisticated Web applications.

One key function in the Flapjax implementation is `lift`. This transforms a pure JavaScript function to accept a behavior as an argument and produce a behavior as a result. Operationally, when the value of the input behavior changes, it recomputes the result and updates the value of the output behavior. Thus (a simplification of) its type, using a Haskell-like type syntax, would be:

```
lift :: forall a b. (a -> b) -> Beh a -> Beh b
```

Note that `lift` does not care about the actual values that the function `a -> b` consumes and produces. It simply sends the values of the behavior `Beh a` into the function, and updates the value of `Beh b` with the results of the function. The type variables `a` and `b` (where the `forall` indicates universal quantification) reflect both this agnosticism and expectation of consistency: the fact that they are type *variables* indicates the agnosticism, while the pattern of repeating `a` and `b` dictates the expectation of consistency.

Another, more subtle, function in the Flapjax implementation is `switch`, which is used to implement conditionals (amongst other things). This function consumes a behavior carrying behaviors, and returns a behavior—but is agnostic to the values carried by the inner behaviors. Therefore, we may ascribe it the following type:

```
switch :: forall a. Beh (Beh a) -> Beh a
```

It is easy to abuse this function by passing it a behavior that does not always carry behaviors. The lack of any static typing in JavaScript means that we can only check this property at runtime. Moreover, this property cannot be checked and dispatched only when `switch` is applied; it must instead be checked every time the value of the argument changes to ensure it is still a behavior. Furthermore, when there is an error, it should be ascribed to programmer code, not to the library that implements Flapjax.

Parametric Contracts To document such functions, it would clearly be useful to have parameterized contracts of the form (**forall** ($\alpha \ldots$) *C*), where the identifiers $\alpha \ldots$ may occur free in the contract *C*. We could then write the contracts for our two introductory examples above as

(**forall** (*a b*) ((*a* → *b*) → ((*Beh a*) → (*Beh b*)))) ; lift
(**forall** (*a*) ((*Beh* (*Beh a*)) → (*Beh a*))) ; switch

Parametricity However, (**forall** ($\alpha \ldots$) *C*) is merely notation that may be interpreted in a number of ways. We wish to interpret such contracts as *parametric contracts*, in the spirit of relational parametricity [25], which is a property of languages such as Standard ML. (We limit our attention to first-order polymorphism, as we are interested in inferring contracts using techniques inspired by the Hindley-Milner type system [18].)

As Wadler has shown [28], relational parametricity provides strong guarantees about functions just from their types. Informally, if a function f accepts an argument of type α, parametricity ensures that the argument is abstract to f—the argument may not be examined or deconstructed by f in any way. Moreover, f may not determine the concrete type of α. This seemingly harsh restriction in fact allows rich abstractions. Since values of type α are abstract to f, a programmer may change the concrete type of α and be assured that the behavior of f will not change. Therefore, we want contracts of the form (**forall** ($\alpha \ldots$) *C*) to preserve the static notion of parametric polymorphism[1] at runtime. Such contracts will implement—that is to say, detect violations of—relational parametricity. Furthermore, we will want to ensure we can correctly blame components when parametricity is violated.

Shortcomings Two important caveats are in order. First, implementing contracts that work with mutation is subtle, especially in the presence of polymorphism; we do not tackle this problem. We also assume that polymorphic functions do not use continuations. When applying contracts to lists (or the user-defined data structures of PLT Scheme), these lists must be immutable. There have recently been proposals to make lists immutable by default in both PLT Scheme and the Scheme standard, based on the experience that mutable lists are, in fact, rarely mutated. Therefore, we do not consider this assumption to be unreasonable. The second caveat is that we do not formally prove that our system ensures relational parametricity; this is left open for future work.

Status The work in this paper has been implemented for PLT Scheme and JavaScript. We will use the Scheme version for the presentation, mainly because Scheme's notational extensibility enables us to provide most of the implementation in the paper, including the syntactic conveniences. Near the end, we will briefly discuss the JavaScript implementation. We make extensive use of these contracts in the Flapjax implementation where, for instance, the blame facilities described in section 5 have proven immensely useful, especially in face of the poor debugging support currently on offer for JavaScript.

2. Background: Contracts

In this paper we use two models of contracts. The first model (section 2.1), which represents contracts as functions, is sufficient for describing most of the work in this paper. Recent research has, however, shown that a model based on

[1] In the rest of this paper, we will frequently elide the modifiers *relational* and *parametric*.

a *pair* of functions provides a better account of how contracts ascribe blame. We present this model in section 2.2. Because this model is used only in section 5, most readers may safely ignore it on their first reading.

2.1 A Simple Model of Contracts

A flat contract simply applies its predicate to each value, raising an error when the test fails:

```
(define (flat pred)
  (λ (val)
    (if (pred val)
      val
      (error "contract violation"))))
```

A function contract is build from two other contracts, one for the argument and the other for the result. A function is guarded by wrapping it in a proxy that applies these contracts to every argument and result (the initialism *ho* is short for *higher-order*):

```
(define (ho dom rng)
  (λ (v)
    (if (procedure? v)
      (λ (x) ; the proxy
        (rng (v (dom x))))
      (error "not a function"))))
```

Note that the contracts that result from applying *flat* and *ho* accept a value, *v*, and either signal an error or return a value that behaves exactly like *v*. If *v* is a function, the returned function may signal errors more often than *v*, but otherwise behaves exactly as *v*.

Because contracts are functions, to guard a value, we simply apply the contract to the value. For example:

```
(define guarded-5 ((flat number?) 5)
```

ensures that 5 is a number, while

```
(define guarded-increment
  ((ho (flat? number) (flat? number))
   (λ (n) (+ n 1))))
```

constructs a guarded increment function that ensures both its input and output are numbers.

When obvious, we will elide (*flat pred*) to simply write *pred*. Additionally, we will write (*ho dom rng*) using the more standard infix notation, *dom* → *rng*.

2.2 Contracts as Pairs of Projections

An error projection is a function that returns its argument unmolested, with the exception that it may signal an error on some inputs. Note that the contracts of section 2.1 are precisely error projections. This model is, however, oversimplified, because it does not track blame. The utility of contracts hinges on their ability to correctly blame the supplier of an invalid value.

Findler and Blume [7] present an alternate model of contracts, treating them as pairs of error projections. They are parameterized over the names of the guarded value and its context. One projection, the server projection, blames the guarded value when appropriate. The other projection, the client projection, blames the context of the guarded value, when appropriate.

Concretely, we define the contract datatype to have two fields, for each projection:

```
(define-struct contract (server client))
```

This employs PLT Scheme's support for user-defined records. It creates a fresh type, which has a predicate, *contract?*, a two-argument constructor, *make-contract*, and the selectors *contract-server* and *contract-client*.[2]

To apply a contract to value, we define a generic guard function that accepts a contract, the value to guard, and names for the guarded value and its calling context:

```
(define (guard ctc val pos neg)
  (let ([server-proj ((contract-server ctc) pos)]
        [client-proj ((contract-client ctc) neg)])
    (client-proj (server-proj val))))
```

Consider the combinator *flat*. *flat* accepts a predicate, which it applies to the guarded value, and blames the supplier of the value (the server) if the contract is violated. Note that *flat* never blames the context (the client). Therefore, the client projection is the identity function:

```
(define (flat pred)
  (make-contract
    (λ (s) ; name of the server
      (λ (v)
        (if (pred v)
          v
          (blame s))))
    (λ (s) ; name of the client
      (λ (v) v))))
```

The function contract combinator, *ho*, is more interesting. In the client and server projections, *ho* wraps the incoming function, *val*, in order to apply the error projections of the domain and range contracts. If the argument to *val* violates the domain contract, *dom*, we blame the context. This is captured by using the server projection of *dom* in the client and the client projection of *dom* in the server. However, if the supplied value, *val*, is not a function, *ho* blames the supplier (the server) for providing an invalid value.

```
(define (ho dom rng)
  (make-contract
   (λ (s)
     (let ([dom-c ((contract-client dom) s)]
           [rng-s ((contract-server rng) s)])
```

[2] PLT Scheme defines a few other operations as well, but these do not concern us here.

```
    (λ (val)
      (if (procedure? val)
          (λ (x) (rng-s (val (dom-c x))))
          (blame s)))))
  (λ (s)
    (let ([rng-c ((contract-client rng) s)]
          [dom-s ((contract-server dom) s)])
      (λ (val)
        (if (procedure? val)
            (λ (x) (rng-c (val (dom-s x))))
            val))))))
```

This definition of *ho* only works with unary functions, but extending it to work with n-ary functions is easy.

To summarize, this contract system consists of three principal functions:

flat	:	$(\alpha \rightarrow bool) \rightarrow contract\ \alpha$
ho	:	$contract\ \alpha \times contract\ \beta$
		$\rightarrow contract\ (\alpha \rightarrow \beta)$
guard	:	$contract\ \alpha \times \alpha \times sym \times sym \rightarrow \alpha$

3. Two Attempted Solutions

To provide some intuition for the problem and for our solution, we first sketch two approaches to implementing parametric contracts. Neither of these will work, but they help put us on a path to the actual solution. Readers already familiar with the concepts of parametric types and parametricity might wish to skip directly to the solution in section 4, but other readers may find that the more gradual development better builds their understanding.

3.1 Using Substitution

In System F, a value of a universally quantified type, $\forall\alpha.\tau$, is a function over types—a *type abstraction*. A type abstraction, $\Lambda X.t$, accepts an arbitrary type, X, and returns a term whose type is parameterized over X. We consider here an approach to parametric contracts that mimics type abstraction and type application. In particular, we will define (**forall** (α ...) *C*) as a function over contracts. Such a function would accept concrete contracts for the contract variables, (α ...), and substitute them into the contract *C*. Defining this function is straightforward, especially with the help of Scheme's macros:

```
(define-syntax forall
  (syntax-rules ()
    [(_ (α ...) C)
     (λ (α ...) C)]))
```

Since we define this analogue of type abstractions as a Scheme function, type applications are simply applications to concrete contracts. Hence, if *T* is a contract, the expression ((**forall** (α ...) *C*) *T*) is an *instantiation* of the quantified contract (**forall** (α ...) *C*). If we have a function *f* guarded by a parametric contract (**forall** (α) *C*), the guarded function may be written as:

```
(define (guarded-f ctc)
  (((forall (α) C) ctc) f))
```

That is, *guarded-f*, when applied to a contract *ctc* returns *f* guarded by (**forall** (α) *C*). Scheme's module system can generate this boilerplate code for us. however, we do need to supply the concrete contract for α at each application.

However, this naive definition of (**forall** (*a* ...) *C*) fails to capture parametricity. Consider the parameterized contract:

(**forall** (α) $\alpha \rightarrow \alpha$)

This contract may be instantiated by applying it to any concrete contract. For example, we may instantiate α to *number?*:

((**forall** (α) $\alpha \rightarrow \alpha$) *number?*)

What functions satisfy this contract? Obviously, we may guard the identity function with this contract, to get the identity function over numbers. In fact, we may guard the identity function with the contract ((**forall** (α) $\alpha \rightarrow \alpha$) *C*), for arbitrary contracts *C*. It turns out that there are other functions we can successfully guard with this contract. Consider the following function:

```
(define (inc-or-id x)
  (if (number? x)
      (+ x 1)
      x))
```

This function satisfies the contract ((**forall** (α) $\alpha \rightarrow \alpha$) *number?*). (Our current implementation of **forall** simply substitutes *number?* for α, which is *number?* $\rightarrow$ *number?*.) However, *inc-or-id* is not parametric and breaks the abstraction we wish to create. If we change the representation of its argument—that is, if we instantiate α to a contract that excludes numbers, *inc-or-id* behaves as the identity function. However, if *x* is a number, *inc-or-id* behaves as the increment function. Note that unlike the identity function, *inc-or-id* does not satisfy all contracts ((**forall** (α) $\alpha \rightarrow \alpha$) *C*) for arbitrary contracts *C*. For example, let

C=(λ (*v*) (**and** (*number? v*) (= *v* 5)))

Evaluating (*inc-or-id* 5) violates the contract.

The problem with *inc-or-id* ostensibly lies with its use of reflection (*number?*). However, consider the following function:

```
(define (const-five x) 5)
```

We may guard *const-five* with the contract (**forall** (α) $\alpha \rightarrow \alpha$) as well. As long as α is instantiated to a contract that accepts the value 5, *const-five* will not violate the contract. However, the fact that the function fails on any other contract shows that it is not agnostic to the choice of α, despite being a seemingly innocuous function.

These two functions readily break the abstraction we wish to create. However, we cannot exclude them statically, as we are interested in dynamically typed languages such as Scheme and JavaScript. Therefore, we need a more robust implementation of parametric contracts that can enforce parametricity and fail gracefully when a function attempts to violate parametricity.

3.2 Using Identity

Given an ML function with type $\forall\alpha.\tau_1 \rightarrow \tau_2$, an intuitive understanding of parametricity states that the function cannot inspect or construct values of type α. This suggests that we should try to use the identity of values by recording them at entry and checking them at egress. It is natural to implement this strategy using, say, a hashtable to store these values. That is, if f were guarded by the contract (**forall** (α) $\alpha \rightarrow \alpha$), we'd create a hashtable for the contract α; arguments guarded by α would get added to the hashtable and results guarded by α would be checked to ensure they were in the hashtable.

PLT Scheme used to provide a mechanism to do this, known as *anaphoric contracts*:

```
(provide/contract  ;; written in PLT Scheme v209
 (f (let-values ([(α_in α_out) (anaphoric-contracts)])
      (α_in → α_out))))
(define (f x) x)
```

α_{in} and α_{out} are a pair of contracts representing a source and a sink. This pair shares a common hashtable, which they use to track the flow of values.

Now consider the type $\forall\alpha.(\alpha \rightarrow \alpha) \times \alpha \rightarrow \alpha$. The following function satisfies this type:

```
(define (app1 f x)
  (f x))
```

Using the hashtable approach, we may try to construct the corresponding contract:

```
(provide/contract
  (app1 (let-values ([(α_in α_out) (anaphoric-contracts)])
          ((α_in → α_out) α_in → α_out))))
```

Though this may look intuitively correct (the same α guards the *x* and the argument position of *f*), it does not correctly capture the intended semantics at all. We want to ensure that all the values that flow into the argument position of *f* came via *x*, and those returning from *app1* are from the result of *f*. To clarify these connections, we can introduce a pair of contract pairs:

```
(provide/contract
  (app1 (let-values ([(α_in α_out) (anaphoric-contracts)]
                     [(β_in β_out) (anaphoric-contracts)])
          ((α_out → β_in) α_in → β_out))))
```

While this correctly captures the local flows of values, it has now lost the connection expressed in the type: namely, that the type of the value that *f* consumes is the same as the type of value that it produces. This leads to another revision:

```
(provide/contract
  (app1 (let-values ([(α_in α_out) (anaphoric-contracts)])
          ((α_out → α_in) α_in → α_out))))
```

This use of contracts properly represents the desired relationship. The key idea here—that values produced must be the same as the values consumed—is significant, and represents what we want to ultimately capture. It is, however, clear that this particular implementation strategy has several problems:

- First, it is unreasonable to demand this level of dataflow analysis on the part of the programmer; we would prefer to write a specification similar to the type and leave it to the system to generate these connections.
- Second, the run-time cost associated with this implementation was frequently onerous.
- Third, in languages that do not permit equality comparison of higher-order values, this strategy cannot be used at all.
- Fourth, the equivalent of instantiating a polymorphic function must be performed manually.

In fact, anaphoric contracts can't even capture the relationships we can express using a proper definition of parameterized contracts instantiated with the *any* contract, as shown in section 7.4.

4. Parametric Contracts (At Last)

Our approach to parametric contracts will define (**forall** (α ...) *C*) as a function over contracts, in the spirit of section 3.1. Hence, it will be necessary to supply concrete contracts for the variables (α ...) at each usage point. Our earlier definition of **forall** as a function permitted non-parametric functions. We will rectify this by borrowing ideas from anaphoric contracts.

Anaphoric contracts tried to ensure that α-values produced by a guarded function were those α-values sent as arguments to the function. It did so by relying on pairs of in and out contracts that were used in consumer and producer positions. We will employ this technique to guarantee this desired property.

Our approach will be to wrap α-values consumed by a function in an opaque container and subsequently unwrap α-values that are produced by the function. We call such opaque containers *coffers*, which we can define as a record:

```
(define-struct coffer (value))
```

Using the module system, we can make the functions *coffer?*, *make-coffer* and *coffer-value* unavailable to user code, making coffers truly opaque.

Since we can make values opaque, given a contract α, we can simultaneously define wrapper-unwrapper pairs:

```
(define-struct parametric-pair (wrapper unwrapper))
(define wrapper parametric-pair-wrapper)
```

```
(define unwrapper parametric-pair-unwrapper)

(define (make-parametric α)
  (make-parametric-pair
    (λ (val)
      (make-coffer (α val)))
    (λ (opaque-val)
      (if (coffer? opaque-val)
          (coffer-value opaque-val)
          (error "expected a coffer")))))
```

This definition actually accomplishes two different things. The use of coffers implements parametricity by preventing the contracted function from examining *val*. It *also* implements parameterization by checking (in the boxed expression) that the calling context supplies values that obey α. Excluding one of these features results in a more relaxed notion of contracts. Using only coffers ensures only that the contracted expression behaves parametrically, while employing just the application of α ensures only that the context supplies values that match α.

This definition yields a pair of contracts, but does not stipulate how to use them. Recall that anaphoric contracts required the programmer to explicitly insert in and out contracts appropriately. For parametric contracts, we will mechanically insert wrapping and unwrapping contracts. Note that in the contract $\alpha \rightarrow \alpha$, the α to the left of the arrow guards arguments to the function, and hence wraps values. The α to the right of the arrow guards results, and hence unwraps values. Now, consider the contract $(\alpha \rightarrow \alpha) \rightarrow \alpha$. Here, the leftmost α unwraps arguments it receives from the guarded function. The center α guards results that are returned to the function, so it must wrap them. The rightmost α guards the result of the function, which it must unwrap.

At each nesting level of $\rightarrow$, the wrapper and unwrapper swap positions. Alternatively, an α to the left of an even number of arrows (positive position) is always an unwrapper. An α to the left of an odd number of arrows (negative position) is always a wrapper. (These positions correspond exactly with positive and negative blame assignment. We will exploit this correlation later when we add support for blame.) For now, we define **forall** using an auxiliary macro that tracks positive and negative positions:

```
(define-syntax forall
  (syntax-rules ()
    [(_ (var ...) contract)
     (λ (var ...)
       (let ([var (make-parametric var)] ...)
         (forall-aux (var ...) contract #f)))]))
```

The boolean argument to **forall-aux** is true when it is wrapping values and false when it is unwrapping values:

```
(define-syntax forall-aux
  (syntax-rules (→)      ;; treat → as a keyword
    [(_ (α ...) (→ dom rng) #t)
     (ho (forall-aux (α ...) dom #f)
         (forall-aux (α ...) rng #t))]
    [(_ (α ...) (→ dom rng) #f)
     (ho (forall-aux (α ...) dom #t)
         (forall-aux (α ...) rng #f))]
    [(_ (α ...) ctc #t)
     (let ([α (wrapper α)] ...)
       ctc)]
    [(_ (α ...) ctc #f)
     (let ([α (unwrapper α)] ...)
       ctc)]))
```

This setup for parametric contracts is almost correct. Consider, however, the contract (**forall** $(\alpha\ \beta)$ $(\alpha\ \beta \rightarrow \alpha)$). The following function fails to satisfy this contract:

```
(define (f x y) y)
```

However, if we instantiate α and β to the same contract, the function is admitted. For example, if both variables are instantiated to *number?*, the expression (*f* 5 10) evaluates to 10, even though *f* did not consume an α-value 10.

We fix this by having each instance of *make-parametric* create a new coffer. This is easily accomplished in PLT Scheme:

```
(define (make-parametric α)
  (define-struct coffer (value))
  (make-parametric-pair
    (λ (val)
      (make-coffer (α val)))
    (λ (opaque-val)
      (if (coffer? opaque-val)
          (coffer-value opaque-val)
          (error "expected a coffer")))))
```

This redefinition exploits the fact that structures in PLT Scheme are *generative*; that is, *each* **define-struct** makes a new type. By placing the structure definition inside the function, the coffers from different invocations of *make-parametric* cannot be commingled. In a language without this facility, the implementation would need to simulate the generativity by, for instance, having an extra tag in the *coffer* structure that records which instance it represents.

This clarifies why the previous *make-parametric* was "almost correct": it failed to keep different instances of contract variables separate. We now have truly opaque values, since we can distinguish concrete values that are identical but guarded with different contracts.

Note that we have defined parametric contracts as functions over contracts. Therefore, we do not guard values with parametric contracts per se; we guard values with the contracts that result from applying parametric contracts. For example, consider the contract (**forall** (α) (*listof* α) $\rightarrow$ (*listof* α)). Suppose we wish to guard the *reverse* function with this contract, and call the result *c-reverse*. The most natural way to do so is to let *c-reverse* accept the concrete contract for α as an additional, curried argument:

```
(define (c-reverse ctc)
  (((forall (α) (listof α) → (listof α)) ctc)
   reverse))
```

c-reverse applies *ctc* to the parametric contract. The resultant contract is applied to *reverse* (as in section 2.1). This code is rather onerous; fortunately, it can be generated automatically using the macro system [10]. When *applying* a function guarded by a parametric contract, however, programmers must supply concrete contracts for the contract-variables.

One design decision involves the application of type-predicates (*number?*, *string?*, etc.) to wrapped values. By default, applying a primitive Scheme type-predicate to a wrapped value will return false, as each *coffer* is a new type. This changes the behavior of certain programs.

Consider *inc-or-id* that we encountered in section 3.1. We guarded this function with (**forall** (α) $\alpha \to \alpha$), with α instantiated to *number?*. Evaluating (*inc-or-id* 5) wraps the argument, so (*number? x*) evaluates to false, and *inc-or-id* behaves as the identity function on numbers. It is reasonable to claim that since *x* is guarded by a parametric contract, we should never have attempted to determine its type—that is, applying a type-predicate to a wrapped value should raise an error. We can implement this behavior by defining a language [10] where the type predicates raise an error when applied to wrapped values, and behave as they do in Scheme on other values.

This completes the description of parametric contracts, but so far we have not yet given an account of lifting the notion of blame to the parametric context.

5. Parametric Contracts with Blame

We now complete our presentation of parametric contracts by adding blame-tracking. For this, we adopt the pair-of-projections model summarized in section 2.2. We begin by encoding the wrapper and unwrapper contracts produced by *make-parametric* as pairs of error projections. The template for doing so is:

```
(define (make-parametric α)
  (define-struct coffer (value))
  (make-parametric-pair
    (make-contract   ; wrapper
      (λ (s)
        (λ (v)  ; server projection
          ...))
        (λ (s)
          (λ (v)  ; client projection
            ...)))
      (make-contract   ; unwrapper
        (λ (s)
          (λ (v)  ; server projection
            ...))
        (λ (s)
           (λ (v) ; client projection
                        ...)))))
```

Consider the wrapper contract. Note that it is only used in positive positions. In positive positions, the guarded value is produced by the context. Hence, for a wrapper the name *s* names the context in its server projection and the guarded function in its client projection. The task of the wrapper is to apply the concrete contract α, which itself consists of a server and a client projection. Hence, we apply each of α's projections in each projection of the wrapper:

```
(make-contract  ; wrapper
  (λ (s)
    ; server projection
    ; (parameterized over context's name)
    (λ (v)
      (make-coffer (((contract-server α) s) v))))
  (λ (s)
    ; client projection
    ; (parameterized over function's name)
    (λ (v)
      (if (coffer? v)
        (make-coffer
          (((contract-client α) s)
           (coffer-value v)))
        v))))
```

Now consider the unwrapper contract. Since the concrete contract α is applied when values are wrapped, the unwrapper simply unboxes values. The only error that may occur is that the incoming value may not be wrapped. If this is the case, the guarded function is to blame for producing an invalid result. Since unwrappers are only used in negative positions, the server projection is parameterized over the name of the guarded function. Therefore, we blame the server (the guarded function) if the value is not wrapped:

```
(make-contract  ; unwrapper
  (λ (s)
    ; server projection
    ; (parameterized over the function's name)
    (λ (v)
      (if (coffer? v)
        (coffer-value v)
        (blame s))))
  (λ (s)
    ; client projection
    ; (parameterized over the context's name)
    (λ (v) v))))
```

6. Contract Inference

Parametric contracts, as described above, require programmers to explicitly instantiate them. This unfortunately means that programs that make heavy use of parametric functions get littered with instantiations. Is it possible to offer a more lightweight technique that infers the contract at run-time, in the spirit of type inference?

The answer depends on what we consider "the" contract of a value to be. A language like ML has a *partitioned* type system, meaning that every value belongs to precisely one type. In contrast, because a contract is just a predicate, a value can satisfy many different contracts at once: for instance, it may be a number, an odd number, and a prime number. Furthermore, the contracts may not satisfy any grouping that enables us to choose a canonical element.

Rather than tackle the problem of inference in its generality, we therefore perform inference over the primitive types of the language, which do provide a partitioning of values. We believe this is sufficiently useful in many circumstances. In situations where the programmer wants to instantiate a parameter with a particular, non-primitive-type, contract, this is always possible using the mechanism described in section 4.

We implement contract inference by creating an indeterminate contract that can be used to instantiate a parameterized contract. This contract is stateful: it remembers the primitive type of the first value is sees, and then checks for conformity with that type for all subsequent values. For flat values, this behavior is very simple:

```
(define (make-var-contract)
  (let ([contract #f])
    (λ (val)
      (unless contract
        (cond
          [(number? val) (set! contract number/c)]
          [(boolean? val) (set! contract bool/c)]
          ; ... and similarly for other flat values
          [(procedure? val) ... ]))
      (contract val))))
```

The case for procedures is more interesting. Given that *val* is a procedure, all we can do is assert that subsequent values are procedures as well. However, when *val* (or a subsequent value) is applied, we can attempt to infer the contract on its domain and range. We use a helper function

```
                          ⋮
[(procedure? val) (set! contract (make-proc-var-contract))]
                          ⋮
```

that is defined as follows:

```
(define (make-proc-var-contract)
  (let ([dom-c (make-var-contract)]
        [rng-c (make-var-contract)])
    (λ (f)
      (if (not (procedure? f))
        (error "contract violation ...")
        (λ (x)
          (rng-c (f (dom-c x))))))))
```

This first asserts that the value *f* is a procedure. Given that it is, it wraps *f* to check argument and result. Note that *dom-c* and *rng-c* are shared by all values that a particular procedure contract is applied to. This ensures that they all have the same domain and range. (Note the close similarly to unification, with the indeterminate contracts behaving as logic variables.)

7. Illustrations

We will work through a few examples of parametric contracts and contract inference to examine the implications of our definitions.

7.1 Simple Parametric Contracts

Consider the contract (**forall** (α) $\alpha \rightarrow \alpha$), which is based on the type $\forall\alpha.\alpha \rightarrow \alpha$. Parametricity ensures that a value of this type is either divergent, raises an error, or is the identity function. We sketch an argument that a value (i.e., the result of a non-erroneous computation that has terminated) that satisfies this contract must be the identity function.

It is clear that the identity function satisfies the contract (**forall** (α) $\alpha \rightarrow \alpha$). Let *f* be an arbitrary function guarded by this contract. Consider what *f* may do with its argument that is guarded by the contract α. By definition of *make-parametric*, parameters to *f* are wrapped in a coffer representing α. Since there is no way for *f* to inspect or construct new α values, it is clear that *f* cannot manipulate them.

Now, consider the contract α on the result of *f*. This contract unwraps values from coffers. Because fresh coffers are created by each invocation of *make-parametric*, this unwrapping contract accepts only those values wrapped by its dual wrapping contract. The only source of wrapped α-values is the argument to *f*. Hence, if *f* satisfies the contract (**forall** (α) $\alpha \rightarrow \alpha$), it must simply return its argument. Hence, *f* must be the identity function.

Let's revisit the *inc-or-id* function we encountered in section 3.1. This function is not parametric, but was guarded by the parametric contract, (**forall** (α) $\alpha \rightarrow \alpha$). When α is instantiated to *number?* and *inc-or-id* is applied to a number, *n*, the wrapping contract on the argument *x* places *n* in a coffer. Therefore, the test (*number? x*) raises an error, indicating that the program attempted to inspect the type of an opaque value.

The function *inc-or-id* attempted to inspect and branch on a parametric argument. In contrast, recall the function *const-five* from section 3.1. We had attempted to guard this function with (**forall** (α) $\alpha \rightarrow \alpha$) as well. If we instantiate α to a contract that excludes the value 5, the contract will naturally be violated. Suppose, instead, α is instantiated to *number?*. Since the result is guarded by α, the function is expected to return a wrapped *number*. However, *const-five* returns an unwrapped number, so applications such as (*const-five* 7) violate the contract. In fact, (*const-five* 5) violates the contract as well! Although the contract on the argument wraps the value 5, the function attempts to return an unwrapped 5.

7.2 Unsatisfiable Contracts

Consider the parametric contract (**forall** (α) α), which is derived from the type $\forall\alpha.\alpha$. There cannot be any values of this type, because no value can satisfy every possible type (i.e., parametrically, this type is uninhabitable). We show that no values satisfy the corresponding contract as well.

For simplicity, consider our model of contracts without blame. By expansion of **forall** and **forall-aux** (section 4), the contract (**forall** (α) α) is implemented as:

```
(λ (α)
  (let ([α (make-parametric α)])
    (unwrapper α)))
```

The application (*make-parametric* α) constructs a related pair of wrapper-unwrapper contracts. The above expression evaluates to (*unwrapper* α), so the values of this contract are those values that (*unwrapper* α) accepts. (*unwrapper* α) only accepts values guarded by its dual, (*wrapper* α). At no point in the code is (*wrapper* α) applied. The contract, therefore, rejects all values, preserving the property of the corresponding type.

7.3 Arbitrary Predicate Contracts

The freedom to instantiate **forall** contracts with arbitrary contracts is valuable. First, it enables us to specify fine-grained constraints on values. For instance, let *c-reverse* be the *reverse* function guarded by the contract:

```
(forall (α) (listof α) → (listof α))
```

Since *c-reverse* is guarded by a parametric contract, it requires a concrete contract for α on each invocation, which it takes as an extra, curried argument. For example:

```
((c-reverse string?) '("xyz" "abc" "123"))
((c-reverse positive?) '(1 2 3 4))
((c-reverse (odd? → even?)) (list add1 sub1))
```

As these examples show, we can specify properties that are finer-grained than those specifiable in a language like ML.

One of the benefits of contracts is that they can be introduced incrementally to existing programs. Therefore, it is conceivable that a function guarded with a parametric contract may be invoked by Scheme code that freely uses heterogeneous lists and other informal data structures. Programmers who can create predicates that capture their data structures can supply these using explicit instantiation. For example, the following instantiation of *c-reverse* accepts numbers and false:

```
((c-reverse (λ (x) (or (number? x) (false? x))))
            '(1 #f 3 7))
```

This is a common idiom for expressing the *Int Option* type of ML.

7.4 The *any* Contract

While incrementally adding contracts, a programmer may not always be able to express the structure of data in a contract (or may not want to). For instance, consider the heterogeneous data structures that are pervasive in traditional Scheme programs, such as s-expressions. In such cases, the programmer can use the *any/c* contract:

```
((c-reverse any/c)
 '((p "paragraph")
   (span ([id "block"])
     "text")))
```

In this context the underlying implementation of parametric contracts still ensures that the function cannot manufacture values, i.e., it ensures the proper "wiring" of values. It does not, however, place any stricter restrictions on the nature of the values. Therefore, *any/c* provides weaker guarantees than contract inference (section 6).

There is another use for *any/c*: it enables a function to return a wrapped value without unwrapping it. For example, consider guarding the identity function with the contract (**forall** (α) $\alpha \to$ *any/c*). When applied, the argument is wrapped by the α-wrapper (assuming the argument satisfies the contract α). The result, however, is not unwrapped, because the contract is *any/c*. Therefore, when guarded with this contract, *id* returns an opaque, wrapped value.

The data abstraction provided by Scheme's **define-struct** mechanism ensures that this wrapped value cannot be unwrapped by code that does not have access to the corresponding selector. This would suggest that returning a wrapped value is useless. The encapsulated value can, however, be used if it comes with appropriate selectors. For instance, consider the following fragment of code:

```
(define (make-hide-show rep/c)
  (let ([ctc (forall (a) (seq (→ a any/c)
                              (→ any/c a)))])
    ((ctc rep/c)
     (list
       (λ (x) x)
       (λ (x) x)))))
```

where *seq* is a contract for a heterogenous list. *make-hide-show* creates an abstract container that holds values of the type specified in *rep/c*, e.g.,

```
(define hs (make-hide-show (flat number?)))
(define hide (first hs))
(define show (second hs))
```

Values created by *hide* can only be opened using the corresponding *show*, with the invariant that (*show* (*hide* n)) = n. Each application of *make-hide-show* creates a distinct constructor and selector pair that cannot be interchanged. Thus, this defines an analog to an existential type abstraction [20].

7.5 Contract Inference

The *c-reverse* function required the programmer to explicitly specify the concrete contract for α each time *c-reverse* was invoked. For example:

((*c-reverse number?*) '(1 2 3 4))
((*c-reverse prime?*) '(2 3 5 7))
((*c-reverse bool?*) '(#t #f #t #t))

Contracts such as *prime?* must be specified by the user, but basic contracts such as *number?* and *bool?* can be inferred by our system. Consider the following:

(**define** *i-reverse*
  (*c-reverse* (*make-var-contract*)))

Suppose we apply *i-reverse* to a list of functions:

(**define** *fns* (*i-reverse* (*list* (λ (*x*) (+ *x* 1))
                          number→string
                          (λ (*x*) ($\times$ *x x*)))))

Since the first element of the list of a function, contract inference binds the variable α to the contract $\alpha_1 \rightarrow \alpha_2$, where α_1 and α_2 are indeterminate contracts. Since all three elements of the list are procedures, each function is sucessfully guarded by $\alpha_1 \rightarrow \alpha_2$. It is important to note that the contracts α_1 and α_2 are shared by the three functions.

Suppose we apply the first function, as in, ((*car fns*) 5). The value 5 is guarded by α_1. Since α_1 is indeterminate, it assumes the *number?* contract. Similarly, the result 6 is guarded by α_2, which assumes the *number?* contract. Now, all three functions are guarded by the contract *number?* → *number?*. Note that the second function, *number→string*, does not satisfy this contract. The application ((*cadr fns*) 5) would thus raise an exception. In particular, the calling context of *reverse* would be blamed for supplying a function that violates the inferred contract.

We do get different results if we first apply the function *number→string*, i.e., ((*cadr fns*) 7). Because the contracts α_1 and α_2 are indeterminate, they assume the contracts *number?* and *string?* respectively. Therefore, applying (*car fn*) and (*caddr fn*) now violates the contract.

8. Contracts for JavaScript

Due to the pervasive growth of rich-client Web applications, we have been exploring the adaptation of this contract library from PLT Scheme to JavaScript [3]. Our JavaScript contract library is built with the following core functions:

$$\begin{array}{rcl} \mathit{flat} & : & (\mathit{any} \rightarrow \mathit{bool}) \times \mathit{string} \rightarrow \mathit{contract} \\ \mathit{func} & : & \mathit{contract} \times \mathit{contract} \rightarrow \mathit{contract} \\ \mathit{args} & : & \mathit{contract} \ldots \rightarrow \mathit{contract} \\ \mathit{guard} & : & \mathit{contract} \times \mathit{any} \times \mathit{string} \times \mathit{string} \end{array}$$

The *flat* combinator creates a contract based on a predicate and a string that describes the predicate. *func* creates a function contract. The domain contract is applied to the array of arguments. The *args* combinator is a variable-arity function that maps n contracts to a contract on an array of n arguments. The *guard* function guards a value with a contract and accepts labels for the calling context and the guarded value.

Contracts are represented as objects whose properties include the server and client projections. In JavaScript, however, public properties, even methods (which are merely references to functions), can be mutated at any time. Hence, the modular encapsulation present in PLT Scheme is absent in JavaScript, greatly compromising our ability to provide any guarantees about contracted functions. Nevertheless, we still find such contracts useful for stating program invariants and for finding errors; we cannot, however, rely on them to establish any mission-critical properties (such as ensuring certain kinds of security).

To properly provide parametric contracts for JavaScript, we must address three major design and implementation issues. We discuss these in the following sections.

8.1 Encapsulation

JavaScript's lack of encapsulation makes it impossible to create opaque coffers. The obvious solution is to create a constructor for coffers that stores the wrapped value in a property. However, properties can always be read and modified. Furthermore, any attempt to mangle the name of a property is also futile, as JavaScript makes it trivial to iterate over all the properties of an object. More creative solutions may involve a private field and a privileged getter method:

```
var Coffer = function(v) {
  var that = this;
  var value = v;
  this.getValue = function() {
    return that.value;
  }
}
```

However, external code may replace the privileged `getValue` method on an object.

Instead of using creative techniques for obfuscation—as opposed to opacity—we represent coffers very simply as objects with a property referencing the enclosed value:

```
var Coffer = function(val,tag) {
  this.val = val;
  this.tag = tag;
}
```

Hence, a sufficiently ill-behaved script can violate all guarantees provided by our contracts. However, our experience with JavaScript suggests that the bulk of real code is not so ill-behaved.

8.2 Notation

Writing parametric contracts involves using wrapping and unwrapping contracts at appropriate places. The **forall** macro of section 4 takes a parametric contract that looks like a type and mechanically inserts wrapping and unwrapping contracts as appropriate. This is done statically by the macro-expander. Since JavaScript does not support syntactic extensibility, we are forced to consider alternatives.

Currently, we must force the JavaScript programmer to simulate the job of the **forall** macro. That is, while in Scheme we can write (**forall** (α β) (($\alpha \rightarrow \beta$) (*listof* α) $\rightarrow$ (*listof* β)), the corresponding JavaScript code is:

```
function cMap(alpha,beta) {
  var alphaP = parametricPair(alpha);
  var betaP  = parametricPair(beta);

  return func(args(func(args(alphaP.unwrap),
                           betaP.wrap),
                     listof(alphaP.wrap)),
               betaP.unwrap);
}
```

Not only is this verbose, it requires the programmer to be keenly aware of the flow of values, which was one of the issues with anaphoric contracts in section 3.2.

Another alternative would be to construct wrappers and unwrappers dynamically at runtime. Doing so, however, adds complexity to the entire contract library. All contracts would be required to propagate information to the parametric contracts.

Our current approach is to build the equivalent of **forall** into the compiler for Flapjax. The Flapjax suite includes a compiler that automatically augments JavaScript code with invocations of procedures such as `lift` and `switch` (described in section 1), relieving the programmer of this burden. Flapjax is semantically compatible [13] with code imported from JavaScript, so programmers can use predicates written in JavaScript as contracts in Flapjax.

8.3 Inference

Unlike Scheme, where the base types are partitioned by a set of well-defined predicates, JavaScript's prototype-based object system supports subtyping. Therefore, we cannot rely on a partition of values as we can in Scheme. We could partition JavaScript objects by their prototypes:

```
function objTypeEq(x,y) {
  return x.__proto__ == y.__proto__;
}
```

However, this approach neglects subtyping, which is present in the language, and behavioral subtyping, which is a common idiom in JavaScript. We therefore leave this topic open for further investigation.

9. Related Work

The idea of boxing and tagging polymorphic values with types is well-established. Harper and Morrisett [11] use runtime tags to specialize polymorphic functions for efficiency. Our generative coffers are runtime tags, but we use them to enforce parametricity and not for specialization.

Matthews and Findler [15] introduce techniques for embedding latently-typed languages (such as Scheme) into statically-typed ones (such as ML). Their operational semantics enables them to focus on high-level cross-language properties such as type-safety. In particular, they achieve safety across language boundaries by using contracts to ensure that Scheme code doesn't misuse ML code. However, since their contracts cannot capture parametric polymorphism, their dialect of ML is monomorphic.

Tobin-Hochstadt and Felleisen [27] incrementally add type annotations to Scheme programs. In their work, unlike in Matthews and Findler, the runtime semantics and values of both languages are the same, but the typed modules obey a static type discipline. They too use contracts to prevent raw Scheme code from abusing Typed Scheme. Their Typed Scheme language is, however, monomorphic, as Scheme's contracts cannot capture parametricity.

Parametric polymorphism is, in essence, a form of information hiding. Zdancewic, et al. [29] add principals to the simply-typed λ-calculus, creating a calculus for multi-agent computations. Our approach to encoding polymorphism is similar to their idea of encoding a polymorphic function, $\Lambda\alpha.e$, as an agent oblivious to α. When such a function is applied, $\Lambda\alpha.e[\tau]$, another agent that is aware of the concrete type, τ, provides unwrapping information. They can, however, rely on a type system to prevent a polymorphic function from violating the type abstraction, whereas we have to wrap values since our functions are not statically typed.

Pierce and Sumii [21] present the *cryptographic λ-calculus* as a model for reasoning, using parametricity, about programs that use encryption for information-hiding. Our technique for preserving parametricity is similar to their encoding of type abstraction with encryption and decryption (section 4.2 of their paper). Their primary concern, however, is with incoming, encoded values being malicious, whereas we must also protect against functions that violate their proclaimed parametric contract. We believe that proving parametricity for our system is related to proving their Full Abstraction Conjecture correct.

10. Conclusion and Future Work

We have presented a contract analog of parametrically polymorphic types. Our technique is based on a judicious use of opaque wrappers to keep values from being inspected or created by functions guarded by such contracts. We also show a form of inference for primitive contracts.

There are many areas for extension and application of this work. First, we have given only an intuition, not formal proof, that our work preserves parametricity. Second, our notion of inference is implicitly parameterized over what it means to be "the same type". We can relax this notion to permit unions, subtypes, and so forth. This would give rise to more liberal notions of inference. Third, the process of inference essentially "learns" the contract of a parameter, then checks it on subsequent applications. Rather than check, we can accumulate all the learned contracts, and use the accumulated set (or bag) for post-hoc analyses. Finally, by altering the strategy of opaque wrapping, we can enable different kinds of polymorphism, such as ad-hoc polymorphism.

Acknowledgments

We are grateful to John Peterson, Philip Wadler, William Cook, David Herman, Sam Tobin-Hochstadt, and Erik Meijer, for enlightening conversations. This work is partially supported by NSF grants CCF-0447509, CNS-0627310, IIS-0613805, and CCF-0429595.

References

[1] Carrillo-Castellon, M., J. Garcia-Molina, E. Pimentel and I. Repiso. Design by contract in Smalltalk. *Journal of Object-Oriented Programming*, 7(9):23–28, 1996.

[2] Conway, D. and C. G. Goebel. Class::Contract – design-by-contract OO in Perl. `search.cpan.org/~ggoebel/Class-Contract-1.14`.

[3] ECMA. ECMAScript language specification.

[4] Elliot, C. and P. Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 163–173, June 1997.

[5] Findler, Barzilay, Blume, Codik, Felleisen, Flatt, Huang, Matthews, McCarthy, Scott, Press, Rainey, Reppy, Riehl, Spiro, Tucker and Wick. The eighth annual ICFP programming contest. `icfpc.plt-scheme.org/`.

[6] Findler, R. and M. Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, 2002.

[7] Findler, R. B. and M. Blume. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming*, pages 226–241, 2006.

[8] Findler, R. B., M. Latendresse and M. Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of ACM Conference Foundations of Software Engineering*, pages 229–236, 2001.

[9] Flapjax Team. Flapjax programming language. `www.flapjax-lang.org`.

[10] Flatt, M. Composable and compilable macros: You want it when? In *ACM SIGPLAN International Conference on Functional Programming*, pages 72–83, 2002.

[11] Harper, R. and G. Morrisett. Compiling polymorphism using intensional type analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, 1994.

[12] Hinze, R., J. Jeuring and A. Löh. Typed contracts for functional programming. In *International Symposium on Functional and Logic Programming*, pages 208–225, 2006.

[13] Ignatoff, D., G. H. Cooper and S. Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *International Symposium on Functional and Logic Programming*, pages 259–276, 2006.

[14] Karaorman, M., U. Hölzle and J. Bruno. jContractor: A reflective Java library to support design by contract. In *Proceedings of Meta-Level Architectures and Reflection*, volume 1616 of *LNCS*, July 1999.

[15] Matthews, J. and R. B. Findler. Operational semantics for multi-language programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–10, 2007.

[16] McFarlane, K. Design by contract framework. `www.codeproject.com/csharp/designbycontract.asp`.

[17] Meyer, B. *Eiffel: The Language*. Prentice Hall, 1992.

[18] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[19] Milner, R., M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[20] Mitchell, J. C. and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[21] Pierce, B. and E. Sumii. Relating cryptography and polymorphism, 2000. Unpublished manuscript.

[22] Plösch, R. Design by contract for Python. In *IEEE Proceedings of the Joint Asia Pacific Software Engineering Conference*, page 213, 1997.

[23] Plösch, R. and J. Pichler. Contracts: From analysis to C++ implementation. In *Technology of Object-Oriented Languages and Systems*, pages 248–257, 1999.

[24] PLT. PLT MzLib: Libraries manual. Technical Report PLT-TR05-4-v300, PLT Scheme Inc., 2005. `www.plt-scheme.org/techreports/`.

[25] Reynolds, J. C. Types, absraction, and parametric polymorphism. In *Information Processing 83*, pages 513–523, 1983.

[26] Rosenblum, D. S. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[27] Tobin-Hochstadt, S. and M. Felleisen. Interlanguage migration: from scripts to programs. In *Object-Oriented Programming Systems, Languages, and Applications*, pages 964–974, 2006.

[28] Wadler, P. Theorems for free! In *International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359, 1989.

[29] Zdancewic, S., D. Grossman and G. Morrisett. Principals in programming languages: a syntactic proof technique. In *International Conference on Functional Programming*, pages 197–207, 1999.