TYPED CONTRACTS FOR GRADUAL TYPING

BRIAN LACHANCE

Submitted in partial fulfillment of the requirements
for the degree of Master of Science

College of Computer and Information Science
Northeastern University
Boston, MA

November 2016

Brian LaChance:
Typed Contracts for Gradual Typing,
Master of Science, Northeastern University, Boston, MA

ACKNOWLEDGMENTS

I thank Asumu Takikawa and Matthias Felleisen for their invaluable guidance, support, and feedback. Without them, this work would not have come this far.

I thank the many members of the PRL for the great times and their friendship. I couldn't have asked for a more warm and welcoming group of people.

I thank the various contributors to Racket for building a great ecosystem and for their help when I was down too far in the weeds. Specifically, crucial advice from Sam Tobin-Hochstadt helped shape the type system presented here.

I thank my family for their never-ending love and support.

I thank my partner Amanda for her love, her patience, and for helping me become who I am today.

CONTENTS

1

INTRODUCTION

Nearly thirty years ago, Eiffel [16] offered programmers an alternative to informal specifications and defensive programming by introducing *contracts*. Contracts allow programmers to define a program's invariants using the same language as the rest of its implementation. By leveraging the entire language, contracts lift familiar boolean expressions into the world of rich specifications. Since Eiffel's first steps, *higher-order* contracts [7] were developed to support contracts on higher-order values such as functions, objects, and mutable references. Higher-order contracts have been adopted in a variety of languages, including JavaScript[1], Haskell[2], Ruby[3], and Racket[4]. Such contracts also enjoy a strong academic history, with publications ranging from dynamic enforcement of affine types [30], playing nicely with laziness [1], using polymorphic functions as contracts [13], and supporting first-class modules [20].

Contracts, tests, and prose all serve as sources of design information for maintenance programmers in an untyped language. When such programs grow large, programmers often want static assurances about their reasoning. Gradual typing [19, 27] allows programmers to turn invariants into types on a piece-by-piece basis. This migration process begins by adding type annotations at the top-level of a module. Gradually typed languages incorporate novel type systems [21, 25, 28] to support the untyped language's idioms such that, ideally, only these top-level annotations are necessary. Thus, programmers can effortlessly add types to the pieces of existing systems where it benefits them, leaving lower priority components untyped. To ensure soundness in this mixed world, the typed side's invariants are enforced dynamically with contracts when typed code interacts with untyped code.

Despite the attention that contracts and gradual typing receive, no prior work has addressed supporting higher-order contracts in gradually typed languages. Without support for contracts in a gradually typed language, contracts in the untyped language cannot be transitioned to the typed world. Migrating an untyped program should not force a programmer to leave contracts behind, as contracts make up an essential part of that program's interface. Languages with contracts *demand* a gradual type system with contracts, too.

1 https://github.com/sefaira/rho-contracts.js
2 https://hackage.haskell.org/package/Contract
3 https://github.com/egonSchiele/contracts.ruby
4 https://docs.racket-lang.org/guide/contracts.html

To develop a solution to this problem, we will look at contracts as found in Racket [10]. Racket has a rich contract system, and many Racket programs use contracts. Additionally, Racket has a gradually typed sister-language, Typed Racket, which can be extended to evaluate how effectively a proposed design supports idiomatic contracts.

This brings me to my thesis

> *gradual typing can effectively and efficiently support higher-order contracts.*

I validate this thesis first by designing a core calculus for typed contracts that supports Racket contract idioms. Additionally, I implement this design by augmenting Typed Racket with typed contracts. I evaluate the effectiveness of these contracts and their type system through four case studies in which I migrate modules with contracts from Racket to Typed Racket.

The remainder of this report begins in chapter 2 by presenting background material on both Racket contracts and Typed Racket, as well as canonical examples that motivate the design. Chapter 3 presents the previously mentioned calculus. In chapter 4, I give details about an implementation of the design in Typed Racket. Chapter 5 presents the evaluation of the design's effectiveness. Finally, chapter 6 presents related work and avenues for future research.

2

CONTRACTS AND GRADUAL TYPING

We begin by introducing contracts and gradual typing through a series of examples, providing background material for the remainder of this report. After the introductory examples, we provide further examples that motivate our design for typing contracts in support of gradual migration.

2.1 CONTRACTS

Higher-order contracts allow programmers to specify invariants in an already familiar programming language. With just predicates written in the same language as the rest of the program, programmers can articulate properties ranging from simple type tests to complex invariants about higher-order functions. These properties are checked through *contract monitoring* [4]. This monitor system will raise an error if the contract is violated. Additionally, the monitor tracks which module is responsible for upholding a particular contract—if the contract is violated, the responsible party is blamed.

To illustrate this concretely, imagine we are writing a warehouse management system for an order fulfillment company. For the order-processing code, our design dictates that a certain item-describing function, `item-name`, expects an argument representing an in-stock item. As the system's implementers, we could check in the body of `item-name` that a given argument passes the `item?` predicate and that it passes the `in-stock?` predicate. But this conflates the function's specification with its implementation. Worse, error detection code can easily accumulate and outweigh the business logic, resulting in a more complex system [15].

Instead, a contract system lets us separate these concerns by writing a function contract like the following:

```
(-> (and/c item? in-stock?)
    string?)
```

This contract checks arguments to `item-name` using the domain contract (bold). The domain contract uses `and/c` to specify that all arguments must pass the first predicate and then they must pass the second predicate. Given an argument to `item-name`, we can immediately check that the argument satisfies the domain contract by applying the argument to the two predicates in sequence. We call contracts *flat* if they can be immediately checked. In addition, the function's contract uses the range contract (also a flat contract) to monitor results from

```
(λ (arg)
  (if (and (item? arg) (in-stock? arg))
      (let ([result (f arg)])
        (if (string? result)
            result
            (error "list-inventory caller broke range")))
      (error "list-inventory broke domain")))
```

Figure 1: Sketch of a contract wrapper for `list-inventory`

`item-name`, assuring clients of our function that they only receive well-specified values. If the domain contract is violated, then the function's caller is blamed. If the range contract is violated, then the function is blamed.

Later on in the system's development, we receive a request for a new function, called `list-inventory`. Given a function that produces item descriptions, it needs to produce a list of descriptions, representing a description of the entire inventory. This time, the feature request came with the following contract.

```
(-> (-> (and/c item? in-stock?) string?)
    (listof string?))
```

Here the domain contract is itself a function contract. Since by Rice's theorem it is impossible to compute interesting properties about an arbitrary function, the monitor cannot immediately check if an argument to `list-inventory` satisfies the domain contract. Thus, function contracts are not flat contracts. The monitor will, when `list-inventory` is given a function as an argument, produce a *wrapper* [7]. The purpose of the wrapper is to delay the checks for the function's domain contract until the wrapper is called and similarly for the function's range contract.

This wrapper, sketched in figure 1, is responsible for checking that all of the function's arguments satisfy the domain contract and that all of the function's results satisfy the range contract. Assume the argument to `list-inventory` is named `f` and is wrapped as in the figure. The wrapper only calls `f` with arguments that pass both of the specified predicates. Similarly, if `f` does not produce an appropriate value, the wrapper does not return a bad value to the caller.

Now we know how to enforce the contract on `list-inventory`, but we have not explained which party is *responsible* for satisfying which part of the contract [4]. On close inspection of the wrapper, the error messages say that `list-inventory` is blamed for not providing a suitable argument to `f`. Although it seems backwards that a function is being blamed for violating part of its domain contract,

`list-inventory`'s clients are trusting it to not violate the contract on `f`. Similarly, the error messages say that `list-inventory`'s caller is blamed if the `f` does not produce a string because `list-inventory` is expecting its clients to provide a string-generating function. The general rule can be explained borrowing terminology from type theory. If we think of the sub-contracts in the function contract as having positive and negative positions as with function types, the function is responsible for the positive positions, and the function's clients are responsible for the negative positions.

Understanding flat contracts, contract composition using `and/c`, and higher-order function contracts is necessary to explain the design of our type system. All of these examples, though, have been untyped. We want to make contracts available in a gradually-typed context, so the next section provides background on our approach to gradual typing.

2.2 GRADUAL TYPING

Untyped languages such as JavaScript, Python and Ruby are in wide use. When programs written in these languages grow large, maintenance activities can be hindered by the lack of types. Maintainers need to understand the component they are maintaining, and they may consult various forms of documentation to discover its invariants. Tests, comments, and reference manuals can all aid in understanding a piece of code, yet even with those resources a key invariant may only be apparent after painstakingly inspecting the implementation. Worse, such an inspection may reveal inconsistencies between documentation and the implementation. By adding meaningful type annotations to the untyped component, a programmer can statically enforce otherwise unspecified invariants and ensure that stated invariants are upheld. These annotations then become a living part of the program's design, evolving with the program and aiding future maintenance work.

Gradual type systems allow programmers to annotate their programs in this manner, taking them from the untyped world to the typed. Roughly, a language with such a type system can be thought of as two related languages: an untyped language and its typed sister language. The type system is designed to support the idioms of the untyped language, giving programmers a smooth transition from untyped to typed. Ideally, the only change that is necessary is adding useful type annotations.

Continuing with our Racket examples from before, we will soon show how our warehouse management system may be migrated to Typed Racket. The program in figure 2, though, shows how our previously-described system retrieves information about the warehouse's inventory, including some tests.

```
(module orders racket
  (provide
   (contract-out
    [item-name (-> (and/c item? in-stock?)
                   string?)]
    [default-item item?]))

  (struct item (name in-stock?))
  (define in-stock? item-in-stock?)

  (define default-item (item "pizza" #t)))

(module test racket
  (require (submod ".." orders))
  (item-name default-item))
```

Figure 2: Version 1

We receive yet another feature request: now, `item-name` needs to distinguish items that do not have a useful name. Its developers should be able to give a replacement name that gets used in the description if the item's name field is the empty string.

We quickly make the necessary changes and send the updated version in figure 3 to our colleagues for review, and they report back that they found one bug: the default name-replacement is `#f`, which means `item-name` does not always produce a string. Our tests and our contracts did not catch this, though it is evident that this is a bug. For more complex code bases, it is easy to imagine more sinister bugs that are harder to detect.

Going forward, we have two options: one, we can fix this bug and maybe just try harder next time, or two, we can migrate our code to Typed Racket, which would have been able to statically check that `item-name` always produces a string. The second option allows us to make stronger statements that more accurately capture our design. Going down that path, though, has another problem: Typed Racket does not support contracts. For reference, the code snippet in figure 4 is what the migrated version looks like in the implementation of the design we present in chapter 3. Specifically, note the annotations added to the `struct` definition and to the function `item-name`. The remainder of the module, including the contracts, did not need annotations.

```
(module orders racket
  (provide
   (contract-out
    [item-name (->* ((and/c item? in-stock?))
                    (string?)
                    string?)]
    [default-item item?]
    [badname-item item?]))

  (struct item (name* in-stock?))
  (define in-stock? item-in-stock?)
  (define (item-name item [name-replacement #f])
    (if (zero? (string-length (item-name* item)))
        name-replacement
        (item-name* item)))

  (define default-item (item "pizza" #t))
  (define badname-item (item "" #t)))

(module test racket
  (require (submod ".." orders))
  (item-name default-item)
  (item-name badname-item "No name given"))
```

Figure 3: Version 2

```
(module orders typed/racket
  (provide
   (contract-out
    [item-name (->* ((and/c item? in-stock?))
                    (string?)
                    string?)]
    [default-item item?]
    [badname-item item?]))
  (struct item ([name* : String] [in-stock? : Boolean]))
  (define in-stock? item-in-stock?)
  (: item-name (->* (item) (String) String))
  (define (item-name item [name-replacement "Default"])
    (if (zero? (string-length (item-name* item)))
        name-replacement
        (item-name* item)))
  ...)
```

Figure 4: Migrated to Typed Racket

2.3 GRADUALLY TYPED CONTRACTS

If contracts are frequently used in a language, its gradually typed version must support contracts too. A good gradual typing system must support the idioms that programmers use. In this section, we describe the particular design challenges that arise when supporting contracts in a gradually typed setting.

Based on experience with Racket's contract system and informal evaluations, we have identified `and/c` as a frequently used contract combinator that warrants special attention from the type system. The following examples reveal the difficulties in typechecking `and/c` that warrant this attention.

As we have already seen, `and/c` is commonly used to guard a predicate with an analogue of a type-test. The following contract from our warehouse example exhibits this behavior:

```
(and/c item? in-stock?)
```

Imagine that this contract is exported from a module. Its purpose statement says that it does two things: it checks that arbitrary values are items and, if they are, it checks that they are in stock. Our type system needs to allow this contract to statically check values of any type, as that is what the remaining untyped modules would expect. As we have seen, this is safe because everything that passes `item?` is also safe for `in-stock?`.

On the other hand, contracts like the following should be rejected:

```
(and/c exact-integer? in-stock?)
```

After checking `exact-integer?` we know that the result is an integer, but the `in-stock?` predicate expects to be called with an item. These sort of contracts are erroneous because we know that integers are never items, and thus should always be rejected.

Reasoning about the numeric tower (and subtyping in general), as in the following contract, needs to be supported as well.

```
(and/c positive? exact-integer?)
```

The first contract, `positive?`, expects to monitor only real numbers and we know that any value it successfully checks is a positive real. The second contract, `exact-integer?`, can monitor anything. The combination of these two, though, lets us conclude that any value passing both contracts must be a positive integer. Combining the overall contract with another contract that expects a positive integer should be allowed.

Our final example has to do with the order of operations when composing higher-order contracts. The previous `and/c` contracts, all flat contracts, are checked from left to right. Unfortunately, function and higher-order contracts in general do not follow this strict left to right order when combined with `and/c`. Consider the following contract.

```
(and/c (-> positive? positive?)
       (-> real? real?))
```

Combining these two contracts results in a contract which, when monitoring a function, will check the predicates in the following order. The `real?` in the domain position does its check first. Next, the `positive?` in the other sub-contract's domain performs its check. A value that passes both of these checks is then given to the monitored function. When the function returns, the `positive?` in the range position checks the result. Following that, the `real?` in the range position performs the final check. As with the earlier examples, the type system must ensure that this flow of values between contracts is safe and not erroneous.

Ultimately, the goal is to determine whether the type of a value that passes a contract is suitable as input to the next contract after it. Occurrence typing [28] solves a similar problem. When a predicate like `number?` is used in a conditional, the type system gains additional information in the then and else branches. In the then branch, it knows the tested value is a number; similarly, in the else branch, it knows the tested value is not a number. We adopt a similar idea for contracts: when a contract succeeds, we may know more information about a value that it monitors. If we monitor an integer with the contract `positive?`, we can assume that the resulting integer must be a positive integer. If the original integer is not positive, the monitor raises an exception blaming the appropriate party.

Let us sketch this reasoning in terms of the type system we propose in the following chapter. We say that the contract `positive?` has type `(Con Real Positive-Real)`. This means that monitoring a `Real` with this contract will have type `Positive-Real`. Because all integers are reals, we allow this contract to monitor an integer. Because positive reals that are integers are also positive integers, monitoring an integer with this contract lets us conclude that the monitored value is a positive integer.

3 TYPING UNTYPED CONTRACT IDIOMS

One of the key goals of gradual typing is for the typed language to support the idioms of its untyped sister language. The examples in chapter 2 show common idioms for Racket programs that use contracts and sketches our design for typed contracts. In this chapter, we articulate our design in terms of a calculus of typed contracts, CPCF_{TR}, which supports those idioms. We begin by introducing plain CPCF.

3.1 CPCF

CPCF is a typed, higher-order language with contracts for capturing the essence of contract monitoring [4]. CPCF itself is an extension of PCF [18] with contracts. To prepare CPCF for later extension, this section presents a slightly modified version that includes subtyping and a generalized contract type. This extension is minimal and mostly straightforward. In the following sections we point out where those changes are necessary.

3.1.1 *Syntax*

Figure 5 shows the type and expression syntax of CPCF.

Its types consist of base types o, function types, and contract types (Con t t). A contract type's first type argument is the *input* type, while the second type argument is the *output* type.

$$
\begin{array}{rl}
o & ::= \mathsf{Natural} \mid \mathsf{Boolean} \mid \bot \\
t & ::= o \mid (\to\ t\ t) \mid (\mathsf{Con}\ t\ t) \\
e & ::= \mathit{natural} \mid \mathit{boolean} \\
 & \quad \mid (\lambda\ (x\ t)\ e) \mid x \\
 & \quad \mid (+\ e\ e) \mid (-\ e\ e) \mid (\mathsf{zero?}\ e) \\
 & \quad \mid (\mathsf{or}\ e\ e) \mid (\mathsf{and}\ e\ e) \mid (\mathsf{if}\ e\ e\ e) \\
 & \quad \mid (e\ e) \mid (\mu\ (x\ t)\ e) \\
 & \quad \mid \mathsf{mon}^{k,l}_{j}\ \kappa\ e \\
 & \quad \mid (\mathsf{error}\ k\ j) \\
\kappa & ::= (\mathsf{flat}\ e) \mid (\mapsto\ \kappa\ \kappa) \mid (\mapsto \mathsf{d}\ \kappa\ (\lambda\ (x\ t)\ \kappa))
\end{array}
$$

Figure 5: CPCF expression and type syntax

A contract with type (Con Natural Natural), for example, can monitor terms according to its input type Natural, and the resulting monitor expression's type is given by the output type, also Natural. One contract with this type is the contract that checks if a natural number is nonzero–that is, if it is positive—which can be written as (flat (λ (n Natural) (if (zero? n) #f #t))). Similarly, a contract on booleans that only passes if the value is not #f can be written as (flat (λ (b Boolean) b)) and has type (Con Boolean Boolean).

Note that this binary contract type generalizes the one found in plain CPCF: CPCF's contract type is unary and is equivalent to ours when both type arguments are the same. Apart from the contract type, the remaining modification to CPCF's syntax is the most specific type $\bot$, which is assigned to blame expressions. All of the remaining syntactic forms are standard with respect to CPCF.

Expressions e include naturals and booleans, functions, variables, primitive operations on values of base type, function application, and a fixed point operator μ, all of which are standard in PCF. The two remaining expression forms are related to contracts.

Contract monitoring $\mathsf{mon}_j^{k,l}\ \kappa\ e$ monitors the inner expression e with the contract κ, ensuring that values flowing into and out of the expression satisfy the contract. The three party labels k, l, and j indicate the parties responsible for satisfying various parts of the contract. With non-dependent contracts on first-order functions, for example, the party in the k position corresponds to the function itself, which must satisfy the contract's range parts. Similarly, the party in the l position corresponds to the function's consumers, which must satisfy the contract's domain parts.

We can give more concrete names to these labels if we recall that, like function types, contracts have positive and negative components. The first label indicates the *positive* party, which is must satisfy a contract's positive components. The second label indicates the *negative* party, which must satisfy a contract's negative components. We will come back to the third label when explaining dependent function contracts.

The final expression form is (error k j), which blames party k for violating the contract corresponding to the label j.

Contracts κ come in flat and higher-order variants. Flat contracts can be constructed with (flat e), which creates contracts out of predicates on base types. Higher-order contracts are constructed using function contracts (↦ κ κ) which monitor a function with a domain contract and a range contract. Dependent function contracts (↦d κ (λ (x t) κ)) are also higher-order, but they allow the range contract (λ (x t) κ) to refer to the function's arguments, which will be bound to x. Importantly, these references to x must be mediated by the domain contract [4]. Consider the following example of a dependent function contract

$$
\begin{array}{rl}
v ::= & \mathit{natural} \mid \mathit{boolean} \\
& \mid (\lambda\ (x\ t)\ e) \\
F ::= & [\,] \\
& \mid (+\ F\ e) \mid (+\ v\ F) \mid (-\ F\ e) \mid (-\ v\ F) \mid (\mathsf{zero?}\ F) \\
& \mid (\mathsf{or}\ F\ e) \mid (\mathsf{and}\ F\ e) \mid (\mathsf{if}\ F\ e\ e) \\
& \mid (F\ e) \mid (v\ F) \\
& \mid \mathsf{mon}^{k,l}_{j}\ \kappa\ F
\end{array}
$$

Figure 6: Value forms and evaluation contexts

```
(↦d (↦ pos? pos?)
    (λ (f (→ Natural Natural))
      (flat (λ (r Natural) (zero? (f 0))))))
```

where pos? is the contract that checks if a natural number is positive. Being a dependent contract, the range contract refers to the function's argument f. The range contract, just like the rest of the program, must respect the domain contract. Thus, attempting to call f with 0 will result in a contract violation.

To summarize, the syntax generalizes CPCF with two minor changes to the types: the generalized contract type (Con t t), and the addition of the type $\bot$.

3.1.2 *Dynamic semantics*

We begin with the language's dynamic semantics, which is identical to standard CPCF. The semantics is stated as a contextual reduction system [5]. To define this language's call-by-value reduction relation, we must first cover its values and evaluation contexts.

Figure 6 gives the language's value forms and evaluation contexts. Values include naturals, booleans, and functions. The dynamic semantics of CPCF_{TR} is defined by the reduction relation in figure 7, which uses evaluation contexts F to lift the relation $\mathbf{v}$ to one that reduces outermost subexpressions from left to right. As in the original CPCF calculus, (flat e) terms are not reduced until they are used as the sole contract in a monitor expression.

Most of the $\mathbf{v}$ relation is straightforward, so we focus our discussion on the rules concerning contracts. E-MonAssign gives the server party s responsibility for contract monitors in the surface-level program. This is because the contract monitor divides the program into two pieces. Roughly, the code inside the monitor is responsible for the server's obligations and the code outside the monitor for the client's obligations. E-MonFlat extracts the predicate from a (flat e) expression and applies it to the monitored value, blaming the party specified in the monitor if the predicate fails.

$$(+\ natural_1\ natural_2) \ \mathbf{v}\ natural_r \quad \text{where } natural_r = natural_1 + natural_2 \qquad \text{[E-Add]}$$

$$(-\ natural_1\ natural_2) \ \mathbf{v}\ natural_r \quad \text{where } natural_r = natural_1 \text{ - } natural_2 \qquad \text{[E-Subtract]}$$

$$(\mathsf{zero?}\ 0) \ \mathbf{v}\ \#\mathrm{t} \qquad \text{[E-ZeroT]}$$

$$(\mathsf{zero?}\ natural) \ \mathbf{v}\ \#\mathrm{f} \quad \text{where } natural \neq 0 \qquad \text{[E-ZeroF]}$$

$$(\mathsf{or}\ boolean\ e) \ \mathbf{v}\ boolean \quad \text{where } boolean \neq \#\mathrm{f} \qquad \text{[E-OrT]}$$

$$(\mathsf{or}\ \#\mathrm{f}\ e) \ \mathbf{v}\ e \qquad \text{[E-OrF]}$$

$$(\mathsf{and}\ boolean\ e) \ \mathbf{v}\ e \quad \text{where } boolean \neq \#\mathrm{f} \qquad \text{[E-AndT]}$$

$$(\mathsf{and}\ \#\mathrm{f}\ e) \ \mathbf{v}\ \#\mathrm{f} \qquad \text{[E-AndF]}$$

$$(\mathsf{if}\ boolean\ e_1\ e_2) \ \mathbf{v}\ e_1 \quad \text{where } boolean \neq \#\mathrm{f} \qquad \text{[E-IfT]}$$

$$(\mathsf{if}\ \#\mathrm{f}\ e_1\ e_2) \ \mathbf{v}\ e_2 \qquad \text{[E-IfF]}$$

$$(\mu\ (x\ t)\ e) \ \mathbf{v}\ e[x \mapsto (\mu\ (x\ t)\ e)] \qquad \text{[E-μ]}$$

$$((\lambda\ (x\ t)\ e)\ v) \ \mathbf{v}\ e[x \mapsto v] \qquad \text{[E-βv]}$$

$$\mathsf{mon}_j^{k,l}\ (\mathsf{flat}\ e)\ v \ \mathbf{v}\ (\mathsf{if}\ (e\ v)\ v\ (\mathsf{error}\ k\ j)) \qquad \text{[E-MonFlat]}$$

$$\mathsf{mon}_j^{k,l}\ (\mapsto \kappa_{dom}\ \kappa_{rng})\ v \ \mathbf{v}\ (\lambda\ (x\ t)\ \mathsf{mon}_j^{k,l}\ \kappa_{rng}\ (v\ \mathsf{mon}_j^{l,k}\ \kappa_{dom}\ x)) \quad \text{where } (\lambda\ (x\ t)\ e) = v \qquad \text{[E-MonHO]}$$

$$\mathsf{mon}_j^{k,l}\ (\mapsto \mathsf{d}\ \kappa_{dom}\ (\lambda\ (x\ t)\ \kappa_{rng}))\ v \ \mathbf{v}\ (\lambda\ (x\ t)\ \mathsf{mon}_j^{k,l}\ \kappa_{rng}[x \mapsto \mathsf{mon}_j^{l,j}\ \kappa_{dom}\ x]\ (v\ \mathsf{mon}_j^{l,k}\ \kappa_{dom}\ x)) \qquad \text{[E-MonD]}$$

$$F[(\mathsf{error}\ k\ j)] \longrightarrow (\mathsf{error}\ k\ j) \qquad \text{[E-Blame]}$$

Figure 7: CPCF's reduction relation

$$\frac{}{t <: t}\ \text{[S-Ref]}$$

$$\frac{}{\bot <: t}\ \text{[S-Bot]}$$

$$\frac{t_{dom2} <: t_{dom1} \qquad t_{rng1} <: t_{rng2}}{(\to\ t_{dom1}\ t_{rng1}) <: (\to\ t_{dom2}\ t_{rng2})}\ \text{[S-Arr]}$$

$$\frac{t_{in2} <: t_{in1} \qquad t_{out1} <: t_{out2}}{(\mathsf{Con}\ t_{in1}\ t_{out1}) <: (\mathsf{Con}\ t_{in2}\ t_{out2})}\ \text{[S-Con]}$$

Figure 8: Subtype relation

E-MonHO takes a monitored function and its contract and produces a proxied form of that function guarded by the domain and range contracts. The proxy monitors the function's inputs using the domain contract and necessarily swaps the responsible party for the argument monitor—after all, if a caller gives an argument to a function that fails the domain contract, it was not the function's fault. All that remains is the range contract, which the proxy uses to monitor the value returned by the function, this time with the original party.

E-MonD is similar, but the variable in its range contract now refers to the *monitored* argument of the resulting proxy. This ensures that the range contract can depend on the function's arguments and prevents the range contract from violating the domain contract.

Finally, E-Blame shows that reducing an error term terminates the remaining computation.

3.1.3 *Static semantics*

Next, we present the static semantics of CPCF. Subtyping is the largest change in our presentation of CPCF, so we begin by quickly walking through the subtype relation in figure 8.

Our subtype relation $<:$ has the usual rule for reflexivity, as well as for the most specific type $\bot$. A function type is a subtype of another if they satisfy the standard variance rules. For the functions' domains, inputs to the second function at type t_{dom2} must be valid inputs to the first function, which expects inputs of type t_{dom1}. For the functions' ranges, the first function's results at type t_{rng1} must also be valid results for the second function at type t_{rng2}.

Two contract types are related if they satisfy constraints similar to function types. Values monitorable by the second contract at type

$$\frac{}{t \sqcup t = t}\ \text{[lub-Ref]}$$

$$\frac{}{\bot \sqcup t = t}\ \text{[Lub-BotL]}$$

$$\frac{}{t \sqcup \bot = t}\ \text{[lub-BotR]}$$

Figure 9: Least upper bound

t_{in2} must be a subtype of what the first contract can monitor at type t_{in1}. Additionally, the resulting monitor expression at type t_{out1} must be a subtype of type t_{out2}. This rule allows contracts with type (Con $\top$ Natural) to be used in a context requiring contracts with type (Con Natural Natural).

Next, we define the least upper bound $\sqcup$ with respect to <:. This auxiliary relation is necessary for typing if expressions introduced during contract monitoring. The types of the two branches differ in such expressions, and the least upper bound of the branches' types is the type of the entire conditional.

For this purpose, the $\sqcup$ defined in figure 9 is sufficient. First, it is reflexive: the least type that is a supertype of any pair of identical types is exactly that type. For any type t and the most specific type $\bot$, the least type that is a supertype of both is t. All other cases are undefined, though this is not essential—we could have introduced a most-general type $\top$ and defined that to be the result in such cases.

The final part of the static semantics is the typing relation, shown in figures 10 and 11. This relation is defined by two mutually recursive judgments $\Gamma \vdash e : t$ and $\Gamma \vdash_{\kappa} \kappa : t$, relating expressions and contracts, respectively, to a type t in a context Γ. The judgments' cases are standard in CPCF apart from (conventional) usages of the subtype relation and the contract monitoring case.

Most cases of the expression judgment are straightforward. T-Natural and T-Boolean give natural and boolean values their corresponding base type, and cases T-zero?, T-Add, T-Subtract, T-Or, and T-And state that operations on base types require arguments of that base type. With a well-typed test expression, if expressions have a type that is the least upper bound of the two branches' types.

T-Var gives variables a type by consulting the context Γ. By T-Arr, a function's domain type is determined by its argument's annotated type. The type of the body, which may refer to that argument, determines the range type. In T-FunApp, applying a function requires the argument

$$\frac{}{\Gamma \vdash \mathit{natural} : \mathsf{Natural}}\ [\text{T-Natural}]$$

$$\frac{}{\Gamma \vdash \mathit{boolean} : \mathsf{Boolean}}\ [\text{T-Boolean}]$$

$$\frac{\Gamma \vdash e : t \qquad t <: \mathsf{Natural}}{\Gamma \vdash (\mathsf{zero?}\ e) : \mathsf{Boolean}}\ [\text{T-zero?}]$$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t}\ [\text{T-Variable}]$$

$$\frac{(x\ t_{dom}\ \Gamma) \vdash e : t_{rng}}{\Gamma \vdash (\lambda\ (x\ t_{dom})\ e) : (\rightarrow\ t_{dom}\ t_{rng})}\ [\text{T-Arr}]$$

$$\frac{(x\ t\ \Gamma) \vdash e : t}{\Gamma \vdash (\mu\ (x\ t)\ e) : t}\ [\text{T-Fix}]$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad t_1 <: \mathsf{Natural} \qquad \Gamma \vdash e_2 : t_2 \qquad t_2 <: \mathsf{Natural}}{\Gamma \vdash (+\ e_1\ e_2) : \mathsf{Natural}}\ [\text{T-Add}]$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad t_1 <: \mathsf{Natural} \qquad \Gamma \vdash e_2 : t_2 \qquad t_2 <: \mathsf{Natural}}{\Gamma \vdash (-\ e_1\ e_2) : \mathsf{Natural}}\ [\text{T-Subtract}]$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad t_1 <: \mathsf{Boolean} \qquad \Gamma \vdash e_2 : t_2 \qquad t_2 <: \mathsf{Boolean}}{\Gamma \vdash (\mathsf{or}\ e_1\ e_2) : \mathsf{Boolean}}\ [\text{T-Or}]$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad t_1 <: \mathsf{Boolean} \qquad \Gamma \vdash e_2 : t_2 \qquad t_2 <: \mathsf{Boolean}}{\Gamma \vdash (\mathsf{and}\ e_1\ e_2) : \mathsf{Boolean}}\ [\text{T-And}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e_{test} : t_{test} \qquad t_{test} <: \mathsf{Boolean} \\ \Gamma \vdash e_{then} : t_{then} \qquad \Gamma \vdash e_{else} : t_{else} \qquad t_{then} \sqcup t_{else} = t\end{array}}{\Gamma \vdash (\mathsf{if}\ e_{test}\ e_{then}\ e_{else}) : t}\ [\text{T-If}]$$

$$\frac{\Gamma \vdash e_1 : (\rightarrow\ t_{dom}\ t_{rng}) \qquad \Gamma \vdash e_2 : t_{arg} \qquad t_{arg} <: t_{dom}}{\Gamma \vdash (e_1\ e_2) : t_{rng}}\ [\text{T-FunApp}]$$

$$\frac{\Gamma \vdash_{\kappa} \kappa : (\mathsf{Con}\ t_i\ t_o) \qquad \Gamma \vdash e : t_{exp} \qquad t_{exp} <: t_i}{\Gamma \vdash \mathsf{mon}^{k,l}_{j}\ \kappa\ e : t_o}\ [\text{T-ConApp}]$$

$$\frac{}{\Gamma \vdash (\mathsf{error}\ k\ j) : \bot}\ [\text{T-Error}]$$

Figure 10: Typing relation: expressions

$$\frac{\Gamma \vdash e : (\to o \ \mathsf{Boolean})}{\Gamma \vdash_\kappa (\mathsf{flat}\ e) : (\mathsf{Con}\ o\ o)}\ [\text{T-FlatCon}]$$

$$\frac{\Gamma \vdash_\kappa \kappa_{dom} : (\mathsf{Con}\ t_{dom\text{-}i}\ t_{dom\text{-}o}) \qquad \Gamma \vdash_\kappa \kappa_{rng} : (\mathsf{Con}\ t_{rng\text{-}i}\ t_{rng\text{-}o})}{\Gamma \vdash_\kappa (\mapsto \kappa_{dom}\ \kappa_{rng}) : (\mathsf{Con}\ (\to t_{dom\text{-}o}\ t_{rng\text{-}i})\ (\to t_{dom\text{-}i}\ t_{rng\text{-}o}))}\ [\text{T-ConArrow}]$$

$$\frac{\Gamma \vdash_\kappa \kappa_{dom} : (\mathsf{Con}\ t_{dom\text{-}i}\ t_{dom\text{-}o}) \qquad (x\ t_{dom\text{-}o}\ \Gamma) \vdash_\kappa \kappa_{rng} : (\mathsf{Con}\ t_{rng\text{-}i}\ t_{rng\text{-}o})}{\Gamma \vdash_\kappa (\mapsto \mathsf{d}\ \kappa_{dom}\ (\lambda\ (x\ t_{dom\text{-}o})\ \kappa_{rng})) : (\mathsf{Con}\ (\to t_{dom\text{-}o}\ t_{rng\text{-}i})\ (\to t_{dom\text{-}i}\ t_{rng\text{-}o}))}\ [\text{T-ConArrowD}]$$

Figure 11: Typing relation: contracts

to be a subtype of the domain type and produces a result which has range type.

A contract monitor expression is given a type according to T-ConApp. Like a function application, the expression the contract monitors must be a subtype of the contract's input type. The monitor expression itself, then, has the same type as the contract's output type. To illustrate this, we will look at the always-passing contract any/c, which can be written as (flat (λ (*x* ⊤) #t)). Because it is a predicate on ⊤, it can monitor any value, as in the following examples.

$$\mathsf{mon}_{\mathsf{l}}^{\mathsf{k,j}}\ \mathsf{any/c}\ 5$$
$$\mathsf{mon}_{\mathsf{l}}^{\mathsf{k,j}}\ \mathsf{any/c}\ \#\mathrm{f}$$

The contract any/c has output type ⊤, and so the examples above all have type ⊤.

Error expressions are given type ⊥ by T-Error.

Next, we present the contract type judgment. T-FlatCon gives arbitrary predicates on base types the contract type (Con *o o*).

T-ConArrow shows how function contracts are typed. Because they monitor functions, their input and output types must be function types. To understand the typing rule for function contracts, it may help to recall the reduction rule E-MonHO for monitors with a function contract. The monitored function is wrapped in a proxy, and the proxy ensures that the function is only called with values that pass the domain contract κ_{dom}. Those values have type $t_{dom\text{-}o}$, which means that the monitored function must accept values of that type. Similarly, values that the monitored function produces must pass the range contract κ_{rng}, meaning that the results of the function must be valid inputs to the range contract at type $t_{rng\text{-}i}$. The remaining two types, $t_{dom\text{-}i}$ and $t_{rng\text{-}o}$, are used to construct the output type of this contract.

$$o ::= .... \mid \top$$
$$\kappa ::= .... \mid \mathsf{natural/c} \mid (\mathsf{and/c}\ \kappa\ \kappa)$$

Figure 12: CPCF_{TR}'s syntax extensions

$$\mathsf{mon}_j^{k,l}\ \mathsf{natural/c}\ \textit{natural}\ \mathbf{v}\ \textit{natural} \qquad \text{[E-MonNat-Pass]}$$

$$\mathsf{mon}_j^{k,l}\ \mathsf{natural/c}\ v\ \mathbf{v}\ (\mathsf{error}\ k\ j) \qquad \text{[E-MonNat-Fail]}$$
$$\text{where } \mathsf{not\text{-}natural?}[\![v]\!]$$

$$\mathsf{mon}_j^{k,l}\ (\mathsf{and/c}\ \kappa_1\ \kappa_2)\ v\ \mathbf{v}\ \mathsf{mon}_j^{k,l}\ \kappa_2\ \mathsf{mon}_j^{k,l}\ \kappa_1\ v \qquad \text{[E-MonAnd/c]}$$

Figure 13: CPCF_{TR}'s dynamic semantics extensions

T-ConArrowD is similar to the non-dependent version. The only difference is that the range contract can refer to the function's argument at the domain contract's output type.

3.2 CPCF_{TR}

Having covered the essentials of CPCF, we now introduce the key extensions to support untyped contract idioms.

3.2.1 *Syntax*

The syntactic extensions are shown in figure 12. We begin with an additional base type, the most general type $\top$ which all values inhabit. To the contracts, we introduce two forms. The first is natural/c, a flat contract that can monitor expressions of any type but raises blame if the expression does not evaluate to an natural. The second form is the contract combinator and/c, which allows two contracts—flat or higher-order—to be composed such that the second contract monitors the result of monitoring with the first contract.

3.2.2 *Dynamic semantics*

To CPCF's dynamic semantics, CPCF_{TR} adds three cases (figure 13). The first two, E-MonNat-Pass and E-MonNat-Fail, specify how a monitor with natural/c evaluates. If it monitors a value that is an natural, the contract passes. If the value is not an natural, though, the contract raises blame. The final case is E-MonAnd/c, which shows how and/c contract monitors turn into two nested contract monitors. The inner

$$\frac{}{t <: \top}\ \text{[S-Top]}$$

Figure 14: Addition to subtype relation

$$\frac{o_1 <: o_2}{\mathsf{comb}[\![o_1, o_2, o_1]\!]}$$

$$\frac{o_2 <: o_1}{\mathsf{comb}[\![o_1, o_2, o_2]\!]}$$

$$\frac{\mathsf{comb}[\![t_{1d}, t_{2d}, t_{rd}]\!] \qquad \mathsf{comb}[\![t_{1r}, t_{2r}, t_{rr}]\!]}{\mathsf{comb}[\![(\to t_{1d}\ t_{1r}), (\to t_{2d}\ t_{2r}), (\to t_{rd}\ t_{rr})]\!]}$$

Figure 15: The comb operator

monitor guards the value with the first contract, and the outer monitor guards the inner monitor with the second contract.

3.2.3 *Static semantics*

The static semantics for CPCF_{TR} involves adding two cases to that of CPCF, one for each contract form. But first, we must add $\top$ to our subtype relation (figure 14) and introduce an auxiliary judgment used when typechecking `and/c` contracts.

Pronounced as in the word combination

The comb operator is given in figure 15 and is similar to a greatest lower bound. For base types that are in the subtype relation, the comb is their greatest lower bound. For function types, their comb is determined by taking the comb pairwise for each function's domain and range. As mentioned, this operator is used when combining contracts with `and/c`. It ensures that the combination's output type tracks relevant information from each sub-contract's output type.

The combination of the two function contracts in the example below helps illustrate this requirement.

```
(and/c (↦ natural/c natural/c)
       (↦ (flat (λ (x Natural) (zero? x)))
          (flat (λ (x Natural) (zero? x)))))
```

Monitoring a function with this contract is only safe if the resulting proxy can only be called with Natural values. If any other values were allowed, the proxy would check the second domain contract with an unsafe value. The contract's output type determines how the

$$\frac{}{\Gamma \vdash_{\kappa} \mathsf{natural/c} : (\mathsf{Con}\ \top\ \mathsf{Natural})}\ \text{[T-natural/c]}$$

$$\frac{\Gamma \vdash_{\kappa} \kappa_1 : (\mathsf{Con}\ t_{i1}\ t_{o1}) \qquad \Gamma \vdash_{\kappa} \kappa_2 : (\mathsf{Con}\ t_{i2}\ t_{o2}) \qquad t_{o1} <: t_{i2} \qquad \mathsf{comb}[\![t_{o1}, t_{o2}, t_o]\!]}{\Gamma \vdash_{\kappa} (\mathsf{and/c}\ \kappa_1\ \kappa_2) : (\mathsf{Con}\ t_{i1}\ t_o)}\ \text{[T-ConAnd]}$$

Figure 16: CPCF_{TR}'s typing judgment cases

proxy can be called, and so the comb of the two contracts' output types ensures this is safe, as shown with the type rule for and/c. We cannot simply use the second contract's output type for the combined contract's output type. To understand why, consider if the positions of the previous two sub-contracts were reversed. The output type of the natural/c-based contract is $(\rightarrow \top\ \mathsf{Natural})$, which means that the monitored function can be called with any value. This hypothetical strategy is thus not safe, as it allows the Natural function to be called with an unsafe argument.

The last two type rules in figure 16 make up CPCF_{TR}'s final extensions to CPCF. The first rule, T-natural/c, gives natural/c its type. The contract monitors terms of any type and its input type reflects that. The contract only passes, though, if what it monitors is actually an natural—thus its output type.

T-ConAnd says that two contracts can be composed if the first contract's output type is a subtype of the second contract's input type. The output type for the overall contract is determined by taking the comb of the two output types. Finally, the input type of the overall contract is the input type of the first contract.

3.3 NOTE: TOWARDS MORE PRACTICAL TYPES

The calculus in this chapter captures important qualities about typechecking contracts and the terms they monitor. To effectively use this system for gradual migration, though, we must address one remaining issue.

When a programmer gradually migrates a program from an untyped language to a typed one, they expect the typechecker to find impedence mismatches in the program, as with any typed programming language. If a function expects to be called with naturals, for example, the programmer will annotate it with the appropriate type and the typechecker will complain if any context attempts to call it with non-naturals. Such a function may also have a contract on it; here, a likely contract is $(\mapsto$ natural/c natural/c$)$.

It is important that monitoring a function with a contract does not surprise the programmer by weakening the types that they wrote down. A function with type (→ Natural Natural) can safely be monitored by the above contract of type (Con (→ Natural ⊤) (→ ⊤ Natural)). The output type of this contract, though, means that the monitored function has a more liberal domain, even though we know from the domain contract's type that it will reject all non-Natural arguments. With the current type rule for contract monitoring, the type system will allow a programmer to call the monitored function with a Boolean. In some sense, this type rule disregards the type of the function and surprises the programmer with a weaker type.

To work around this problem, the programmer could replace the type test in the domain contract with a dummy contract like (flat (λ (*x* Natural) #t)). But requiring that workaround makes the migration process more arduous, even if a programmer is likely to remove such redundant type tests at some later point.

Due to time constraints, we only sketch a solution this problem. From a high level, we modify the type rule T-ConApp such that it uses the comb operator to avoid this problem. In the conclusion of the rule, replace the overall expression's type with the combination of monitored expression's type t_{exp} and the contract's output type t_o. That is, the type of the overall expression should be (comb t_{exp} t_o).

4 THE IMPLEMENTATION

In this chapter, we give a high-level overview of the steps for implementing our typed contract language as an extension of Typed Racket (TR). Because TR is a production gradual type system, extending it means we can evaluate our language's effectiveness on real-world programs. Before diving into the implementation of the contract-specific extensions, we provide some general, background on TR's existing implementation strategies.

4.1 HANDLING SYNTACTIC EXTENSIONS

Typed Racket aims to fully support idiomatic Racket code. Racket, being an extensible language, frequently uses macros for its syntactic extensions, requiring TR to typecheck programs that use macros. Its original solution was presented by Tobin-Hochstadt et al. [29]. TR typechecks programs with macros by only typechecking code after macro expansion finishes—whatever a macro expands into is all that the typechecker sees. Many macros expand into code that is easily handled by the typechecker, meaning those do not require any special support from TR. Unfortunately, this does not scale to complex macros such as those in the contract library. The expansion of these forms typically contain identifiers that are private to the contract library, among other pieces of information not relevant to typechecking. TR already implements tricks to handle similar private identifiers, but doing so forces TR to depend on specific details of a macro's expansion which is brittle and cumbersome. In particular, the contract library defines roughly thirty identifiers that can appear in common contract expansion. As an alternative, we employ a technique used in typechecking other complicated macros.

Racket implements first-class classes [9] and first-class modules [8] using macros. Typechecking the expansion of these macros, as described by Takikawa [23] and Feltey [6], respectively, involves defining new versions that annotate the surface syntax. Once the surface syntax is annotated, the new macros delegate to the original form, as shown in the following example:

```
(define-syntax (and/c stx)
  (syntax-parse stx
    [(and/c ctc:expr ...)
     (define subs (for/list ([ctc (in-syntax #'(ctc ...))]
                             [idx (in-naturals)])
                    (ctc:and/c-sub-property ctc idx)))
```

```
(ctc:and/c #`(untyped:and/c #,@subs))]))
```

This defines a macro `and/c` that annotates the surface syntax with a running index before delegating to the original Racket form, `untyped:and/c`. In Racket, macros are functions from syntax to syntax. The argument to the macro, here named `stx`, is bound to the piece of syntax that invoked the macro. For example, The invocation `(and/c exact-integer? positive?)` causes `stx` to be bound to the entire syntax representing that invocation. This macro uses `syntax-parse` [2], a library for writing rich pattern-based macros, to identify the series of sub-forms inside the `and/c` and bind the result of the pattern to `ctc`.

The macro binds the annotated sub-forms to `subs`, using `ctc:and/c-sub-property` to annotate each sub-form with its index in the macro invocation. The order is important during typechecking because we want to typecheck from left to right. Next, it splices that list of annotated syntax into new syntax that delegates to the original Racket form. Finally, the resulting syntax is annotated with `ctc:and/c` so that the typechecker can know which type rule to use.

Racket's syntax model preserves these annotations throughout expansion, which means that this general technique allows type rules to traverse fully expanded code and identify the relevant pieces, regardless of how the delegated-to macro expands. We apply this technique throughout our implementation for contract combinators such as `->`, `->i`.

4.2 Simple Racket Contracts in Typed Racket

For typechecking simpler contract forms, we can use a less intrusive strategy. In particular, not all of Racket's library of contracts are defined using macros. The contract `natural-number/c`, for example, is a function that can be given the type `(Con Any Natural)`. Similarly, simple contract combinators are implemented with functions and thus can be assigned function types. When given a `Real`, the combinator `</c` produces a contract that checks if a value is a number less than the original argument. Its type, then, is `(-> Real (Con Any Real))`. We assign types to simple contract forms in the base type environment. This environment maps identifiers to their types, and Typed Racket uses this for much of Racket's standard library.

Beyond the predefined contracts and combinators in the contract library, we can allow general predicates to be used as contracts. Our implementation lets functions of type `(-> S Boolean)` be used as contracts at type `(Con S S)`. In the model this was restricted to functions on base types, but our implementation allows predicates on higher-order types to be contracts. Additionally, a function of type `(-> S Boolean #:+ T)` can be used as a contract at type `(Con S T)`. The information after `Boolean`, called a *filter*, tells us that if the function produces `#t`, we know that its argument has type `T`. Although TR

uses them for occurrence typing [28], filters directly correspond to the meaning of our contract type's output position and so we can use them to give more precise contract types.

4.3 TYPECHECKING COMPLEX RACKET CONTRACTS

Once the macro expander has fully expanded the surface syntax, the typechecker uses the source annotations to typecheck the resulting expansion. For a given piece of syntax, a type rule's implementation performs three tasks: find the contract's sub-contracts, typecheck those sub-contracts, and check that the types line up according to the model's type rule.

We continue to illustrate this process with `and/c`, whose type rule is implemented in figure 17. The first `define` form is where the earlier annotations are used. It uses the helper function `trawl-for-subs` to find sub-contracts in a piece of syntax, its first argument, and uses the previously annotated properties to identify the relevant pieces of syntax. Additionally, it does not conflate nested sub-contracts in contracts like `(and/c k (and/c l m))`. When traversing the outer `and/c`, it produces the sub-contract `k` and the entire inner `and/c`. It will not produce the inner sub-contracts `l` and `m`.

Now that we have the form's sub-contracts, we can calculate their types and ensure that they line up in accordance with the model. A list of types is bound to `subs-tys`: it contains each sub-contract's type, calculated using `tc-expr/t`, which is immediately coerced to a contract type. Next, we enforce the requirement that each contract's output type is a valid input for the immediately following contract. Finally, we produce the overall contract's type using the input and output types bound by the `define-values` form.

4.4 BEYOND THE MODEL

The parts of the typechecker described so far are a relatively straightforward implementation of the model's type rules. While the model captures the essence of Racket's contract system, it simplifies some of the pragmatic details.

For example, Racket does not have a general contract combinator that can negate any contract. It has `not/c`, but this only applies to flat contracts. That is, negating contracts with higher-order behavior, as in `(not/c (-> real? real?))`, is always erroneous. To resolve this issue, we introduce a new type constructor `FlatCon`. `FlatCon` is identical to `Con` except that only flat contracts have this type. Our earlier statement about predicates can now be refined: predicates of type `(S -> Boolean)` have type `(FlatCon S S)` and similarly for predicates with a filter. Racket's other forms of contracts are limited to the higher-order `Con` types.

```
(define (tc-and/c form)
  (define subs (sort (trawl-for-subs
                      (ctc:and/c-sub-property form #f)
                      (syntax-parser [:ctc:and/c^ #t]
                                     [_ #f])
                      ctc:and/c-sub-property)
                     <
                     #:key ctc:and/c-sub-property))
  (define subs-tys
    (map (compose coerce-to-con tc-expr/t) subs))
  (define-values (in-ty out-ty)
    (if (empty? subs-tys)
        (values Univ Univ)
        (for/fold ([in-ty (Con*-in (first subs-tys))]
                   [out-ty (Con*-out (first subs-tys))])
                  ([ty (in-list (rest subs-tys))])
          (define next-in-ty (Con*-in ty))
          (unless (subtype out-ty next-in-ty)
            (tc-error/fields
             "preceding contract's output type is not ..."
             #:delayed? #f
             "previous output type" out-ty
             "next input type" next-in-ty))
          (values in-ty (comb out-ty
                              (Con*-out ty))))))
  (ret (-Con in-ty out-ty)))
```

Figure 17: and/c's type rule

To further support negating contracts with `not/c`, we can use TR's filters to give more precise types to another kind of contract found in idiomatic Racket programs. Earlier we saw that filters convey facts for when a predicate returns true, but they also provide facts for when a predicate returns false. For example, the function `positive?` effectively has type `(-> Real Boolean : Positive-Real)`. In the true case, this type says that the argument is a `Positive-Real`. In the false case, we can conclude that the argument is *not* a `Positive-Real`. We use the negative information when `not/c` is given a function to conclude that `(not/c positive?)` is a `(FlatCon Real Nonpositive-Real)`.

4.5 LIMITATIONS

TR's filter machinery plays a key part in giving expressive types to contracts. Unfortunately, the language's ability to generate these filters is limited. For example, it does not have filters that precisely describe hash tables. This limitation is the source of a migration difficulty we briefly describe during our evaluation in chapter 5.

An additional limitation is that it is unclear how to assign types to low-level parts of Racket's contract library. The library includes, for example, hooks to approximately compare two contracts for whether one accepts fewer values than the other. Giving useful types to such generic functions is difficult.

Finally, similar to `not/c`, Racket's contract system the values that particular contracts can monitor. To explain this, we need to introduce two interposition tools at the core of Racket's contract system called *impersonators* and *chaperones* [22]. Impersonators are the most general form of interposition in Racket and can proxy operations on values, such as retrieving an element from a vector with `vector-ref`. Chaperones are a kind of impersonator but with an additional restriction. In particular, a chaperone cannot arbitrarily change the value returned by an operation. They can either raise an error, such as blame, or further chaperone the returned value. This means that programmers can trust an immutable vector wrapped with a chaperone to still be immutable—if the interposition were allowed to arbitrarily replace the values returned by an operation, such as `vector-ref`, then an immutable vector would effectively be mutable. Non-chaperone impersonators *can* perform such arbitrary replacements, and Racket forbids them from interposing on immutable values. Similarly, immutable values can only be monitored by contracts that are implemented with chaperones.

Because TR does not distinguish between mutable and immutable values, such as hash tables, our contract type system cannot enforce the restriction that only chaperone-based contracts can monitor immutable values. Therefore, programmers could unknowingly attempt to monitor an immutable value with a non-chaperone impersonator,

resulting in a runtime error. In the future, we may able to rule out these erroneous attempts, as we do with `not/c`. For now, though, we do not support contracts that have this restriction.

5 THE EVALUATION

In this chapter, we investigate whether our implementation supports migrating Racket programs with contracts to Typed Racket in practice. Languages with gradual type systems must support incremental change, and extensions to the language must preserve that feature. To that end, we report on two aspects of migrating contracts: the effort necessary to migrate a Racket program's contracts, and the impact on runtime performance from migrating a Racket program's contracts to Typed Racket. We measure migration effort and runtime performance for four case studies. We begin by describing the features supported in our current implementation and their impact on our evaluation.

5.1 FEATURE COVERAGE

Racket's rich contract library should be supported as much as possible by Typed Racket. We aim to support commonly used contracts, and so our targets are two core sets of contracts: Racket's data-structure [10, Section 8.1] and function contracts.

Our evaluation hypothesis is that the majority of contracts use only these two parts of the contract library. The data-structure contracts provide contracts and combinators for many of Racket's built-in data structures. To specify contracts on lists, programmers write a contract like the following example:

```
(listof natural-number/c)
```

This contract checks if a list contains only natural numbers. For the function contracts, we initially focused on Racket's `->`, `->*`, and `->i` forms. The first two are useful for specifying invariants on functions that do not have dependencies between their inputs and their outputs. Additionally, `->*` and `->i` can be used to specify pre- and post-conditions.

At this time, our implementation supports only a subset of the data-structure contracts and does not support `->*`. Some of the data structure contracts cannot be supported due to limitations of our design, while others simply require more effort. None of these are fatal for our case studies, but they do impose additional requirements for migrating contracts to Typed Racket.

Our design means that data-structure contracts related to hash tables, vectors, and boxes are not supported. Typed Racket lacks a type-level distinction between mutable and immutable variants, which means that our implementation's type system cannot rule out certain erroneous contract applications, as explained in section 4.5

The most significant data structure contract that is limited by our implementation work is the previously mentioned `struct/c`. Similarly, its variant for expressing dependencies between the fields of the struct, `struct/dc`, is also unsupported.

The lack of support for `->*` is also a limitation of our implementation. With `->*`, programmers can express more invariants than with `->`, but all `->*` contracts can be rewritten with `->i`. Given more time, the implementation could support `->*` and rewriting these contracts would not be necessray.

Finally, contracts on polymorphic functions are currently not supported. The presentation of our results expands on these limitations with details specific to each case study.

5.2 PROCESS

Our evaluation explores two dimensions of how well our system supports migrating Racket contracts: migration effort and runtime performance.

Migration effort quantifies the changes necessary to migrate a contract from Racket to Typed Racket. This part of the evaluation echos the evaluations done for functional Typed Scheme [26], for Typed Racket with units [6], and for object-oriented Typed Racket [23]. For each of our case studies, we report on five kinds of changes:

- annotations for anonymous functions
- making contracts more precise (such as replacing a `real?` in a function's domain contract when the the domain type is `Integer`)
- general changes to migrate contracts, such as workarounds
- fixes for bugs that the migration of contracts revealed
- unsupported contracts that could not be migrated with reasonable workarounds.

Runtime performance is the final measurement in our evaluation. Specifically, we want to assess whether a Racket program with contracts performs differently than the same program migrated to Typed Racket, both with identical contracts. Post-migration performance could regress if, for example, an artifact of our implementation prevents the contract library from performing an optimization.

The case studies we use to evaluate our system are from the gradual typing performance benchmark initially developed by Takikawa et al. [24]. Three of the case studies—Gregor, Snake, and Tetris—are from the original version. The fourth, Acquire, is from a more recent draft version of the same benchmark [12]. Gregor is an adaptation of a Racket library for manipulating date and time values. Snake and Tetris are both simulations of the classic games, which were first used

in the Soft Contract Verification [17] benchmark. Finally, Acquire is an offline adaptation of a networked simulation of a board game.

Our selected case studies were originally written in Racket and the gradual typing performance benchmark's authors migrated the programs to Racket. Additionally, the programs all originally used Racket's contract library. During the migration, the contracts were either removed or had their assertions inlined into the program. For our experiments, we add the removed or inlined contracts back into the program in their original form.

In the following sections, we present the results of our evaluation.

5.3 Migration Effort

If migrating realistic contracts requires excessive changes to the program, our system fails to implement practical gradual typing. Crucially, the version of Typed Racket used for the evaluation includes the extension sketched in section 3.3. We begin by explaining the specific measurements that make up this part of the evaluation and follow with the results.

Figure 18 summarizes the effort necessary for each of our case studies. The first four rows provide the following high-level information about each of the case studies:

- Simple~Complex: Count of simple and complex contracts
- Lines: Lines of post-migration code
- $\text{Lines}_{\text{CTC}}$: Lines of contract code
- Increase: Line count difference between pre- and post-migration

We distinguish *simple* contracts from *complex* contracts. Simple contracts are contracts whose specification can be practically enforced using Typed Racket's type system. All other contracts are deemed complex. Type tests, such as `real?`, as well as contracts such as `(or/c 'red 'blue)` are simple, the latter of which corresponds to the type `(U 'red 'blue)`. General disjunctions and their corresponding union types can become impractically large, though, so we make an arbitrary choice and say that contracts corresponding to a union of more than twelve members are complex. Finally, as a clear example of a complex contract, consider one of the contracts from Acquire which tests sortedness: it succeeds for sorted lists and fails otherwise.

This cut-off corresponds to the largest union in our case studies, integers in the interval [1,12].

Lines is a simple count of the number of lines in a *migrated* case study and includes white space, comments, and other non-code lines. Increase is the change in lines after contract migration. Finally, $\text{Lines}_{\text{CTC}}$ is the lines of code that defines the contracts in a program, including functions defined at the top-level of the module that are deemed essential to a contract's specification.

	Gregor	Snake	Tetris	Acquire
Simple~Complex	60~19	30~0	49~0	55~23
Lines	1579	284	525	2777
Increase	77	13	47	-45
$\text{Lines}_{\text{CTC}}$	147	34	54	370
λ Ann	0	0	0	3
Precision	16	1	1	5
Problems	0	3	4	13
Unsupp	2	0	0	7
Fixes	2	0	0	4

Figure 18: Migration effort for each case study

In the second group of rows, the five rows of data count the number of changes due to:

- λ Ann: annotations on anonymous function arguments
- Precision: contract type being weaker than migration type
- Problems: other changes necessary to type check contracts
- Fixes: inconsistencies, obviated code, or possible-bug removal
- Unsupp: contracts that could not be migrated

λ Ann counts the number of anonymous functions whose formal arguments required an annotation. Typed Racket's type inference is only local and thus does not infer the argument types for all anonymous functions. Precision counts the number of cases where a contract was not precise enough to monitor an expression. Our type system requires, for example, that all values flowing through a domain contract must be a valid input to the function the contract monitors—if the function can only accept `Integer` values, we say that a domain contract of `real?` is not precise enough. Problems counts the number of changes required due to other problems in typechecking the contracts. Specifically, these problems required different changes than the problems related to imprecise contract types. Unsupp tracks the number of contracts that are unsupported by the current implementation and do not have a workaround, and Fixes tracks the number beneficial changes that the contracts uncovered.

increase, $\text{lines}_{\text{CTC}}$ For the line of code measurements, two case studies warrant discussion. In Gregor, the benchmark's authors only commented out its contracts, they did not entirely remove the contracts from the files. This explains why the 147 lines of contract-relevant code only incurred an increase of 77 lines to the entire program. As later

paragraphs explain in detail, the benchmark's authors did not entirely remove Acquire's contracts, either. Many of them were turned into wrapper functions that check the contract's preconditions, delegate to the wrapped function, and check the contract's postconditions. Converting these wrappers into contracts allows the same properties to be enforced with much fewer lines, which explains why the INCREASE for this case study is negative.

λ ANN Anonymous functions were rarely used for contracts in the case studies. The only instances were in Acquire, where three of the contracts required the binder to be annotated. The following contract is one of the three cases, modified to include its type annotation.

```
(and/c string? (λ ([n : String]) (<= (string-length n) 20)))
```

In all cases, the required annotation was clear from the context: each anonymous function was preceded by a simple contract that told us the type to write. Extending Typed Racket's bidirectional type checking may be able to synthesize such information, though it is unclear how often this will benefit programmers.

PRECISION For the first three case studies, the precision changes involve either the numeric tower or unions of singleton types. The numeric tower cases are all similar to the following distilled example from the Gregor Benchmark that the type system rejected.

```
(provide
 (contract-out
  [leap-year? (->/c integer/c boolean?)]))
(: leap-year? (-> Natural Boolean))
(define (leap-year? year) ...)
```

Recall that, for a function contract to monitor a function, all values passing the domain contract must be valid arguments to the function. In the contract above, though, non-`Natural` integers pass through. For the case study, the fix was to define a predicate `nonneg?` and combine it with the existing domain contract. Alternatively, we could have used the built-in contract `natural-number/c`.

For the changes related to union types, the excerpted example below from an intermediate state in Snake's migration illustrates the rejected contracts.

```
(provide
 (contract-out
  [snake (-> DIR/C (nelistof POSN/C) SNAKE/C)])
 (define DIR/C (or/c "up" "down" "left" "right"))
 (define-type Dir (U "up" "down" "left" "right"))
 (struct: snake ([dir  : Dir] [segs : (NEListof Posn)]))
 ...)
```

According to the type annotations, the constructor for `snake` structs (also named `snake`) expects to be given only `dir` values corresponding to the `Dir` type. Unfortunately, the current implementation does not generate a precise enough type for the contract `DIR/C`—that contract's output type is only `String`, whereas it needs to be the union type that `Dir` aliases. The solution here is to use Typed Racket's `make-predicate` function, which generates a predicate corresponding to the type `Dir`. This predicate has the appropriate filter and thus, as a contract, the more precise type.

For the fourth case study, Acquire, some contracts requiring precision changes involved `cons?` and others involved custom predicates being used as contracts. The contracts using `cons?` did so to quickly test that a purported list was non-empty. Any function monitored with `cons?` as its domain contract, though, is not guaranteed to get a proper list—which is precisely why our type system rejects these contracts. Changing the contract from `cons?` to `(and/c cons? list?)` solves this problem. The contracts involving predicates required annotating the domain with a more precise type, as a filter for the more precise type could not be produced.

PROBLEMS The remaining general problems encountered while typechecking Snake and Tetris are nearly identical. Contracts describing `struct` values using `struct/c` are currently unsupported, and both case studies use those to describe the shape of their core datastructures. Additionally, both use Racket's contract library's `contract-out` sub-form—itself called `struct`—for attaching contracts to struct constructors and accessor functions, which is also currently unsupported. All is not lost: the way these contracts are used does allow for a tedious workaround, ensuring that identical invariants hold. It is not clear if this workaround is sufficiently general.

Unlike the other two case studies, Acquire's contract migration problems are more diverse. The interesting issues come from two sources: a polymorphic predicate that checks sortedness and translating `->*` to `->i`. The problem with the sorted contract was that it could not be migrated to Typed Racket—it was polymorphic over key and comparison function, yet the default key function caused problems for typechecking. Finally, the remaining problem arose due to a lack of support for `->*`. All contracts using `->*` must be translated to the corresponding `->i` contract because of our current implementation. Thankfully, this workaround is fully general. All `->*` contracts can be translated to use `->i`.

UNSUPP Two case studies had various contracts that could not be supported at all. The issues behind these unsupported contracts are limitations in the current implementation. In Gregor, one of its modules tries to monitor two identifiers required from a separate module,

which our current implementation does not support—only identifiers defined in the current module can be monitored. For Acquire, one of the unsupported contracts uses `any` to specify no constraints on a function's range and another contract uses `unsupplied-arg?` to see if an optional argument was given. Additionally, because contracts on polymorphic functions are not currently supported, four contracts on Acquire's auxiliary functions were not migrated.

FIXES The fixes to Gregor are related to one of its representation of timezones, integers in the range [-64800, 64800]. The type of this contract is `(Con Any Integer)` yet it was the domain contract for two functions with domain `Natural`—thus, our type system forbids this contract from monitoring that function. This inconsistency made us reexamine the function's purpose and in fact the function's domain should have been annotated as `Integer`.

The remaining fixes were all in Acquire. During Acquire's initial migration, many of its contracts were turned into wrapper functions that check the pre-conditions, delegate to the wrapped function, and check the post-conditions. We left these wrapper functions intact while we added the original contracts back into the program, but the changes we made to the types of functions those contracts relied on (which the wrappers also relied on) resulted in type errors due to incorrect assumptions in the wrappers. These errors were all in code that was not exercised by the benchmark.

Finally, another fix in Acquire was discovered dynamically. The problem was in a predicate that checks various properties of a hash table. Specifically, this predicate intended to check that a hash table contained particular strings for its keys and that it associated particular integers to those keys. The problem, though, was that it checked whether the *keys* were integers (in addition to checking that they were strings). Using this predicate as a contract caused valid hash tables to be reported as invalid, raising blame. The original benchmark did not use this predicate, otherwise this problem would have likely not existed.

5.4 MEASURING PERFORMANCE

Migrating untyped Racket programs to Typed Racket, even programs that include contracts, should not incur unreasonable slowdowns. Our performance evaluation investigates whether contracts in Typed Racket run slower than the same contracts in Racket.

For each of the case studies in section 5.2, we measure the running time with contracts and without contracts for two configurations: the fully untyped configuration, labeled R, and the fully typed configuration, labeled TR. The running time for the fully untyped with contracts is labeled R_{CTC} and the fully typed with contracts is labeled TR_{CTC}.

	Gregor	Snake	Tetris	Acquire
R	550	537	676	125
TR	572	622	709	255
R_{CTC}	843	5389	6047	142
TR_{CTC}	909	6294	8166	282
Profiled time in contracts (ms)				
R_{CTC}	320	3926	4583	18
TR_{CTC}	330	4446	5312	15

Figure 19: Benchmark runtimes (ms)

Additionally, we measure time spent in contracts using the contract profiler. Because of the overhead of profiling and other unknown factors, these numbers will not necessarily be the delta between the other two runtime measurements.

To remove confounds, both programs are identical up to type annotations and minor contract details, such as the previously mentioned use of `make-predicate`. Additionally, the benchmark's authors had to insert casts to get certain parts of the program to typecheck. Typed Racket's casts are implemented using contracts, which, if left in, would unfairly increase the time spent in contracts for the typed program. To remove that confounding factor, we modified Typed Racket to not insert these contracts for casts.

We obtain the results in figure 19 from averaging the running time of ten executions of each version of each configuration. All programs were run using Racket version 6.6 with minor modifications to the contract system's syntax properties and a version of Typed Racket that was extended to support the contract library. [1] Each execution was run individually on a laptop with a 2.6 GHz dual-core Intel Core i5 running OS X 10.11.

The final two rows of the table are most significant in determining if contracts in Typed Racket run slower than in Racket. Gregor and Acquire both have negligible differences between their typed and untyped contracts. Interestingly, Acquire's results indicate that it spent less time in typed contracts than in untyped.

Snake and Tetris, though, do not fare as well. We focus on Tetris because its typed contracts consumed 15% more time than its untyped contracts, whereas Snake's only consumed 13% more time. To try to understand the source of Tetris' slowdown, we looked at the per-contract runtime reported by the contract profiler. One source appears to be the contract on the comparison function `block=?`, which accounted for about 1000ms more time in Typed Racket than in Racket—in Typed Racket, it contributed to roughly 2600ms of contract execution time.

1 The experimental setup can be found at `https://github.com/btlachance/ms-thesis-experiments`.

Unfortunately, it is unclear why this slowdown exists. When typed and untyped code interact, Typed Racket inserts contracts to enforce the type system's invariants. The aforementioned function `block=?`, though, does not appear to cross such a boundary, ruling out that possible source of overhead. Due to time constraints, we are unable to further investigate the root cause of this issue.

5.5 DISCUSSION

Our evaluation results are mixed. On the one hand, we do not support all of the features we set out to support, even the ones we conceivably could. On the other hand, this only led to a handful of problems for our case studies. Similarly, although some migrations require changes to many of their contracts, the overall effort is minimal. The runtime performance is also mixed. The typed contracts for two of our benchmarks consumed negligibly more time than the untyped contracts. The other two benchmarks' typed contracts were worse, consuming roughly 15% more time than the corresponding untyped contracts.

6 RELATED WORK AND CONCLUSION

This chapter first presents the existing body of related work in order to place our work in the larger context of contract systems for typed languages. We conclude with thoughts on future research and directions for gradually-typed contracts.

6.1 RELATED WORK

The most closely related work is at the intersection of types and contracts, with the first source being Findler and Felleisen [7]. Their calculus was the first to enforce higher-order contracts and enforce blame for them. Additionally, they show how to adapt their calculus to an implementation using wrapper lambda terms. Our formalism is a descendant of their work but differs at the type-level. Specifically, our type system includes subtyping and the necessary machinery to support idiomatic Racket contracts, whereas their calculus only needed to focus on simple types.

Later, Hinze et al. [13] show how to build a Haskell contract library with combinators for dependent functions, pairs, lists, and contract composition (analogous to Racket's `and/c`). Crucially, their system allows polymorphic functions as contracts. Like other functions in Haskell, these contracts can then use type-classes to define a contract, such as a contract for ordered lists that relies on the *Ord* class. Our work could not be implemented as a library in Typed Racket, requiring changes to its internals. Additionally, it does not directly address how to use polymorphic functions as contracts.

Dimoulas and Felleisen [3] presents an extension of PCF, dubbed CPCF, for contracts in a simply typed language with recursion and blame. Using observational equivalence, they formalize different definitions of contract satisfaction and show how their technique allows them to explore the design space of contract satisfaction. Although the work in this thesis builds on CPCF, that is as far as the relationship goes. We offer no formal claims, only our language's design embodied in a calculus.

Chitil [1] also presents a contract library for Haskell. It has richer combinators than in Hinze et al. [13] that are fundamentally lazy. A lazy function contract, for example, will not force the monitored function to evaluate more than necessary. This means that their contract library does not allow dependent function contracts, as the dependent range contract could explore the function on an unrestricted number

of arguments. Our calculus does not consider laziness, even though Racket's contract system does support limited forms of lazy checking.

Finally, *manifest* contracts as presented in Greenberg et al. [11] (and similar work in Knowles and Flanagan [14]) offer an alternative view of contracts. In such a system, a contract's assertions are lifted to the type level. These systems permit expressing the type of natural numbers as $\{x : Int \mid x \geqslant 0\}$, the type of the successor function as $\{x : Int \mid x \geqslant 0\} \mapsto \{y : Int \mid y = x + 1\}$, and so on. Manifest contracts, though, are about using powerful types that are like contracts, while our system (a latent contract system) is about using types to reason about contracts.

6.2 FUTURE WORK

Our system has a number of limitations, such as an incomplete coverage of the contract library's features. Removing these limitations and supporting more of Racket's contract library in Typed Racket would be useful. It would also be generally useful for exploring the applicability of our type system, but for future work we focus on four specific limitations.

Polymorphism is not considered by our type system. This includes using polymorphic functions as contracts, contracts to enforce polymorphism (for example, those that use runtime sealing), as well as contracts on polymorphic functions. Due to Typed Racket's subtype relation, our implementation has some support for the latter: the Acquire case study has a polymorphic function, `sorted`, which is monitored by a contract. These features certainly require more investigation into their benefit and their utility.

Accessing all of Racket's contract machinery is currently not supported. Racket programmers can create new contracts with the function `make-contract`, which provides hooks to integrate with the rest of the contract library. As an example, the these hooks allow for heuristically comparing two contracts to determine whether one accepts fewer values than another. This is used in the contract library's random generation and testing functionality.

We have not addressed dynamically enforcing our contract types, a key part of a gradually typed language. One reason for leaving this out of our design is that preliminary investigations into enforcing these types raises concerns about its performance due to the accumulation of additional checks. Although supporting this could enable using custom contracts defined in vanilla Racket, like those mentioned in the previous paragraph, it is hard to find idiomatic uses of this feature.

Our language's design ensures that contract applications are safe, but our evaluation uncovered reasonable contracts that our type system rejected. To recap, one function in Acquire could only be called with lists but it initially had a domain contract of `cons?`. Our current type system disallows this, as our type rules require this domain con-

tract to ensure that only lists get through—in some sense, our design ignores the guarantees already provided by the function's type. While developing such a type system seems feasible, we are unsure whether new type rules like this would increase the difficulty in migrating Racket programs with contracts.

6.3 conclusion

Contracts and types go hand in hand: contracts permit expressing invariants using the language of the rest of the program that are otherwise cumbersome, if not impossible, to write in a language's type system. Types, on the other hand, verify rich guarantees about all possible executions of a component before execution. These tools are readily available in traditional untyped and typed languages. Therefore, gradual type systems should provide those benefits to users of gradually-typed languages in an incremental manner, but research in gradually typed languages has yet to provide a way for idiomatic, untyped contracts to be migrated to the typed world. This work presents a language design that fills this gap. We validate the design at a preliminary level by implementing it in Typed Racket and by evaluating its effectiveness through a series of case studies.

BIBLIOGRAPHY

[1] Olaf Chitil. Pactical Typed Lazy Cotacts. In *Proc. ACM Intl. Conf. Functional Programming*, 2012.

[2] Ryan Culpepper and Matthias Felleisen. Fortifying Macros. In *Proc. ACM Intl. Conf. Functional Programming*, 2010.

[3] Christos Dimoulas and Matthias Felleisen. On Contract Satisfaction in a Higher-Order World. *Trans. Programming Languages and Systems* 33(5), 2011.

[4] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete Monitors for Behavioral Contracts. In *Proc. European Sym. on Programming*, 2012.

[5] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[6] Daniel Feltey. Gradual Typing for First-Class Modules. MS dissertation, Northeastern University, 2015.

[7] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 48–59, 2002.

[8] Matthew Flatt and Matthias Felleisen. Units: Cool Modules for HOT Languages. In *Proc. ACM Conf. Programming Language Design and Implementation*, 1998.

[9] Matthew Flatt, Robby Findler, and Matthias Felleisen. Scheme with Classes, Mixins, and Traits. In *Proc. Asian Sym. Programming Languages and Systems*, 2006.

[10] Matthew Flatt and PLT. Reference: Racket. PLT Design Inc., PLT-TR-2010-1, 2010. `https://racket-lang.org/tr1/`

[11] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts Made Manifest. In *Proc. ACM Sym. Principles of Programming Languages*, 2010.

[12] Ben Greenman, Asumu Takikawa, Max S. New, Jan Vitek, and Matthias Felleisen. Is Sound Gradual Typing Dead? Unpublished manuscript, 2016.

[13] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed Contracts for Functional Programming. In *Proc. Sym. Functional and Logic Programming*, 2006.

[14] Kenneth Knowles and Cormac Flanagan. Hybrid Type Checking. *Trans. Programming Languages and Systems* 32(2), 2010.

[15] Bertrand Meyer. Applying "Design by Contract". *Computer* 25(10), 1992.

[16] Bertrand Meyer. *Eiffel: The Language*. Pentice-Hall, Inc., 1992.

[17] Phúc C. Nguyễn, Sam Tobin-Hochstadt, and David Van Horn. Soft Contract Verification. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 139–152, 2014.

[18] Gordon Plotkin. LCF Considered as a Programming Language. Theoretical Computer Science, 1977.

[19] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Proc. Wksp. Scheme and Functional Programming*, 2006.

[20] T. Stephen Strickland and Matthias Felleisen. Contracts for First-Class Modules. In *Proc. Dynamic Languages Symposium*, 2009.

[21] T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical Variable-Arity Polymorphism. In *Proc. European Sym. on Programming*, 2009.

[22] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and Impersonators: Run-Time Support for Reasonable Interposition. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, 2012.

[23] Asumu Takikawa. Gradual Typing for First-Class Classes. PhD dissertation, Northeastern University, 2016.

[24] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max New, Jan Vitek, and Matthias Felleisen. Is Sound Gradual Typing Dead? In *Proc. ACM Sym. Principles of Programming Languages*, pp. 456–468, 2016.

[25] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual Typing for First-Class Classes. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 793–810, 2012.

[26] Sam Tobin-Hochstadt. Typed Scheme: From Scripts to Programs. PhD dissertation, Northeastern University, 2010.

[27] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: from Scripts to Programs. In *Proc. Dynamic Languages Symposium*, pp. 964–974, 2006.

[28] Sam Tobin-Hochstadt and Matthias Felleisen. Logical Types for Untyped Languages. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 117–128, 2010.

[29] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as Libraries. In *Proc. ACM Conf. Programming Language Design and Implementation*, pp. 132–141, 2011.

[30] Jesse A. Tov and Riccardo Pucella. Stateful Contracts for Affine Types. In *Proc. European Sym. on Programming*, 2010.