Nested and Dynamic Contract Boundaries[⋆]

T. Stephen Strickland and Matthias Felleisen

PLT @ Northeastern University
{sstrickl,matthias}@ccs.neu.edu

Abstract. Previous work on software contracts assumes fixed and statically known boundaries between the parties to a contract. Implementations of contract monitoring systems rely on this assumption to explain the nature of contract violations and to assign blame to violators. In this paper, we explain how to implement arbitrary, nested, and dynamic contract boundaries with two examples. First, we add nestable contract regions to a static, first-order module system. Second, we show that even a dynamic, higher-order, and hierarchical module system can be equipped with software contracts that support precise blame assignment.

1 Contracts for Modules

PLT Scheme [1] comes with a widely used contract system for specifying behavioral (functional) properties of module exports and imports. Roughly speaking, a behavioral software contract imposes restrictions on the domain and range of a function that flows from one module to another. If the function does not produce the kind of values promised in a contract, the run-time monitoring system raises a contract exception and blames the server module for exporting an ill-behaved function. Conversely, if the client module applies an imported function to values that fail to satisfy the domain contract, the run-time system blames the client for not living up to its promises.

Unlike other systems for monitoring behavioral software contracts [2–17], PLT Scheme's contract system does not restrict contracts to first-order functions and methods. Instead, programmers may formulate contracts for all kinds of values, including higher-order values [18]. The module system, however, is restrictive. In particular, modules are merely first-order namespaces, without mechanism for nesting them or linking them in a recursive fashion. Naturally programmers chafe under this module system and call for more flexibility.

At the same time, the theory of contracts assumes fixed and statically known boundaries between contract parties. Contract implementations combine compilers that can determine the parties to each contract from the source text with a run-time checking system that exploits this knowledge for blaming violators.

In this paper, we relax these restrictions and show how to add arbitrary contract boundaries to the PLT Scheme module system and how to implement

[⋆] This research was partially supported by the US Air Force Office of Scientific Research and the National Science Foundation.

contracts for its units, a higher-order, hierarchical, and dynamic component system [19, 20]. Sections 3 and 4 make up the core of the paper. Both use the same organization, explaining the nature of the contract boundary first and, based on that, its enforcement. In particular, the third section adds nested contract regions to PLT Scheme's module system, while the fourth section explains contracts for its unit system. In section 5 we revisit the design decisions concerning blame assignment with a side-by-side comparison of the two extensions. Finally, the last section compares this paper with a concurrent publication on a theoretical model of a structural (ML-like) module system [21]. The paper starts with a section that briefly describes the existing module system and its contracts.

2 Static Modules in PLT Scheme

PLT Scheme provides static modules that are neither nestable nor first-class. Figure 1 contains an example consisting of two modules. A module may export values via their names[1] through the use of the **provide** form. Another module makes use of these values via a **require** form. Modules may not require each other (or themselves) in a cyclic fashion.

server

```
#lang scheme
(define (sqrt n) ...)
... (sqrt 3) ... (sqrt 0) ...
(provide sqrt)
```

client

```
#lang scheme
(require server)
... (sqrt 3) ...
... (sqrt 0) ...
```

Fig. 1. Example modules

Findler and Felleisen's work on higher-order contracts [18] presents a model for adding contract checks to such a module system. The implementation of this model in the PLT Scheme module system operates via the **provide/contract** form, which specifies a sequence of names paired with contracts. The main idea behind this implementation is that module interfaces serve as natural *contract boundaries*. Values that flow across a contract boundary are checked for the specified properties, while values that stay on one side remain unchecked.

A contract boundary in PLT Scheme brings together two contract parties. One is the exporting server module; the other one is the importing client module in which the name is used. The contract monitoring system uses the module names to assign blame when it discovers and signals contract violations. In analogy to type theory, we call the name of the server module a *positive blame label* of a specific contract, and the name of the client module a *negative blame label.*

[1] In PLT Scheme, modules can also export values for use at compile-time as well as run-time. Here we focus on run-time values.

The design decision of not monitoring the uses of contracts within a module is due to both software engineering considerations and compilation issues. In particular, we consider the inside of a module a space where programmers should trust their own instincts, even allowing temporary violations of contracts; the alternative poses severe challenges known as the callback problem [22]. Furthermore, monitoring the uses of contracts within a module would negatively affect opportunities for tail-call optimizations, an essential element of functional and object-oriented program design.

server

```
#lang scheme
(define (sqrt n) ...)
... (sqrt 3) ... (sqrt −1) ...
(provide/contract
 [sqrt (→ positive? positive?)])
```

client

```
#lang scheme
(require server)
...
... (sqrt 3) ...
... (sqrt −1) ...
```

Fig. 2. Modules with an example contract

Figure 2 displays contracted versions of the modules from figure 1. Specifically, the *server* module exports *sqrt* with a contract that demands positive numbers as inputs and promises the same for the results. The use of *sqrt* with -1 in *client*—see boxed code—triggers a contract violation error that blames *client* for applying *sqrt* to an inappropriate value. In contrast, the gray-shaded call to *sqrt* on -1 within *server* is not monitored and so does not signal a contract error; presumably *server* knows how to deal with complex numbers.

3 Nested Contract Regions

While programmers appreciate the rationale of not monitoring contracts within a module, they also commonly wish to isolate regions that they can protect with contracts, even within modules. This is especially true for debugging sessions or for modules that grow into large bodies of code. Unlike the static module system in section 2, one such region may be nested within another, or a module may contain several parallel regions. In response to this request, we introduce *contract regions*. The first subsection introduces the idea via a series of examples, which at the same time suggests design desiderata for this new feature. The second subsection describes our implementation.

3.1 The Pragmatics of Contract Regions

Consider the module fragment on the left-hand of figure 3. It displays the fragment of a module that contains the definition of the *serve* function, which implements a basic webserver, and two applications; the second one is faulty, applying

```
#lang scheme
(define (serve p)
  (let ([s (tcp-listen p)])
    (handle-request s)
    (serve p)))
...
(serve 8080)
...
(serve 80) ;; error, no superuser permissions
```

```
#lang scheme
(with-contract serve
  ([serve (→ high-tcp-port? void?)])
  (define (serve p)
    (let ([s (tcp-listen p)])
      (handle-request s)
      (serve p)))
...
(serve 80) ;; contract violation
```

Fig. 3. Modules and contract regions

serve to a low TCP port, that is, a TCP port with a numeric value less than 1024, for which the program does not have the necessary permissions.

To protect *serve* from such errors *within* the module, a programmer could create a separate static module that defines *serve* and export it with an appropriate contract. Of course this strategy imposes a high overhead. Worse, it may not work if the to-be-separated parts are mutually referential, because the PLT Scheme module system does not support mutually recursive linking.

Instead we introduce the **with-contract** form. The right-hand side of figure 3 shows the simplest way to use the **with-contract** form. It consists of three pieces: a name, which is used to assign blame; a sequence of contracted variables; and a sequence of definitions. Every variable listed in the second part must have a definition in the third part, but there may be additional definitions in this third part that do not come with a contract.

Since the **with-contract** syntax is heavy-weight for single definitions, we introduce a convenience abbreviation named **define/contract**. The syntax of **define/contract** is similar to that of PLT Scheme's **define**, except for the addition of a contract before the body of the definition. With **define/contract**, the code in the above figure would look like this:

```
#lang scheme
(define/contract (serve n)
  (→ high-tcp-port? void?)
  (let ([s (tcp-listen p)]) (handle-request s) (serve p))))
...
```

The abbreviation is translated into a contract region that uses the name of the defined value as the blame label.

The **with-contract** form introduces a block of definitions. As such, it can be used in any syntactic position where definitions are allowed. It does *not* introduce a new lexical scope, meaning both contracted and uncontracted definitions are accessible in the surrounding lexical scope. Conversely, all internal definitions may access any external definitions at will.

Not surprisingly, the name of the contract region serves as the positive blame label for all contracts listed in the **with-contract** form. The question is what

```
#lang scheme
(define/contract (encode key msg)
  (→ prime? string? string?)
  ...)
(define/contract (send-msg msg)
  (→ string? void?)
  ... (encode 20 m) ...) ;; contract error, blame: ???
```

Fig. 4. Two parallel contract regions

we should consider as the client of the region and what we should use as the negative blame label for the contracts. Obviously, if there is no other contract region in the module, the rest of the module is the client. If, however, a module contains several regions or regions are nested, we have a choice to make.

Consider the two regions in figure 4. It appears convenient to use *send-msg* as the most precise negative blame label for the use of *encode* here. Put differently, all parallel contract regions could be considered as clients of each other. Although this design choice is appealing, it is inappropriate. Instead we say that a contract region introduces a contract boundary between itself and its surrounding context. If this surrounding context allows the contracted value to flow into other contract regions, those regions are clients of the context not the original contract region. For our above example, this means that when *send-msg* is called, the enclosing module context is blamed for the misuse of *encode*. We revisit this choice at length in section 5.

```
#lang scheme
(with-contract serve
  ([serve (→ high-tcp-port? void?)])
  (define (serve p)
    (let ([s (tcp-listen p)]) (handle-request s) (serve p)))
  (define (serve-80) (with-su (serve 80))) ;; ok
(serve 8080)
(serve 80) ;; contract error, module misused serve
```

Fig. 5. External vs. internal uses

Just as with module-based contracts, a contract region does not monitor internal uses of the contracted definitions. To illustrate this point, take a look at the *serve-80* function in figure 5, which correctly sets up the necessary permissions for accessing low TCP ports with the *with-su* form. The external uses of *serve* are checked according to the contract, and so the last call to *serve* still fails, but the internal use in the definition of *serve-80* is not restricted and succeeds.

```
#lang scheme
(with-contract student
  ([id? (→ any/c boolean?)] [make-student (→ string? id? student?)]
   [student-name (→ student? string?)] [student-id (→ student? id?)]
   [student? (→ any/c boolean?)])
  (define (id? n) (and (natural-number? n) (< n 1000000000)))
  (define (make-student s n) (list s n))
  (define (student-name s) (first s))
  (define (student-id s) (second s))
  (define (student? s)
    (and (list? s) (= (length s) 2) (string? (first s)) (id? (second s)))))
```

Fig. 6. Operations for student records

The introduction of a distinction between internal and external uses of contracted variables naturally raises the question where the contract itself lives. In order to explain this issue, we use the example in figure 6. The contract region contains several functions to operate on record-like list values that represent student information. These records contain two fields: names, represented as strings, and nine-digit numbers used for unique identification. The predicates *id?* and *student?* are used within the contracts for the other operations. However, these predicates are also contracted, and thus we must decide whether the uses within those contracts must be checked. Since the contracts are a part of the **with-contract** form itself, we consider it reasonable to treat such uses as internal uses, meaning they are not protected.

```
#lang scheme
(define (salary? s) (or (natural-number? s) (eq? s #f)))
(define (make-employee s n) (list s n))
(define (employee? s)
  (and (list? s) (= (length s) 2) (string? (first s)) (salary? (second s))))
(define (employee-name e) (first e))
(define (employee-salary e) (second e))
;; test data
(define loe1 (list (make-employee "Bob" 45000) (make-employee "Stan" 50000)))
(define loe2 (list (make-employee "Ana" 50000) (make-employee "James"  #f)))
...
(define (get-salaries loe) (map employee-salary loe))
...
```

Fig. 7. An evolving payroll program

Protection Against Externally Defined Values In contrast to modules, contract regions can exchange values in both directions, and this has serious implications for the contract system. In particular, the creator of a contract region may wish to protect it from values that flow in from its context. Thus, our **with-contract** construct supports this form of protection, too.

Consider the code snippets in figures 7 and 8, representing two stages in the evolution of an application. The application stores employee records, which are similar to the student records we saw earlier. Originally, employee salaries were always numbers, and the given definition for *get-salaries* sufficed. However, records are not removed immediately when the employee leaves the company. For now, an interim solution has been found in which the *salary* field is set to #f, but *get-salaries* has not been updated to report a salary of 0 for these cases.

```
...
(define/contract (payroll loe)
  (→ (listof employee?) number?)
  (foldl + 0 (get-salaries loe)))
...
(payroll loe1)
...
...
(payroll loe2) ;; error
...
...
```

```
...
(define/contract (payroll loe)
  (→ (listof employee?) number?)
  #:freevars ([get-salaries
               (→ (listof employee?)
                  (listof number?))])
  (foldl + 0 (get-salaries loe)))
...
(payroll loe1)
...
(payroll loe2) ;; contract error
```

Fig. 8. Payroll contracts

A new function *payroll* is added that retrieves the current payroll total for the company: see the left-hand side of figure 8. When the value #f used for James's salary flows into the payroll function, however, it causes an error. After all, the programmer of *payroll* expected *get-salaries* to return a list of numbers, but it doesn't always do so.

To express this kind of expectation and to pinpoint the contract violator, the **with-contract** and **define/contract** forms come with optional contracts on their free variables, which are introduced via the keyword #:freevars. These contracts affect all uses of the listed free variables within the contract region. With this feature, the programmer may add a contract for the *get-salaries* function as shown in the right-hand side of figure 8. This contract fails on the second use and appropriately blames *payroll* for providing a bad value for *get-salaries*.

3.2 Implementing Contract Regions

The addition of nested contract regions poses novel problems for the implementation of contract monitoring. Specifically, the revised contract monitor must be able to retrieve the blame label for the current contract region, replace uses of contracted definitions outside of the region with guarded versions, and replace uses of contracted free variables inside the region with guarded versions.

We describe the compilation of contract regions in terms of substitution. The macro-based compiler [23, 24] inspects the list of contracted names. For each contracted name, it chooses a fresh name and substitutes that name for uses of the original name within the body of the contract region. The compiler also replaces the original name where it is bound in its definition. These substitutions ensures that the definition of the contracted name is exchanged for a definition of the fresh name, and that all uses of the contracted name refer instead to the uncontracted, fresh name.

At this point, the compiler could create a new definition of the contracted name that wraps the value associated with the fresh name with a contract. This would, however, disassociate the value internal to the contract region, referred to by the fresh name, and the external value, referred to by the contracted name. If either code internal or external to the contract region mutates their respective binding, that mutation is not reflected in the other portion of code.

To allow for checked mutation, our system binds the contracted name to a syntax transformer that expands each use to a guarded use of the fresh name. Doing so ensures that the use evaluates to the current value of the fresh name, and it also enforces that the contract system checks the current value for adherence to the contract. Furthermore, the syntax transformer also allows the compiler to track mutation of the contracted name. When this occurs, the compiler generates an expression that instead mutates the fresh name, guarding the new value with the contract. Here our system uses the context as the positive blame and the contract region as the negative blame, as the new value flows into the contract boundary during mutation.

To protect free variables with contracts, a similar set of substitutions is performed. The compiler produces a fresh name for each protected free variable, and creates a syntax transformation for that fresh name that expands references into guarded references and mutations into guarded assignments to the free variable.

The macros for contract regions need to access the blame label for the context. For this, we turn to *syntax parameters* [1], which provide a mechanism for temporarily setting compile-time values for the macro expansion of a specific region of code. Our system binds a syntax parameter to the appropriate blame label during the expansion of the body of a contract region; otherwise, the syntax parameter is instead set to the blame label for the current module.

4 Contracts for Nominally Linked Units

In addition to static first-order modules, PLT Scheme supports a separate component system, called units [19, 20]. Units are analogous to ML's functor module system [25, 26] and the mixins and traits of OO programming languages [27, 28].

Roughly speaking, the unit system supports hierarchical programming with first-class components. Each unit is parameterized over its linking context; each unit also exports a set of names. The unit system supports two operations on units: linking and invoking. A number of units with matching signatures can be linked in a graph-based fashion; the result is a new unit with its own parameterization over its future contexts, which flow into its constituent units, and its own exports, which flow out of its constituent units. A unit whose parameterization is empty may be invoked, meaning the unit's body is evaluated sequentially. Units are first-class values and may even be loaded at run-time. They co-exist with modules and as such may flow across module boundaries.

Understanding unit signatures is key to understanding units as contract boundaries. The first subsection therefore describes signatures, which name collections of variables for import or export from a unit. It also introduces the addition of contracts for signatures. The second and third subsections then present examples of uncontracted and contracted units. The last subsection explains how to implement units as contract boundaries in PLT Scheme and how the addition of contracts affects our implementation.

```
#lang scheme
(define-signature world^
  (key?
   key=?
   big-bang))


(define-signature client^
  (world?
   tock
   clack))
...
```

```
#lang scheme
(define-signature world/c^
  ((contracted
     [key? (→ any/c boolean?)]
     [key=? (→ key? key? boolean?)]
     [big-bang (→ any/c void?)])))

(define-signature client/c^
  ((contracted
     [world? (→ any/c boolean?)]
     [tock (→ world? world?)]
     [clack (→ world? any/c world?)])))
...
```

Fig. 9. Signatures with contracts

4.1 Signatures and Contracts

A unit *signature* is a named collection of variables. Units use sequences of signatures to specify their imports and exports. An exported signature can satisfy an import requirement for another unit only if that unit imports the signature with the *same* name. In other words, the unit system uses nominal matching.

For our examples, we use the two signatures on the left side of figure 9.[2] These signatures describe interfaces that are useful for implementing interactive animations in a world-passing style [29]. The *world^* signature contains three names: *key?*, which is a predicate that determines whether a value is a keyboard event; *key=?*, which is an equivalence predicate; and *big-bang*, which launches an

[2] The ^ character at the end of signature names is merely a convention.

animation when applied to a world (*world?*). The *client^* signature also contains three names: *world?*, which is a predicate on worlds; *tock*, which is an event handler for clock ticks, mapping worlds to worlds; and *clack*, which is an event handler for keyboard events, from worlds and keyboard events to worlds.

Naturally, programmers wish to express such specifications as contracts in order to protect units. We have therefore extended the language of signatures with the **contracted** keyword, which combines signature variables with contracts. The right hand side of figure 9 shows the contracted versions of the signatures. Notice that signature contracts can involve elements of the same signature.

4.2 Units without Contracts

The import signatures of a unit introduce bindings for all their variables for the unit body; conversely, if a unit exports a signature, it must define all the variables listed in the signature. Figure 10 contains some sample units[3] that utilize the uncontracted signatures from the preceding subsection.

```
...
;; get-last-key, a primitive, returns #f if no key was pressed
;; since the last call; otherwise it returns the pressed key
(define-unit world@ (import client^) (export world^)
  (define (key? k) (memq k (list "up" "down")))
  (define (key=? ke1 ke2) (string=? ke1 ke2))
  (define (big-bang w)
    (let ([ke (get-last-key)])
      (if ke (big-bang (clack w ke))
        (begin (sleep .1) (big-bang (tock w)))))))
;; Here a world is a number that represents the height of
;; a rocket on a 500 pixel high canvas (not shown here).
(define-unit client@ (import world^) (export client^)
  (define (world? n) (and (integer? n) (>= n 0) (<= n 500)))
  (define (tock n) (+ n 10))
  (define (clack n ke)
    (cond [(key=? ke "up") (+ n 10)]
          [(key=? ke "down") (− n 10)]))
  (big-bang 0))
...
```

Fig. 10. Example interactive animation units

When **compound-unit** is used to link a collection of units, the exported definitions from one unit are typically used to satisfy import requirements for one or more of the other units. Thus we can link *client@* and *world@* like this:

[3] As with ^, the use of @ is a naming convention for units.

```
(define pgrm@
  (compound-unit/infer (import) (export) (link world@ client@)))
```

The “infer” suffix is a variant of **compound-unit** that infers how to wire up the exports and imports of the constituents.

In general, the result of linking is a unit that has its own list of imports and exports and whose body is a sequence of the constituent unit bodies in the order listed in the **link** clause. The exports of the compound unit are satisfied from the exports of the constituent units, and the imports of the compound unit may be used to satisfy imports of the constituents. In contrast to modules, units can thus be compounded hierarchically, and they may refer to each other’s exports and imports in a mutually referential manner.

Finally, units with empty **import** signatures can be invoked, e.g.

```
(invoke-unit pgrm@)
```

The effect is to execute the body of *world@*, which consists entirely of definitions, and then to execute the body of *client@*, which calls *big-bang*.

4.3 Units with Contracts

The use of signatures with contracts turns units into contract regions and their boundaries into contract boundaries. In the following code, the definitions of *world@* and *client@* differ from the earlier definitions only in the import/export specification, and so we elide the bodies:

```
(define-unit world@ (import client/c^) (export world/c^) ...)
(define-unit client@ (import world/c^) (export client/c^) ...)
```

When we link *client@* and *world@* and invoke the result:

```
(invoke-unit
  (compound-unit/infer (import) (export) (link world@ client@)))
```

then *client@* is blamed if either *tock* or *clack* cause the world to become negative or increase beyond 500.

The signatures *world/c^* and *client/c^* illustrate that a contract in a signature may refer to other elements from the same signature. Thus, we must decide how these contracts interact with the linked units’ contract boundaries. In particular, we must decide whether references to signature elements within contracts are guarded or not. For the purposes of this paper, we consider all signature contracts as occurring within the importing unit’s contract boundary and therefore the compiler guards all uses of contracted signature elements inside those contracts. This ensures that exported variables are not misused by the contracts and concurs with our implementation strategy.[4]

[4] This design decision is overly conservative and deserves to be revisited once we have enough experience with our new contract system. Furthermore the current contract system does not permit programmers to use elements from one signature in a different signature for the specification of contracts. Extending the contract system in this direction may also force us to revisit the design decision on how to check contracted functions within contracts.

4.4 Implementing Units as Contract Boundaries

Adding contracts to the unit system poses several challenges. First, units do not enter a contract with a known party; instead they specify via signature contracts what they expect from their context. Second, the same unit may be linked to several different units at run-time and may thus enter contracts with several different parties. Hence, the compiler cannot pass on enough knowledge about the contract parties to the run-time checks. Third, due to nominal linking, a compound unit may only link constituent units whose contracts are identical. Therefore blame labels can be exchanged as units are linked.

The first part of this section describes how units are implemented in PLT Scheme. The second part explains the addition of signature-based contracts to the existing implementation. The third part covers additional features of the unit system.

Units in PLT Scheme The current unit system in PLT Scheme follows the model by Owens and Flatt [19] for first-class modules. In this model, signatures are matched nominally when units are linked. The implementation exploits this nominal matching to provide inference for linking.

The compiler[5] translates a unit into a thunk that is hidden in a unique structure value. On application, the thunk returns two values:

- a mapping from exports to reference cells, and
- a function that implements the body of the unit. The function consumes a mapping from imports to reference cells; it returns the last value computed by the unit body.

In the unit's body, the compiler replaces uses of imports with accesses to the import mapping. To each definition of an exported item, the compiler adds an assignment to the appropriate reference cell. Once an export cell is set, its value never changes.

A unit invocation invokes the thunk to obtain a export mapping and a body function. The latter is then applied to an empty import mapping, which evaluates the unit body.

Since units are represented as thunks, the compiler translates a **compound-unit** form to a thunk, too. This thunk performs the following operations:

1. It applies the thunks for the constituent units and collects the resulting export mappings and body functions.
2. It constructs an export mapping for the compound unit from the collected export mappings.
3. It creates a body function that consumes the import mapping of the compound unit. For each linked unit in listed order, this new body function:
   (a) creates an import mapping from the compound unit's import mapping and the collected export mappings of the other units, and

[5] The unit system is actually implemented as a library based on the PLT macro system, though it is impossible for a programmer to discover this programmatically.

   (b) applies that unit's body function to the created import mapping.
4. It returns the new export mapping and body function.

Contracts in Signatures Since units must agree on their shared signatures *by name* and since we add contracts to signatures, linked units automatically agree on all of the contracts of the shared variables. That is, unlike a module that contains two parallel contract regions, a compound unit cannot possibly link two units whose contracts don't match, as in figure 11. Thus, it is impossible for the linker to assume any responsibility for contract errors. Put differently, there is no need for checking contracts within the compound unit and it need never be blamed. Put positively, our implementation limits blame to the exporting unit and the importing unit.

```
#lang scheme
(define-signature lexer^ ((contracted [lex (→ string? (listof token?))])))
(define-signature lexer2^ ((contracted [lex (→ input-port? (listof token?))])))
(define-signature parser^ ((contracted [parse (→ string? ast?)])))
(define-unit lexer@ (import) (export lexer2^)
  (define (lex str) ...))
(define-unit parser@ (import lexer^) (export parser^)
  (define (parser str) (let ([tokens (lex str)]) ...)))
(compound-unit/infer (link lexer@ parser@))
```

Fig. 11. Mismatched signatures and contracts

The key to our addition of contracts is to separate the translation of contracted signature variables from those of uncontracted ones. For contracted exports, the compiler generates code that sets the cell for the exported value to a structure with two fields:

– one for the value of the exported variable, and
– one that uniquely identifies the exporting unit, i.e., its blame label.

When the compiler encounters a contracted import, it deconstructs this kind of structure and retrieves the contract from the imported signature. From these two pieces, the compiler constructs an appropriate guard expression for the imported value. This contract-guard uses the export blame label for positive blame report and the importing blame label for negative blame reports.[6]

[6] Roughly speaking, it applies two projections to the value: one for its "elimination" (negative) and one for its "introduction" (positive). If something goes wrong with the negative position, the client is blamed; otherwise the server is blamed. For details on the general idea, see Findler and Blume's report [30].

Structural Linking and a Contract Combinator The unit system supports two more important linguistic constructs whose full descriptions are beyond the scope of this paper. One form, *unit/s*, provides a mechanism for linking units structurally. This provides backwards compatibility for use with an early implementation of units in PLT Scheme [20].

The *unit/s* form takes import and export specifications as well as a unit value and creates a new unit value. Its imports and exports must structurally match the imports and exports of the given unit value; the resulting unit value uses the given imports and exports and the given unit's body. Since this operation on units changes the import and export signatures, the contracts on the imported and exported values may be inappropriate for the original unit. Hence, the compiler must introduce contract checks into the result of *unit/s* that blames the new unit value when contract mismatches occur, instead of allowing either the original unit value or any unit with which it is linked to be blamed.

The other form, *unit/c*, is a new form of contract specification, since units are first-class values that also can cross contract boundaries. Technically, the contract combinator *unit/c* is used in contracts to express contracts on units. A contract on units is essentially a sequence of contracts for a unit's exports and imports. We implement this operation as a projection[6] on unit values, which means that it takes a unit value as input and returns a new unit value that monitors the flow of values across the unit boundary.

Both of these forms require similar changes to the unit implementation, because both introduce structural notions of matching a unit's exports to another unit's imports. Structural units, in turn, are a central piece of related work, which we briefly compare to this work in section 6.

5 A Question of Blame

Now that we have described two new contract extensions—contract regions and unit contracts—we are in a good position to compare and contrast the blame story for the two. Examine the modules in figure 12. The module *regions* contains two contract regions: *server*, which provides the implementation of a webserver, and *client*, which (mis-)uses that implementation. Similarly, the module *units* contains two units, *server@* and *client@*, which are in a relationship that is analogous to that of *server* and *client*.

When evaluated, both modules result in a contract violation. In *regions*, the module itself is blamed, since it is the context of the contract region *server*, whereas in *units*, *client@* is blamed. The inquisitive reader may be surprised that in the former case, the contract system did not blame *client* instead, which would be a more specific region. After all, the purpose of blame assignment is to assist programmers with debugging, calling for the most specific blame justifiable.

One reason for this design decision is that only parties that explicitly enter into a contract should be blamed for bad behavior. In the second module, the various units, via signatures, enter into contracts for both their imports and regions

```
#lang scheme
(with-contract server
  ([serve (→ high-tcp-port? void?)])
  ...)
...
...
...
...
...
...
(with-contract client
  ([...])
  ... (serve 80) ...)
```

units

```
#lang scheme
(define-signature web^
  ((contracted
    [serve (→ high-tcp-port? void?)])))
(define-unit server@
  (import) (export web^)
  ...)
(define-unit client@
  (import web^) (export)
  ... (serve 80) ...)
(invoke-unit
  (compound-unit/infer
    (link server@ client@)))
```

Fig. 12. A comparison between contract regions and units

exports. That is, *server@* (respectively, *client@*) declares that the exported (respectively, imported) function *serve* is contracted through its use of the signature *web^*. Since both parties have agreed to the contract, the two units are the only sources of blame.

In the first module, only *server* declares a contract on the function *serve*. This agreement is with its context, i.e., the rest of the module *regions*. Thus only *server* or *regions* can be blamed if part of the contract is violated. If *client* had declared the same contract on *serve* via #:freevars, then it, too, would have agreed to the contract and could be blamed appropriately.

regions2

```
#lang scheme
(with-contract server
  ([serve (→ high-tcp-port? void?)])
  ...)
...
(with-contract client
  ([...])
  #:freevars ([serve (→ tcp-port? void?)])
  ... (serve 80) ...)
```

Fig. 13. Regions with differing contracts

Then again, *client* doesn't have to specify the same contract as *server*. Thus, in figure 13, neither contract region should be at fault, as both regions use the *serve* function according to their own contract. Instead, the fault lies with the context that ties the two regions together. It allows the value *serve* to flow from one region to another even though the two impose distinct requirements at their respective boundaries. This is analogous to the behavior of contracts for structurally linked units, which we briefly mentioned in conjunction with *unit/s* and which we discuss more extensively in the section on related work.

In fact, the first example can be seen as a special case of the second, if we treat all uncontracted free variables flowing into a contract region as if they had the implicit contract *any/c*, i.e., the most permissive contract. Thus, having the context of the contract region serve as the negative blame leads to a consistent handling of blame for contract regions.

6 Related Work

Our paper benefits from two pieces of related work. First, a parallel paper [21] explores the theory of contracts for the units described by Flatt and Felleisen[20], i.e., units with structural signature matching. Matching signatures structurally requires much deeper changes to the compiler and the run-time environment than PLT Scheme's unit system with nominal matching. Most importantly, it introduces a third party of potential blame—the compounding unit—and therefore demands contract machinery for linking.

Structural signature matching is closely related to the world of ML-like module systems based on functors and structures. As such, the parallel paper directly applies to this world. In contrast, the implementation presented here is much closer to the world of nominal interfaces from OO programming languages and should therefore carry over to contracts for mixins [27] and traits [28].

By inheritance, our paper extends the work by Findler and Felleisen [18] on higher-order contracts for static and global contract boundaries. Our implementation heavily relies on Findler's work with Blume [30], which is the current theoretical underpinning for contracts. It explains contracts as pairs of projections and is the model for the implementation of contracts in PLT Scheme.

Historically, the notion of contracts and modules is due to Parnas [31] though he did not coin the phrase "contract." Meyer's "design by contract" work introduces this terminology [13]; his work on Eiffel popularized the idea in the object-oriented community.

7 Conclusion

Software contracts enable programmers to protect collections of functions and methods with simple, executable descriptions of expected behavior. Contract monitoring ensures that all values that flow into and out of a protected region satisfy its stated boundary invariants. When the contract monitor discovers a contract violation, it must be able to pinpoint the guilty party and explain the nature of the violation. Doing so is critical for the debugging process.

Given the growing importance of contracts, our work provides the important generalization of introducing nested and dynamic contract boundaries. Technically, this paper introduces hierarchical contract regions for static modules and contract boundaries for a hierarchical and dynamic module system. We conjecture that future work on contract boundaries can benefit from either of those two or a mix of them. Our implementation is available with the current release of PLT Scheme (http://www.plt-scheme.org/).

Acknowledgments We gratefully acknowledge comments and suggestions from Robby Findler and the anonymous reviewers for IFL on early drafts of this paper.

References

1. Flatt, M., et al.: PLT Scheme. Reference Manual PLT-TR2009-reference-v4.2.1, PLT Scheme Inc. (January 2009) http://plt-scheme.org/techreports/.
2. Beugnard, A., Jézéquel, J.M., Plouzeau, N., Watkins, D.: Making components contract aware. IEEE Software (June 1999) 38–45
3. Carrillo-Castellon, M., Garcia-Molina, J., Pimentel, E., Repiso, I.: Design by contract in Smalltalk. Journal of Object-Oriented Programming **7**(9) (1996) 23–28
4. Duncan, A., Hölzle, U.: Adding contracts to Java with Handshake. Technical Report TRCS98-32, The University of California at Santa Barbara (December 1998)
5. Edwards, S., Shakir, G., Sitaraman, M., Weide, B., Hollingsworth, J.: A framework for detecting interface violations in component-based software. In: Proceedings of the Fifth International Conference on Software Reuse, IEEE (June 1998) 46–55
6. Helm, R., Holland, I.M., Gangopadhyay, D.: Contracts: specifying behaviorial compositions in object-oriented systems. In: Proceedings of Object-Oriented Programming, Systems, Languages, and Applications. (1990) 169–180
7. Kim, M., Kannan, S., Lee, I., Sokolsky, O., Viswanathan, M.: Java-MaC: a runtime assurance tool for Java. Electronic Notes in Theoretical Computer Science **55**(2) (2001) 218–235
8. Kramer, R.: iContract: the Java design by contract tool. In: Proceedings of Technology of Object-Oriented Languages and Systems. (August 1998) 295–307
9. Karaorman, M., Hölzle, U., Bruno, J.: jContractor: a reflective Java library to support design by contract. In: Proceedings of Meta-Level Architectures and Reflection. Volume 1616 of LNCS., Springer (July 1999) 175–196
10. Leavens, G.T., Leino, K.R.M., Poll, E., Ruby, C., Jacobs, B.: JML: notations and tools supporting detailed design in Java. In: Proceedings of Object-Oriented Programming, Systems, Languages, and Applications, Companion. (2000) 105–106
11. Luckham, D.C.: Programming with Specifications: An Introduction to Anna, a Language for Specifying ADA Programs. Springer-Verlag (1990)
12. Microsoft Corporation: Microsoft C# Language Specifications. Microsoft Press (2001)
13. Meyer, B.: Applying design by contract. IEEE Computer **25**(10) (October 1992) 40–51
14. Weck, W.: Inheritance using contracts and object composition. In: Proceedings of the Workshop on Components-Oriented Programming. (1997) 384–388
15. Gomes, B., Stoutamire, D., Vaysman, B., Klawitter, H.: A Language Manual for Sather 1.1. (August 1996)
16. Plösch, R., Pichler, J.: Contracts: from analysis to C++ implementation. In: Proceedings of Technology of Object-Oriented Languages and Systems. (August 1999) 248–257

17. Ruby, C., Leavens, G.T.: Safely creating correct subclasses without seeing superclass code. In: Proceedings of Object-Oriented Programming, Systems, Languages, and Applications. (October 2000) 208–228
18. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of the International Conference on Functional Programming. (October 2002) 48–59
19. Owens, S., Flatt, M.: From structures and functors to modules and units. In: Proceedings of the International Conference on Functional Programming. (September 2006) 87–98
20. Flatt, M., Felleisen, M.: Units: Cool modules for HOT languages. In: Proceedings of Programming Language Design and Implementation. (June 1998) 236–248
21. Strickland, T.S., Felleisen, M.: Contracts for first-class modules. In: Proceedings of the Fifth Dynamic Languages Symposium. (October 2009) 27–38
22. Szyperski, C.: Component Software. Addison-Wesley (1997)
23. Flatt, M.: Composable and compilable macros: You want it *when?* In: Proceedings of the International Conference on Functional Programming. (October 2002) 72–83
24. Culpepper, R., Tobin-Hochstadt, S., Flatt, M.: Advanced macrology and the implementation of Typed Scheme. In: Proceedings of the Scheme Workshop, Université Laval Technical Report DIUL-RT-0701. (September 2007) 1–14
25. Leroy, X.: Manifest types, modules, and separate compilation. In: Proceedings of Principles of Programming Languages. (January 1994) 109–122
26. Harper, R., Lillibridge, M.: A type-theoretic approach to higher-order modules with sharing. In: Proceedings of Principles of Programming Languages. (January 1994) 123–137
27. Flatt, M., Findler, R.B., Felleisen, M.: Scheme with classes, mixins, and traits. In: Proceedings of the Asian Symposium on Programming Languages and Systems. (November 2006) 270–289
28. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. In: Proceedings of the European Conference on Object-Oriented Programming. Volume 2743 of LNCS., Springer (July 2003) 248–274
29. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: A functional i/o system or, fun for freshman kids. In: Proceedings of the International Conference on Functional Programming. (October 2009) 47–58
30. Findler, R.B., Blume, M.: Contracts as pairs of projections. In: Proceedings of Functional and Logic Programming. Volume 3945 of LNCS., Springer (April 2006) 226–241
31. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM **15** (1972) 1053–1058