Contracts for First-Class Modules *

T. Stephen Strickland Matthias Felleisen
PLT @ Northeastern University, Boston, Massachusetts
{sstrickl,matthias}@ccs.neu.edu

Abstract

Behavioral software contracts express properties concerning the flow of values across component (modules, classes, etc) interfaces. These properties are often beyond the reach of theorem provers and are therefore monitored at run-time. When the monitor discovers a contract violation, it raises an exception that simultaneously pinpoints the contract violator and explains the nature of the violation.

Currently contract monitoring assumes static module interfaces. Specifically, the contract compiler partitions a contract into disjoint obligations for the static modules that communicate to an interface. At run-time, the information is used for catching and explaining contract violations. While static modules suffice for many situations, first-class modules—such as the units provided by PLT Scheme—support the dynamic and multiple linking that is often required in open software systems. The problem is, of course, that in such a world, it becomes impossible to tell from the source program alone which components have agreed to which contracts.

In this paper, we develop the semantic framework of monitoring contracts for dynamic modules. We establish the internal consistency of the semantics, and we sketch an implementation based on our experience of equipping PLT Scheme with such contracts.

Categories and Subject Descriptors D [*2*]: 2 Modules and interfaces; D [*2*]: 4 Programming by contract

General Terms Reliability

Keywords contracts, first-class module systems

1. Contracts and First-Class Components

With behavioral[1] software contracts programmers express strong logical assertions about the flow of values from one program component (module, class) to another. A contract monitoring system turns these assertions into dynamic checks that raise exceptions as soon as there is evidence of a contract violation. The exception messages should pinpoint the violators and include explanations of the violations as hints for the debugging process. In sum, contracts turn interfaces into monitored boundaries between components and blames those parties that send inappropriate values across these boundaries.

Existing contract systems combine compiler support with dynamic checking to implement this informative style of monitoring. The contract compiler partitions each contract into obligations for each party to the agreement. It maps these pieces into appropriate dynamic checks and equips the checks with information about which static component promised to live up to which obligations.

About five years ago, we implemented a powerful contract system [Findler and Felleisen 2002] for PLT Scheme's static module system [Flatt et al. 2009]. Programmers use the contract system for a range of purposes. Some attach type-like contracts to interfaces to make up for the lack of a type system; others formulate complex conditions for all kinds of values, including higher-order functions.

Beyond static modules, PLT Scheme offers a higher-order component system, dubbed the unit system [Flatt and Felleisen 1998, Owens and Flatt 2006]. In contrast to the hardwired import specifications of static modules, units are connected via *external* linking specifications. Furthermore, programs can consist of deeply nested hierarchies of modules that are extended at run-time. A code inspection of libraries and software produced in PLT Scheme shows that programmers frequently use units for parameterizing libraries over their context and for creating open software systems (e.g., web servers). PLT Scheme's unit system shares these attributes with other component features, e.g., mixins [Flatt et al. 1998, 2006], traits [Schärli et al. 2003, Allen et al. 2008, Odersky 2009], and functors in ML [Milner et al. 1990].

The addition of external linking is a critical step toward a true component market. A client component C that depends on the services of some component S relies only on some interface specification I, e.g., a module signature or a Java-style interface. It is in no way connected to the actual implementation of S. If S turns out to be faulty, it can be replaced with some component S' that implements the same interface I *without modifying anything else.*

When it comes to behavioral contracts, external linking poses two major problems. First, as explained, blame assignment in existing contract monitoring system assumes that each contract comes with exactly one "server module," that is, one module exports a value that satisfies the contract to which many other "consumer modules" subscribe. Second, blame assignment also relies on the identification of the parties to a contract at compile time. In a component system, however, client modules do not directly specify to which server module they are connected. As a matter of fact, they may be connected to *several distinct* server modules in one program, and any of them may be loaded at run-time.

Over the past year we have implemented the first contract system for PLT Scheme's units. This paper explains the design with an operational model so that others can adapt our experience to mixins, traits, or other dynamic component systems. The paper starts with a brief introduction to units. Section 3 adds contracts to units on an informal basis. Section 4 develops the generalized contract system. The rest of the paper sketches an implementation strategy and provides some additional context.

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

[1] For contract terminology, see the work of Beugnard et al. [1999].

DLS'09, October 26, 2009, Orlando, Florida, USA.

2. Units

Flatt and Felleisen [1998] introduced units into PLT Scheme as a dynamic and dynamically-typed analog to SML's functors. Unlike the latter, units are first-class values that programs can manipulate at run-time (compound, invoke, splice). Furthermore, while functors act as transformation functions on atomic modules, linking specifications for units may arrange units in potentially cyclical graphs. Last but not least, programs may decide at run-time which units to link in, and indeed, they may even load units from the file system after the execution has started.

The original model of units [Flatt and Felleisen 1998] consists of two separate pieces: a reduction semantics and a type system. In this section, we acquaint the reader with units and their untyped operational model. This section ignores types because they are orthogonal to a study of contracts.

p ::= (**program** d ... e)
d ::= (**define** x e)
e ::= x | n | b | s | (λ (x) e) | (e e) | (op e e) | (**if** e e e)
op ::= + | - | * | = | <=
b ::= #t | #f

Figure 1. Syntax for the base model

2.1 The Core Language ...

Following tradition [Leroy 1994], the original unit model is parameterized over the core language, making only some minimal assumptions such as the existence of mutually recursive definitions.

For concreteness, we start from the higher-order functional core language shown in figure 1. A program in this language is a list of definitions followed by an expression. The top-level definitions define distinct variables.

In addition to functions as first-class values, the language includes three kinds of atomic values:

- numbers (n),
- strings (s),
- and booleans (b) using Scheme notation.

It also supports operations on numbers and conditional expressions.

While PLT Scheme includes assignment statements, destructive structure operations, and powerful control operations, our model omits such imperative extensions because their inclusion would only increase the notational overhead without any additional benefits. As Flatt and Felleisen [1998] explained, units are entirely orthogonal to the existence of imperative features in a language.

2.2 ... plus Units

To add units to our model language, we extend the syntax with the additional terms in figure 2. The **unit** form creates an atomic component, which is a first-class value. A **compound-unit** operation synthesizes a unit hierarchy from two existing units, usually resolving some imports in the process. Finally, the **invoke-unit** operation evaluates the body of a unit that requires no more imports.

In general, a unit consists of three pieces: an import specification, an export specification, and a body. The first two are just sequences of names. The body of a **unit** is, like the body of a program, a series of distinctly-named definitions followed by an expression. Imported variables are in scope in the body of the unit; they are resolved to values during unit linking. Each exported variable must be defined in the body of the unit. Imported variables cannot be directly exported, because this might lead to definitional cycles.

e ::= **....** | (**begin** e e) | (**invoke-unit** e) | u | cu
u ::= (**unit** (**import** x ...) (**export** x ...) d ... e)
cu ::= (**compound-unit** (**import** x ...) (**export** x ...) e e)

Figure 2. Syntactic extensions for units

Here is a program fragment, extracted from a full-fledged PLT Scheme program,[2] that uses all unit forms and operations:

```
(program

  (define world@
    (unit (import tock clack) (export key=? big-bang)
      (define key=? (λ (ke1) (λ (ke2) ...)))
      (define big-bang (λ (w) ... tock ... clack ...))
      "the default expression"))

  (define client@
    (unit (import key=? big-bang) (export tock clack)
      (define tock (λ (w) ...))
      (define clack (λ (ke) (λ (w) ... (key=? ke ...) ...)))
      (big-bang ...)))

  (define cmain@
    (compound-unit (import) (export) world@ client@))

  (invoke-unit cmain@))
```

The unit named *world@*[3] defines and exports two functions: *key=?* (an equality predicate on keyboard events) and *big-bang* (which launches an interactive graphical program when applied to some value). In return, the *world@* unit imports from its clients two functions: *tock* (called in response to clock ticks) and *clack* (called in response to keyboard events).

The second definition introduces and names a unit that plays the role of a potential client to the *world@* unit. Dually to *world@*, this *client@* unit imports *key=?* and *big-bang* and exports *tock* and *clack*. Also note that this unit's last piece is an expression that invokes the imported *big-bang* function.

Next, the third definition illustrates the compounding of two units into a single module. The **compound-unit** expressions allows the exports of *world@* to flow into *client@* and vice versa. Because the two satisfy each other's export and import needs, the resulting compound unit does not import anything else. It could export values, if so desired, but this is independent of the rest of the program.

Finally, the last expression invokes the import-less *cmain@* unit. This action runs all the expressions of the given unit, which in this case are all the expressions of the two units that went into the construction of *cmain@*.

In PLT Scheme, compound units may link as many units as needed. Pragmatically this multi-unit linking mechanism is highly useful. From the perspective of a model, it adds nothing but notational complications. Also, PLT Scheme's unit system supports an operation for invoking a unit and splicing its exported definitions into the local lexical scope. Handling this case, even with the later addition of contracts, is straightforward and does not add much to the exposition.

[2] This program is a stripped down, unit-based version of PLT Scheme's universe.ss teachpack, a library for teaching functional GUI programming to novices [Felleisen et al. 2009b]. The complete program is beyond the scope of this paper, but it is typical of the kind of linking that units allow.

[3] The use of "@" is a PLT Scheme convention that helps readers identify the names of units.

P[(**invoke-unit** (**unit** (**import**) (**export** x_e ...) (**define** x_d e_d) ... e_b))] [Invoke]
$\longmapsto$ add-defs⟦P[e_b{(x_d ...) := (x_n ...)}], ((**define** x_n e_d{(x_d ...) := (x_n ...)}) ...)⟧
where (x_n...) fresh

(**compound-unit** (**import** x_i ...) (**export** x_e ...) [Compound]
(**unit** (**import** x_{i1} ...) (**export** x_{e1} ...) (**define** x_{d1} e_{d1}) ... e_1)
(**unit** (**import** x_{i2} ...) (**export** x_{e2} ...) (**define** x_{d2} e_{d2}) ... e_2))
$\longrightarrow$ (**unit** (**import** x_i ...) (**export** x_e ...) (**define** x_m e_{d1}{(x_{d1} ...) := (x_m ...)}{(x_{d2} ...) := (x_n ...)}) ...
(**define** x_n e_{d2}{(x_{d2} ...) := (x_n ...)}{(x_{d1} ...) := (x_m ...)}) ... ρ⟦x_e, ([x_{d1} x_m] ... [x_{d2} x_n] ...)⟧ ...
(**begin** e_1{(x_{d1} ...) := (x_m ...)}{(x_{d2} ...) := (x_n ...)} e_2{(x_{d2} ...) := (x_n ...)}{(x_{d1} ...) := (x_m ...)}))
where (x_i ... x_{e1} ... x_{e2} ...) distinct, {x_{i1} ...} ⊆ {x_i ... x_{e2} ...}, {x_{i2} ...} ⊆ {x_i ... x_{e1} ...},
{x_e ...} ⊆ {x_{e1} ... x_{e2} ...}, (x_m...), (x_n...) fresh

add-defs⟦(**program** d_1 ... e), (d_2 ...)⟧ = (**program** d_1 ... d_2 ... e)
ρ⟦x, ([x_{o1} x_{n1}] ... [x x_n] [x_{o2} x_{n2}] ...)⟧ = (**define** x x_n)

Figure 5. Reduction rules for units

pv ::= n | b | s
nuv ::= pv | (λ (x) e)
nlv ::= pv | u
v ::= pv | u | (λ (x) e)
dv ::= (**define** x v)
P ::= (**program** dv ... D d ... e) | (**program** dv ... E)
D ::= (**define** x E)
E ::= [] | (E e) | (v E) | (op E e) | (op v E) | (**begin** E e)
| (**if** E e e) | (**invoke-unit** E) | CU
CU ::= (**compound-unit** (**import** x ...) (**export** x ...) E e)
| (**compound-unit** (**import** x ...) (**export** x ...) v E)

Figure 3. Values and evaluation contexts for units

((λ (x) e) v) $\longrightarrow$ e{x := v} [Beta]
(op v_1 v_2) $\longrightarrow$ δ⟦op, v_1, v_2⟧ [Delta]
(**if** #t e_1 e_2) $\longrightarrow$ e_1 [If-True]
(**if** #f e_1 e_2) $\longrightarrow$ e_2 [If-False]
(**begin** v e) $\longrightarrow$ e [Begin]
P[x] $\longmapsto$ P[v] [Var]
where (**define** x v) ∈ P[x]

Figure 4. The semantics of the core language

2.3 Dynamic Semantics

A reduction semantics [Felleisen et al. 2009a] classifies the syntactic elements of a language as values and computations and then specifies via relations on the syntax how the latter reduce to the former. Here the specification of the relations involves so-called evaluation[4] contexts, turning the relations into functions on the syntax. In sum, a reduction semantics specifies a machine whose states are complete programs and whose instructions are functions from programs to programs; the machine reduces complete programs to programs in canonical form, known as values.

[4] The hole of an evaluation context identifies the leftmost-outermost position in a term. Thus the use of evaluation contexts introduces the standard reduction function of a calculus as a semantics [Plotkin 1975].

Our semantics consists of two pieces: the usual reduction semantics for the core language and an extension that specifies the meaning of units and operations on units. Figure 3 classifies a subset of our language's syntax as values and defines evaluation contexts for programs, expressions, and unit expressions.

Next, figure 4 introduces the reduction relations for the core language. The relations for the core model are straightforward. Relations that use the $\longrightarrow$ notation occur within program contexts $P[\,]$ but have no need to either examine their context or change it in any way. In contrast, the [Var] relation shows how free variables are resolved by finding the corresponding top-level definition in the program via a condition on the surrounding evaluation context.

The two reduction relations in figure 5 explain how units are compounded and invoked. The [Invoke] rule describes the process of invoking a unit value. The invocation process involves renaming the internal definitions and their references to fresh variables and lifting the resulting definitions to the program level. Then the body expression of the unit is evaluated in place of the **invoke-unit** form.

The [Compound] rule shows how **compound-unit** takes two unit values and links them together into a single unit. Like a **unit** form, the **compound-unit** form also has a list of imports and a list of exports. Each imported variable in a constituent unit must be listed in either the exports of the other constituent unit or the imports of the **compound-unit** form, but not both. This means that the imports of the **compound-unit** form and the exports of each unit value must be distinct. Each exported variable of the **compound-unit** form must correspond to an export of either of the two unit values.

The result of the **compound-unit** form is a new unit value that has the same imports and exports as the **compound-unit** form:

```
(compound-unit (import x y) (export a)
  (unit (import x y) (export a)
    (define a (λ (z) (+ x y)))
    "default 1")
  (unit (import x a) (export b)
    (define b (λ (w) (- x (a w))))
    "default 2"))
⟶
(unit (import x y) (export a)
  (define a (λ (z) (+ x y)))
  (define b (λ (w) (- x (a w))))
  (begin
    "default 1"
    "default 2"))
```

This new unit value contains the definitions from both component units. It also contains the expressions from both units. These two expressions are sequenced using the **begin** form, corresponding to the order of the two units in the **compound-unit** form.

When the definitions from both units are combined, all the definitions and their uses from each unit are renamed. The following program illustrates why:

```
(program
  (define server1@
    (unit (import) (export n) (define n 3) 1))
  (define server2@
    (unit (import) (export n) (define n 4) 1))
  (define client@
    (unit (import n) (export) (* n 2)))
  (define comp1@
    (compound-unit (import) (export) server1@ client@))
  (define comp2@
    (compound-unit (import) (export) server2@ client@))
  (compound-unit (import) (export) comp1@ comp2@))
```

Both *server1@* and *server2@* provide different implementations of *n*, but neither *comp1@* nor *comp2@*, which link these units to *client@*, export *n*. Then *comp1@* and *comp2@* are linked in the body of the program. If we did not rename the internal definitions of *n* and their uses in each unit at this step, then the resulting unit would contain two conflicting definitions of *n*.

In addition to renaming all the definitions, the reduction rule also renames the uses of each unit's exports in the other unit and constructs the export definitions for the combined unit by defining each export to be the value of the appropriate renamed variable.

2.4 Unit Pragmatics

Units are particularly useful when a program needs the functionality of one and the same client module in the context of several different service modules or when a program must decide at run-time which module to link in.

Let us illustrate each case with examples from the code of the DrScheme program development environment [Findler et al. 1997]. DrScheme supports several text books with teaching languages. A teaching language restricts the syntax of some production language for the purposes of a particular part of a book. Furthermore teaching languages come in series—typically BSL (for beginning student language), ISL (intermediate), and ASL (advanced)—so that students gradually see more and more of the full language.

Each teaching language is parsed to a common intermediate representation. The results flow into an interpreter (and other tools) with annotations so that, for example, run-time error messages use concepts known to students of the given language and nothing else. In this context, the common interpreter and each parser come in the form of a unit: *interpreter@*, *bsl-parser@*, *isl-parser@*, and *asl-parser@*. The last three export a *parse* function; the first one imports it and exports an *eval* function.

DrScheme's test suite ensures that certain programs from, say, BSL, have the same outcome in all three levels. To achieve this, the test suite links *interpreter@* to all three parser units and runs all three *eval* functions on the same program: see figure 6. The sketch shows how **compound-unit** links three parsers to one interpreter to obtain three complete evaluators. Via renaming, all three evaluators are linked to a testing unit and become available simultaneously.

Naturally DrScheme allows programmers to switch from one teaching language to another without shutting down and restarting the IDE. That is, the system decides at run-time which units should be linked in. The IDE uses the currently selected language to decide which language front-end to link with the interpreter:

```
(compound-unit (import) (export)
  ;; the parsers
  [PBSL (bsl-parser@)]
  [PISL (isl-parser@)]
  [PASL (asl-parser@)]
  ;; the evaluators
  [BSL (interpreter@ PBSL)]
  [ISL (interpreter@ PISL)]
  [ASL (interpreter@ PASL)]
  ;; the testing
  [TEST ((unit (import (prefix "bsl:" eval)
                       (prefix "isl:" eval)
                       (prefix "asl:" eval))
               (export)
               (define prog1 '((define i (λ (x) x)) (i 10)))
               ...
               ...(check
                     10
                     (bsl:eval prog1)
                     (isl:eval prog1)
                     (asl:eval prog1)) ...)
         BSL ISL ASL)])
```

Figure 6. Linking clients to many servers

```
(define (get-interpreter current-selection)
  (compound-unit (import) (export run)
    (cond
      [(eq? current-selection 'beginner)
       bsl-parser@]
      [(eq? current-selection 'intermediate)
       isl-parser@]
      [(eq? current-selection 'advanced)
       asl-parser@])
    interpreter@))
```

Last but not least, DrScheme can also launch arbitrary front-ends by loading units at runtime:

```
(define (get-interpreter parser-path)
  (compound-unit (import) (export run)
    (load parser-path) interpreter@))
```

For further examples of how units provide useful software abstractions, see Findler and Flatt [1998], Flatt and Felleisen [1998], and Graunke's papers on the PLT web server [Graunke et al. 2001].

2.5 Types

Flatt and Felleisen [1998] equipped their unit model with a type system in order to allow a comparison with a statically typed module system, such as SML's. In this paper, we have no need for a type system, because types are orthogonal to contracts. With contracts a programmer can express most of the types directly, and, of course, contracts also permit the specification of logical assertions that are beyond the constraints of a regular type system.

3. Contracts

While contracts for first-order functions in the context of static modules are easy to understand, contracts for higher-order functions and contracts for a dynamic module system demand a gradual introduction. We therefore use this section to introduce contracts for functions and higher-order functions before we provide a series of examples of units with contracts. Dually, this section develops desiderata for our contract system for units, including the ability to introduce contracts on a gradual basis.

3.1 Contracts for Functions

Contracts on first-order functions come with an intuitive semantics that is easy to explain. Assume we have a module *S* (service provider) that exports a function *f* with a contract and an importing module *C* (client). The contract for *f* is specified in the interface between *C* and *S* and uses this type-like notation:

[*f* (⟶ *n? p?*)]

It says that the arguments for *f* must satisfy the (user-defined) predicate *n?*, i.e., if *a* is an argument to *f*, (*n? a*) must evaluate to true; also the results of *f* are to satisfy the *p?* predicate.

In this context, consider the following three possibilities:

- *S* calls *f* with an argument that doesn't pass the *n?* predicate or *f* produces results for which *p?* returns false. Since the call to *f* occurs *within the boundaries of S*, no exception is raised.
- *C* applies *f* to an argument for which *n?* doesn't hold. In this case, the contract monitoring system raises an exception pointing at *C* as a client that misapplies *f*.
- A call to *f* in *C* produces a result for which *p?* returns false. Now, the contract system blames *S* for not living up to its promise to produce *p?* results via *f*.

PLT Scheme uses this type-like syntax to specify contracts, because it easily generalizes to higher-order functions. For example a function-consuming function *F* may come with this contract:

[*F* (⟶ (⟶ *ppp? n?*) *p?*)]

This contract implies that *F*'s argument satisfies the contract (⟶ *ppp? n?*). Naturally, it is impossible to check that a function always produces results that satisfy *n?* and that it is always applied to arguments for which *ppp?* holds. After all, in a higher-order language, it is impossible to predict all call sites or the legality of all results.

Findler and Felleisen [2002] resolved this seeming paradox with two assumptions and one immediate consequence. The first assumption is that higher-order contracts aren't violated until a first-order predicate can produce evidence of the violation. The second assumption is that only the two parties to the contract can possibly enforce it, i.e., no other party should take the blame for violations. The key consequence of these assumptions is that the positive positions of a function contract are obligations for the server module *S* and that the negative positions turn into obligations for the client *C*. Given a function contract the *positive* positions are those that are to the left of an even number (including 0) of arrows; the remaining positions are negative. The above examples use *n?*, *p?*, and *ppp?* to suggest positive and negative positions.

Here is a concrete example:

[*encrypt* (⟶ *string?* (⟶ *natural? prime?*) *string?*)]

This higher-order contract specifies a function that consumes two arguments: a string and a function from the natural numbers to prime numbers. The result is a string. Naturally if the server applies the prime-choosing function to `"hello world"`, *S* is blamed for misapplying its second argument. If, however, the prime-picking function produces `9`, the contract monitor blames *C* for supplying a bad argument.

Before moving on, note that tracking blame information in a higher-order world requires non-trivial machinery, and a full explanation is beyond the scope of this example section. For some details see the next section.

3.2 Units with Contracts

Units are naturally the parties to contracts in our world, though for notational simplicity we do not introduce explicit interface definitions. Instead, we attach contracts directly to the import and export specifications of a unit, e.g.,

```
(define convert@
  (unit (import)
        (export [convert (⟶ string→number real?)])
        (define convert (λ (n) ...))
        "default"))
```

This first sample contracts states that *convert* maps strings convertible to numbers[5] to reals. Of course, the unit could also export the predicate that protects convert, like this:

```
(define convert@
  (unit (import) (export
                  [convertible? (⟶ any/c boolean?)]
                  [convert (⟶ convertible? real?)])
        (define convert (λ (n) ...))
        (define convertible (λ (x) ...))
        "default"))
```

To use the services of *convert*@, we must write a unit that imports the *convert* function:

```
(define cvrt-client@
  (unit (import [convert (⟶ string→number real?)])
        (export)
        (convert 4)))
```

Since both units use the same contract, we can obviously link them:

```
(compound-unit (import) (export) convert@ cvrt-client@)
```

Furthermore, it is also obvious that *convert*@ is responsible for the positive positions of the contract, and *cvrt-client*@ is responsible for the negative positions.

Units don't have to use the same contract to be compatible, however. A programmer may just know that two units should be able to collaborate even if the contracts aren't quite the same:

```
(define string-client@
  (unit (import [convert (⟶ string? real?)]) (export)
        (convert "hellow")))
```

Here *string-client*@ imports a *convert* function that is expected to accept all strings. A programmer who believes that *string-client*@ does not apply *convert* to inconvertible strings may still link the two units:

```
(compound-unit (import) (export) convert@ string-client@)
```

Of course, an invocation of the result raises a contract error.

The question is which unit should be blamed. The *convert*@ unit provides a *convert* function from number-convertible strings to numbers, and we may assume that it fulfills its contract. The *string-client*@ unit applies *convert* to `"hellow"`, which is appropriate for its contract. Clearly, neither *convert*@ or *string-client*@ should be blamed. This leaves us with the compound unit, which becomes a third party to the contracts, an implicit adapter.

3.3 Adding Contracts Gradually

Programmers use dynamic languages because they wish to get something running quickly with as little overhead as possible. Hence language designers must be prepared to deal with programs that mix modules with and without contracts.

Let us revisit the preceding example with the contracts from *string-client*@ removed:

[5] In PLT Scheme, *string→number* translates some given string into a number, if possible, and produces false otherwise. Since all non-false values count as true, this function can act as a basic predicate, too.

```
c ::= (unit/c (import [x c] ...) (export [x c] ...)) | (-> c c) | e
u ::= (unit s (import [x c] ...) (export [x c] ...) d ... e)
cu ::= (compound-unit s (import [x c] ...) (export [x c] ...) e e)
```

Figure 7. Surface syntax extensions for contracts

```
(define plain@
  (unit (import convert) (export) (convert −4)))

(define c@
  (compound-unit (import) (export) convert@ plain@))
```

If the program also invokes *c*@, an exception is raised again.

As mentioned, we assume that *convert*@ lives up to its contract. The *plain*@ unit does not subscribe to a contract for *convert* (or anything else) and thus can't be the target of a "blame message." Again we are forced to blame the linking **compound-unit** expression, because it allowed the *convert* function, which was exported with a contract, to flow to a party that misapplies the function.

In short, we are forced to conclude:

- All units guarantee the negative positions of their import contracts and the positive positions of their export contracts.
- Compound units guarantee for their constituent units the positive positions of the import contracts and the negative positions of the export contracts.

3.4 A Contract Combinator for Units

Because units are first-class values, our language also needs a method of ascribing contracts to units in interfaces. After all, values are first-class when they can be used wherever other values are used.

Consider the following unit:

```
(define combine-client@
  (unit
   (import) (export [combine ...])
   (define combine
     (λ (u@)
       (invoke-unit
        (compound-unit (import) (export)
          u@ client@))))))
```

It exports a function that consumes a unit, links it with the above *client*@ unit, and invokes the result.

To protect the *combine* export of *combine-client*@, we need a contract combinator that guards units. This contract combinator must be able to describe those details important for interfacing with a unit, i.e., the imports and exports. In PLT Scheme, **unit/c** plays this role. For the given example, the combinator would be used as follows:

```
(export
  [combine
    (⟶ (unit/c (import)
               (export [convert (⟶ string→number real?)]))
       positive?)] ...)
```

A unit that is being guarded with a unit contract must import a subset of the imports listed by the **unit/c** expression and export a superset of the exports. Furthermore, we must decide whether the unit which results from applying a unit contract has the same imports and exports as the original unit, or whether it has the same imports and exports as those listed in the contract. Here we take the latter view, as contracts constitute a promise of how a value is utilized, and thus assuming unlisted imports or exports is inappropriate.

```
(program
  (define any/c (λ (x) #t))
  (define world@
    (unit "world"
          (import [world? (⟶ any/c any/c)]
                  [tock (⟶ world? world?)]
                  [clack (⟶ key? (⟶ world? world?))])
          (export [key? (⟶ any/c any/c)]
                  [key=? (⟶ key? (⟶ key? any/c))]
                  [big-bang (⟶ world? any/c)])
      (define key? (λ (ke) ...))
      (define key=? (λ (ke1) (λ (ke2) ...)))
      (define big-bang (λ (w) ... tock ... clack ...)) 1))
  (define client@
    (unit "client"
          (import [key? (⟶ any/c any/c)]
                  [key=? (⟶ key? (⟶ key? any/c))]
                  [big-bang (⟶ world? any/c)])
          (export [world? (⟶ any/c any/c)]
                  [tock (⟶ world? world?)]
                  [clack (⟶ key? (⟶ world? world?))])
      (define world? (λ (w) ...))
      (define tock (λ (w) ...))
      (define clack (λ (ke) (λ (w) ... (key=? ke ...) ...)))
      (big-bang 0)))
  (invoke-unit
    (compound-unit "linker" (import) (export)
      world@ client@)))
```

Figure 8. Extended example with predicate exports

4. Units with Contracts

Equipped with an informal understanding of contracts in a unit setting, we proceed to formulate a model of this world. First, we extend the syntax of our model to accommodate contracts. Second, we formulate a model of contract monitoring. Third, we state two essential theorems about the model: one concerning "type soundness" and one for "contract soundness."

4.1 Syntax for Units with Contracts

Figure 7 spells out our revisions to the syntax of section 2 that add contracts to our model. The first clause specifies contracts as unit contracts, functions contracts, or (expressions that evaluate to) arbitrary predicates. The second line adds contracts to the import and export specifications of units. The second and third line add names to both the **unit** and **compound-unit** forms to represent blame; a production system would naturally use source locations instead. Figure 8 presents our running example from section 2—a GUI module and its clients—in this revised syntax.

4.2 The Idea

Findler and Felleisen [2002] describe a model of higher-order contracts and static modules. Their model assumes that programs consist of a sequence of modules and a main expression. Contract checking in this model compiles each reference to an imported function into a guard expression. Roughly speaking guard expressions wrap imported functions with a contract and information about the parties to the contract. The wrapper knows how to check every kind of value, including higher-order functions.

In a unit world, however, the compiler lacks two critical pieces of information to use this strategy. First, while units specify their imports, they can't possibly know the contracts that the exporting service module imposes on them. Second, client units don't

$\xi[\![u]\!]$ = (**unit** s_u (**import** [x_i c_{in}] ...) (**export** [x_e c_{en}] ...) (**define** x_s "unknown")
(**define** x_f e_n) ... $\rho c[\![x_e, c_{er}, ([x_d\ x_f] \ldots), s_u, x_s]\!]$ ...
$\xi[\![e_b]\!]$ {(x_i ... x_d ...) := ((**guard** x_i c_{ir} x_s s_u) ... x_f ...)})

where u = (**unit** s_u (**import** [x_i c_i] ...) (**export** [x_e c_e] ...) (**define** x_d e_d) ... e_b),
x_s = fresh-in$[\![u]\!]$,
(x_f ...) = fresh-in$[\![u, (x_d \ldots)]\!]$,
(c_{in} ...) = ($\xi[\![c_i]\!]$ ...),
(c_{en} ...) = ($\xi[\![c_e]\!]$ ...),
(c_{ir} ...) = ($\Sigma[\![c_{in}, ([x_i\ c_{in}] \ldots), x_s, s_u]\!]$ {(x_d ...) := (x_f ...)} ...),
(c_{er} ...) = ($\Sigma[\![c_{en}, ([x_i\ c_{in}] \ldots), x_s, s_u]\!]$ {(x_d ...) := (x_f ...)} ...),
(e_n ...) = ($\xi[\![e_d]\!]$ {(x_i ... x_d ...) := ((**guard** x_i c_{ir} x_s s_u) ... x_f ...)} ...)

$\xi[\![$(**compound-unit** s (**import** [x_i c_i] ...) (**export** [x_e c_e] ...) e_1 e_2)$]\!]$ = (**compound-unit** s (**import** [x_i $\xi[\![c_i]\!]$] ...) (**export** [x_e $\xi[\![c_e]\!]$] ...) $\xi[\![e_1]\!]$ $\xi[\![e_2]\!]$)

$\xi[\![$(**invoke-unit** e)$]\!]$ = (**invoke-unit** $\xi[\![e]\!]$)

$\xi[\![(e_1\ e_2)]\!]$ = ($\xi[\![e_1]\!]$ $\xi[\![e_2]\!]$)

$\xi[\![(\lambda\ (x)\ e)]\!]$ = (λ (x) $\xi[\![e]\!]$)

$\xi[\![(op\ e_1\ e_2)]\!]$ = (op $\xi[\![e_1]\!]$ $\xi[\![e_2]\!]$)

$\xi[\![$(**if** e_1 e_2 e_3)$]\!]$ = (**if** $\xi[\![e_1]\!]$ $\xi[\![e_2]\!]$ $\xi[\![e_3]\!]$)

$\xi[\![$(**begin** e_1 e_2)$]\!]$ = (**begin** $\xi[\![e_1]\!]$ $\xi[\![e_2]\!]$)

$\xi[\![x]\!]$ = x

$\xi[\![n]\!]$ = n

$\xi[\![b]\!]$ = b

$\xi[\![s]\!]$ = s

$\xi[\![$(-> c_1 c_2)$]\!]$ = (-> $\xi[\![c_1]\!]$ $\xi[\![c_2]\!]$)

$\xi[\![$(**unit/c** (**import** [x_i c_i] ...) (**export** [x_e c_e] ...))$]\!]$ = (**unit/c** (**import** [x_i $\xi[\![c_i]\!]$] ...) (**export** [x_e $\xi[\![c_e]\!]$] ...))

$\Sigma[\![x_i, ([x_{i1}\ c_{i1}] \ldots [x_i\ c_i]\ [x_{i2}\ c_{i2}] \ldots), e_p, e_n]\!]$ = (**guard** x_i $\Sigma[\![c_i, ([x_{i1}\ c_{i1}] \ldots [x_i\ c_i]\ [x_{i2}\ c_{i2}] \ldots), e_p, e_n]\!]$ e_p e_n)

$\Sigma[\![x, ([x_i\ c_i] \ldots), e_p, e_n]\!]$ = x

$\Sigma[\![(\lambda\ (x)\ e), ([x_{i1}\ c_{i1}] \ldots [x\ c]\ [x_{i1}\ c_{i1}] \ldots), e_p, e_n]\!]$ = (λ (x) e)

$\Sigma[\![(\lambda\ (x)\ e), ([x_i\ c_i] \ldots), e_p, e_n]\!]$ = (λ (x) $\Sigma[\![e, ([x_i\ c_i] \ldots), e_p, e_n]\!]$)

$\Sigma[\![(e_1\ e_2), ([x_i\ c_i] \ldots), e_p, e_n]\!]$ = ($\Sigma[\![e_1, ([x_i\ c_i] \ldots), e_p, e_n]\!]$ $\Sigma[\![e_2, ([x_i\ c_i] \ldots), e_p, e_n]\!]$)

$\Sigma[\![(op\ e_1\ e_2), ([x_i\ c_i] \ldots), e_p, e_n]\!]$ = (op $\Sigma[\![e_1, ([x_i\ c_i] \ldots), e_p, e_n]\!]$ $\Sigma[\![e_2, ([x_i\ c_i] \ldots), e_p, e_n]\!]$)

$\Sigma[\![$(**if** e_1 e_2 e_3), ([x_i c_i] ...), e_p, $e_n$$]\!]$ = (**if** $\Sigma[\![e_1, ([x_i\ c_i] \ldots), e_p, e_n]\!]$ $\Sigma[\![e_2, ([x_i\ c_i] \ldots), e_p, e_n]\!]$ $\Sigma[\![e_3, ([x_i\ c_i] \ldots), e_p, e_n]\!]$)

$\Sigma[\![$(**begin** e_1 e_2), ([x_i c_i] ...), e_p, $e_n$$]\!]$ = (**begin** $\Sigma[\![e_1, ([x_i\ c_i] \ldots), e_p, e_n]\!]$ $\Sigma[\![e_2, ([x_i\ c_i] \ldots), e_p, e_n]\!]$)

$\Sigma[\![$(**unit** s_u (**import** [x_{ui} c_{ui}] ...) (**export** [x_{ue} c_{ue}] ...) (**define** x_d e_d) ... e_b), ([x_i c_i] ...), e_p, $e_n$$]\!]$ = (**unit** s_u (**import** [x_{ui} c_{ui}] ...) (**export** [x_{ue} c_{ue}] ...)
(**define** x_d $\Sigma[\![e_d, ([x_{fi}\ c_{fi}] \ldots), e_p, e_n]\!]$) ... $\Sigma[\![e_b, ([x_{fi}\ c_{fi}] \ldots), e_p, e_n]\!]$)
where ([x_{fi} c_{fi}] ...) = filter-out$[\![(x_{ui} \ldots x_{ue} \ldots), ([x_i\ c_i] \ldots)]\!]$

$\Sigma[\![$(**compound-unit** s_u (**import** [x_{ui} c_{ui}] ...) (**export** [x_{ue} c_{ue}] ...) e_1 e_2), ([x_i c_i] ...), e_p, $e_n$$]\!]$ = (**compound-unit** s_u (**import** [x_{ui} c_{ui}] ...) (**export** [x_{ue} c_{ue}] ...)
$\Sigma[\![e_1, ([x_{fi}\ c_{fi}] \ldots), e_p, e_n]\!]$ $\Sigma[\![e_2, ([x_{fi}\ c_{fi}] \ldots), e_p, e_n]\!]$)
where ([x_{fi} c_{fi}] ...) = filter-out$[\![(x_{ui} \ldots x_{ue} \ldots), ([x_i\ c_i] \ldots)]\!]$

$\Sigma[\![$(**invoke-unit** e), ([x_i c_i] ...), e_p, $e_n$$]\!]$ = (**invoke-unit** $\Sigma[\![e, ([x_i\ c_i] \ldots), e_p, e_n]\!]$)

$\Sigma[\![$(**guard** e c e_1 e_2), ([x_i c_i] ...), e_p, $e_n$$]\!]$ = (**guard** $\Sigma[\![e, ([x_i\ c_i] \ldots), e_1, e_2]\!]$ $\Sigma[\![c, ([x_i\ c_i] \ldots)]\!]$ $\Sigma[\![e_p, ([x_i\ c_i] \ldots)]\!]$ $\Sigma[\![e_n, ([x_i\ c_i] \ldots)]\!]$)

$\Sigma[\![$(**error** s_e), ([x_i c_i] ...), e_p, $e_n$$]\!]$ = (**error** s_e)

$\Sigma[\![n, ([x_i\ c_i] \ldots), e_p, e_n]\!]$ = n

$\Sigma[\![b, ([x_i\ c_i] \ldots), e_p, e_n]\!]$ = b

$\Sigma[\![s, ([x_i\ c_i] \ldots), e_p, e_n]\!]$ = s

$\Sigma[\![$(**unit/c** (**import** [x_{ui} c_{ui}] ...) (**export** [x_{ue} c_{ue}] ...)), ([x_i c_i] ...), e_p, $e_n$$]\!]$ = (**unit/c** (**import** [x_{ui} $\Sigma[\![c_{ui}, ([x_i\ c_i] \ldots), e_p, e_n]\!]$] ...)
(**export** [x_{ue} $\Sigma[\![c_{ue}, ([x_i\ c_i] \ldots), e_p, e_n]\!]$] ...))

$\Sigma[\![$(-> c_1 c_2), ([x_i c_i] ...), e_p, $e_n$$]\!]$ = (-> $\Sigma[\![c_1, ([x_i\ c_i] \ldots), e_p, e_n]\!]$ $\Sigma[\![c_2, ([x_i\ c_i] \ldots), e_p, e_n]\!]$)

$\rho c[\![x, c_e, ([x_{o1}\ x_{n1}] \ldots [x\ x_n]\ [x_{o2}\ x_{n2}] \ldots), e_p, e_n]\!]$ = (**define** x (**guard** x_n c_e e_p e_n))

Figure 9. Contract compilation

E ::= **....** | (**guard** E c e e) | (**guard** v c E e) | (**guard** v c v E)
CU ::= (**compound-unit** s (**import** [x c] ...) (**export** [x c] ...) E e) | (**compound-unit** s (**import** [x c] ...) (**export** [x c] ...) v E)

P[(**invoke-unit** u)] $\longmapsto$ add-defs⟦P[e_b{(x_d ...) := (x_n ...)}], ((**define** x_n e_d{(x_d ...) := (x_n ...)}) ...)⟧ [Invoke]
where u = (**unit** s (**import**) (**export** [x_e c_e] ...) (**define** x_d e_d) ... e_b), (x_n...) fresh

(**compound-unit** s (**import** [x_i c_i] ...) (**export** [x_e c_e] ...) [Compound]
(**unit** s_1 (**import** [x_{i1} c_{i1}] ...) (**export** [x_{e1} c_{e1}] ...) (**define** x_{s1} "unknown") (**define** x_{d1} e_{d1}) ... e_1)
(**unit** s_2 (**import** [x_{i2} c_{i2}] ...) (**export** [x_{e2} c_{e2}] ...) (**define** x_{s2} "unknown") (**define** x_{d2} e_{d2}) ... e_2))
⟶ (**unit** s (**import** [x_i c_i] ...) (**export** [x_e c_e] ...) (**define** x_s "unknown") (**define** x_{s1n} s) (**define** x_{s2n} s) d_1 ... d_2 ...
ρc⟦x_e, c_{en}, ([x_{d1} x_m] ... [x_{d2} x_n] ...), s, x_s⟧ ... (**begin** e_{1n} e_{2n}))
where (c_{in} ...) = (Σ⟦c_i, ([x_i c_i] ...), x_s, s⟧ {(x_e ...) := (x_{en} ...)} ...),
(c_{en} ...) = (Σ⟦c_e, ([x_i c_i] ...), x_s, s⟧ {(x_e ...) := (x_{en} ...)} ...),
(e_{d1n} ...) = (e_{d1}{(x_{s1} x_{d1} ...) := (x_{s1n} x_m ...)}{(x_{d2} ...) := (x_n ...)}{(x_i ...) := ((**guard** x_i c_{in} x_s s) ...)} ...),
e_{1n} = e_1{(x_{s1} x_{d1} ...) := (x_{s1n} x_m ...)}{(x_{d2} ...) := (x_n ...)}{(x_i ...) := ((**guard** x_i c_{in} x_s s) ...)},
(e_{d2n} ...) = (e_{d2}{(x_{s2} x_{d2} ...) := (x_{s2n} x_n ...)}{(x_{d1} ...) := (x_m ...)}{(x_i ...) := ((**guard** x_i c_{in} x_s s) ...)} ...),
e_{2n} = e_2{(x_{s2} x_{d2} ...) := (x_{s2n} x_n ...)}{(x_{d1} ...) := (x_m ...)}{(x_i ...) := ((**guard** x_i c_{in} x_s s) ...)},
(d_1 ...) = ψ⟦((**define** x_m e_{d1n}) ...), (x_e ...), (x_{en} ...)⟧, (d_2 ...) = ψ⟦((**define** x_n e_{d2n}) ...), (x_e ...), (x_{en} ...)⟧,
(x_i ... x_{e1} ... x_{e2} ...) distinct, {x_{i1} ...} ⊆ {x_i ... x_{e2} ...}, {x_{i2} ...} ⊆ {x_i ... x_{e1} ...}, {x_e ...} ⊆ {x_{e1} ... x_{e2} ...},
x_s, x_{s1n}, x_{s2n}, (x_m...), (x_n...), (x_{en}...) fresh

(**guard** (λ (x) e) (-> c_1 c_2) s_p s_n) ⟶ (λ (x) (**guard** ((λ (x) e) (**guard** x c_1 s_n s_p)) c_2 s_p s_n)) [HO-λ]

(**guard** nlv (-> c_1 c_2) s_p s_n) ⟶ (**error** s_p) [HO-Notλ]

(**guard** (**unit** s (**import** [x_{ui} c_{ui}] ...) (**export** [x_{ue} c_{ue}] ...) (**define** x_d e_d) ... e_b) [UC-Unit]
(**unit/c** (**import** [x_i c_i] ...) (**export** [x_e c_e] ...)) s_p s_n)
⟶ (**unit** s (**import** [x_{ni} c_{ni}] ...) (**export** [x_{ne} c_{ne}] ...) (**define** x_{new} e_{dn}) ... ρc⟦x_e, c_e, ([x_d x_{new}] ...), s_p, s_n⟧ ... e_n)
where (e_{dn} ...) = (e_d{(x_d ...) := (x_{new} ...)}{(x_{ui} ...) := ((**guard** x_{ui} c_{ui} s_p s_n) ...)} ...),
e_n = e_b{(x_d ...) := (x_{new} ...)}{(x_{ui} ...) := ((**guard** x_{ui} c_{ui} s_p s_n) ...)},
([x_{ni} c_{ni}] ...) = expand-imports⟦([x_{ui} c_{ui}] ...), (x_i ...)⟧,
([x_{ne} c_{ne}] ...) = contract-exports⟦([x_{ue} c_{ue}] ...), (x_e ...), ()⟧, {x_{ui} ...} ⊆ {x_i ...}, {x_e ...} ⊆ {x_{ue} ...},
(x_{new}...) fresh

(**guard** (**unit** s (**import** [x_{ui} c_{ui}] ...) (**export** [x_{ue} c_{ue}] ...) d ... e_b) (**unit/c** (**import** [x_i c_i] ...) (**export** [x_e c_e] ...)) s_p s_n) [UC-Mismatch]
⟶ (**error** s_p)
where otherwise

(**guard** nuv (**unit/c** (**import** [x_i c_i] ...) (**export** [x_e c_e] ...)) s_p s_n) ⟶ (**error** s_p) [UC-NotUnit]

(**guard** v e s_p s_n) ⟶ (**if** (e v) v (**error** s_p)) [FO-Guard]

P[(**error** s)] $\longmapsto$ (**error** s) [Error]

Figure 10. New and revised reduction rules

e ::= **....** | (**guard** e c e e) | (**error** s)

Figure 12. Internal syntax for guard expressions

know which server unit provides the services. Hence, the compiler doesn't know the name for the positive blame positions in a function contract.

To overcome these knowledge gaps, our compiler for unit contracts uses a different compilation strategy for exports *and* delays additional aspects of contract checking to the linking step. Specifically, it adds guards to the exported functions as well as imported ones. Furthermore, it uses a protocol for introducing and changing placeholders for the positive blame positions.

Figure 12 shows the extensions to the expression syntax with guard expressions and contract errors. In our model, these new syntactic forms are invisible to the programmer; an implementation can re-use existing syntax (e.g., **if** expressions and **error** functions). The **guard** form consists of an expression, a contract, and two strings (blame labels) for the positive and negative blame positions of contracts, respectively.

4.3 Contract Compilation

Figure 9 specifies the compiler of our unit contract model. The metafunction ξ is applied to all top-level expressions in the program. It is mostly a straightforward homomorphism on expressions and contracts except for the **unit** case. For units, ξ renames all internal definitions and their uses, creates (using ρc) new export definitions that guard the internal implementation, and replaces all uses of import variables with guarded versions of the same. Finally, the function adds a definition for x_s to unit bodies; this new definition acts as a placeholder for the blame label of the yet-to-be-determined compound unit that links this unit into the full program.

$\psi[\![(d_1 \ldots (\textbf{define } x_d\ e_d)\ d_2 \ldots), (x_{o1} \ldots_1 x_d\ x_{o2} \ldots), (x_{n1} \ldots_1 x_{new}\ x_{n2} \ldots)]\!] = \psi[\![(d_1 \ldots (\textbf{define } x_{new}\ e_{new})\ d_2 \ldots), (x_{o1} \ldots x_d\ x_{o2} \ldots), (x_{n1} \ldots x_{new}\ x_{n2} \ldots)]\!]$
where $d_1 = (\textbf{define } x_1\ e_1)$,
$e_{new} = e_d\{(x_{o1} \ldots x_d\ x_{o2} \ldots) := (x_{n1} \ldots x_{new}\ x_{n2} \ldots)\}$,
$\{x_1 \ldots\} \cap \{x_{o1} \ldots x_d\ x_{o2} \ldots\} = \varnothing$

$\psi[\![((\textbf{define } x\ e) \ldots), (x_1 \ldots), (x_2 \ldots)]\!] = ((\textbf{define } x\ e) \ldots)$

$any_1\{() := ()\} = any_1$

$any_1\{(x_1\ x_2 \ldots) := (any_2\ any_3 \ldots)\} = any_1\{x_1 := any_2\}\{(x_2 \ldots) := (any_3 \ldots)\}$

$x_a\{x_a := e_a\} = e_a$

$x\{x_a := e_a\} = x$

$(e_1\ e_2)\{x_a := e_a\} = (e_1\{x_a := e_a\}\ e_2\{x_a := e_a\})$

$(\textbf{if } e_1\ e_2\ e_3)\{x_a := e_a\} = (\textbf{if } e_1\{x_a := e_a\}\ e_2\{x_a := e_a\}\ e_3\{x_a := e_a\})$

$(op\ e_1\ e_2)\{x_a := e_a\} = (op\ e_1\{x_a := e_a\}\ e_2\{x_a := e_a\})$

$(\textbf{invoke-unit } e)\{x_a := e_a\} = (\textbf{invoke-unit } e\{x_a := e_a\})$

$(\textbf{guard } e\ c\ e_s\ e_p)\{x_a := e_a\} = (\textbf{guard } e\{x_a := e_a\}\ c\{x_a := e_a\}\ e_s\{x_a := e_a\}\ e_p\{x_a := e_a\})$

$(\textbf{error } s)\{x_a := e_a\} = (\textbf{error } s)$

$(\textbf{begin } e_1\ e_2)\{x_a := e_a\} = (\textbf{begin } e_1\{x_a := e_a\}\ e_2\{x_a := e_a\})$

$(\textbf{unit } s\ (\textbf{import } [x_{i1}\ c_{i1}] \ldots [x_a\ c_a]\ [x_{i2}\ c_{i2}] \ldots)\ (\textbf{export } [x_e\ c_e] \ldots)\ d \ldots e)\{x_a := e_a\}$
$= (\textbf{unit } s\ (\textbf{import } [x_{i1}\ c_{i1}] \ldots [x_a\ c_a]\ [x_{i2}\ c_{i2}] \ldots)\ (\textbf{export } x_e \ldots)\ d \ldots e)$

$(\textbf{unit } s\ (\textbf{import } [x_i\ c_i] \ldots)\ (\textbf{export } [x_e\ c_e] \ldots)\ d_1 \ldots (\textbf{define } x_a\ e_d)\ d_2 \ldots e_b)\{x_a := e_a\}$
$= (\textbf{unit } s\ (\textbf{import } [x_i\ c_i] \ldots)\ (\textbf{export } [x_e\ c_e] \ldots)\ d_1 \ldots (\textbf{define } x_a\ e_d)\ d_2 \ldots e_b)$

$(\textbf{unit } s\ (\textbf{import } [x_i\ c_i] \ldots)\ (\textbf{export } [x_e\ c_e] \ldots)\ (\textbf{define } x_d\ e_d) \ldots e_b)\{x_a := e_a\}$
$= (\textbf{unit } s\ (\textbf{import } [x_i\ c_i\{x_a := e_a\}] \ldots)\ (\textbf{export } [x_e\ c_e\{x_a := e_a\}] \ldots)\ (\textbf{define } x_d\ e_d\{x_a := e_a\}) \ldots e_b\{x_a := e_a\})$

$(\textbf{compound-unit } s\ (\textbf{import } [x_{i1}\ c_{i1}] \ldots [x_a\ c_a]\ [x_{i2}\ c_{i2}] \ldots)\ (\textbf{export } [x_e\ c_e] \ldots)\ e_1\ e_2)\{x_a := e_a\}$
$= (\textbf{compound-unit } s\ (\textbf{import } [x_{i1}\ c_{i1}] \ldots [x_a\ c_a]\ [x_{i2}\ c_{i2}] \ldots)\ (\textbf{export } [x_e\ c_e] \ldots)\ e_1\ e_2)$

$(\textbf{compound-unit } s\ (\textbf{import } [x_i\ c_i] \ldots)\ (\textbf{export } [x_e\ c_e] \ldots)\ e_1\ e_2)\{x_a := e_a\}$
$= (\textbf{compound-unit } s\ (\textbf{import } [x_i\ c_i\{x_a := e_a\}] \ldots)\ (\textbf{export } [x_e\ c_e\{x_a := e_a\}] \ldots)\ e_1\{x_a := e_a\}\ e_2\{x_a := e_a\})$

$(\lambda\ (x_a)\ e_b)\{x_a := e_a\} = (\lambda\ (x_a)\ e_b)$

$(\lambda\ (x_f)\ e_b)\{x_a := e_a\} = (\lambda\ (x_f)\ e_b\{x_a := e_a\})$

$b\{x_a := e_a\} = b$

$n\{x_a := e_a\} = n$

$s\{x_a := e_a\} = s$

$(\textbf{unit/c } (\textbf{import } [x_i\ c_i] \ldots)\ (\textbf{export } [x_e\ c_e] \ldots))\{x_a := e_a\} = (\textbf{unit/c } (\textbf{import } [x_i\ c_i\{x_a := e_a\}] \ldots)\ (\textbf{export } [x_e\ c_e\{x_a := e_a\}] \ldots))$

$(\text{-> } c_1\ c_2)\{x_a := e_a\} = (\text{-> } c_1\{x_a := e_a\}\ c_2\{x_a := e_a\})$

Figure 11. Substitution and Definition Renaming

The trickiest part of this conversion is handling the use of imported identifiers in contracts. Since uses of variables in contracts are considered to be internal to the unit, we do not check the contracts for exported variables, but we do need to check those on imports. Contracts on imports may use other imports, however, and to deal with such uses, our compiler unrolls the contracts for imports and exports completely.

The metafunction Σ appropriately unrolls these contracts. As defined, this unrolling may not terminate if two or more imports use each other in their respective contracts:

```
(unit "loop" (import [x? (⟶ y? any/c)]
                     [y? (⟶ x? any/c)])
             (export)
             (x? 3))
```

An implementation of Σ can detect such cyclic dependencies in import contracts for each **unit** or **compound-unit** expression and refuse to compile the program if such a cycle is detected. We skip this complication for Σ because it adds little to this paper.

4.4 Dynamic Contract Checking

The rest of the contract checking process is performed at run-time for both higher-order functions and first-class units. The former follows Findler and Felleisen [2002]. The latter demands changes to the reduction rules for units.

Figure 10 specifies the revised reduction relations, along with the revised definitions of evaluation contexts. These new relations rely on a number of auxiliary functions, including substitution and renaming functions (ψ). The latter are defined in figure 11.

Two aspects of this process deserve special attention: unit linking and unit guarding. The rest of this section is dedicated to these two aspects of run-time checking.

4.4.1 Linking Units

As noted in section 4.2, compounding units introduces new parties to contracts, especially for contracts that don't match. To assign blame to the proper compound unit, our run-time system must synthesize guard expressions during a linking step. A revised [Compound] reduction relation achieves this as follows:

- create a new placeholder blame label for future linking;
- replace the placeholder blame label from each unit with the blame label of the compound unit;
- rename the definitions from each unit and their uses, which includes imported uses in the other unit;
- guard all uses of the compound unit's imports;

- create guarded versions of the compound unit's exports; and
- sequence the body expressions.

The renaming step ensures that linking does not inadvertently add contract checks in a place where they are unwanted.

Consider this program:

```
(program
 (define ge/c
   (λ (m)
     (λ (n)
       (<= m n))))
 (define server@
  (unit "server"
    (import)
    (export [f (⟶ (ge/c 2) (ge/c 4))])
   (define f (λ (n) (* n n)))
   "default"))
 (define client@
  (unit "client"
    (import [f (⟶ (ge/c 2) (ge/c 4))])
    (export)
   (f 2)))
 (define link@
  (compound-unit "link"
    (import)
    (export [f (⟶ (ge/c 3) (ge/c 9))]) server@ client@))
 ...)
```

Here, for example, we do not want the use of f in *client@* to be checked with the export contract from *link@*.

The compiler translates the program as follows:

```
(program
 (define ge/c
   (λ (m)
     (λ (n)
       (<= m n))))
 (define server@
   (unit "server"
     (import)
     (export (f (⟶ (ge/c 2) (ge/c 4))))
    [(define x "unknown")]
     (define [f1] (λ (n) (* n n)))
    [(define f [f1] (guard f1 (⟶ (ge/c 2) (ge/c 4)) "server" x))]
    "default"))
 (define client@
   (unit "client"
     (import (f (⟶ (ge/c 2) (ge/c 4))))
     (export)
    [(define x "unknown")]
    ([(guard] f [(⟶ (ge/c 2) (ge/c 4)) x "client")] 2)))
 (define link@
   (compound-unit "link"
     (import)
     (export (f (⟶ (ge/c 3) (ge/c 9)))) server@ client@))
 ...)
```

After substituting the two units into the **compound-unit** expression, the [Compound] relation applies and linking takes place:

```
(program
 (define ge/c (λ (m) (λ (n) (<= m n))))
 (define server@
  (unit "server" (import) (export [f (⟶ (ge/c 2) (ge/c 4))])
   (define x "unknown") (define f1 (λ (n) (* n n)))
   (define f (guard f1 (⟶ (ge/c 2) (ge/c 4))) "server" x)
   "default"))
 (define client@
  (unit "client" (import [f (⟶ (ge/c 2) (ge/c 4))]) (export)
   (define x "unknown")
   ((guard f (⟶ (ge/c 2) (ge/c 4)) x "client") 2)))
 (define link@
  (unit "link" (import) (export [f (⟶ (ge/c 3) (ge/c 9))])
   [(define x "unknown")]
   (define [x1] ["link"]) (define [x2] ["link"])
   (define [f2] (λ (n) (* n n)))
   (define [f1]
     (guard [f2] (⟶ (ge/c 2) (ge/c 4)) "server" [x1]))
   [(define f (guard f1 (⟶ (ge/c 3) (ge/c 9)) "link" x))]
   (begin "default"
          ((guard [f1] (⟶ (ge/c 2) (ge/c 4)) [x2] "client")
           2))))
 ...)
```

The use of f from the body of *client@* has been appropriately altered to refer to the definition *f1*, which corresponds to the guarded export of *server@*, and not f, which is the guarded export of *link@*.

4.4.2 Guarding Values with unit/c

A **unit/c** contract requires a few first-order run-time checks. Specifically, the [UC-Unit] reduction ensures that the given value is a unit and that it has the required exports and imports. For other cases, the remaining UC rules pinpoint and explain the violation.

The result of a [UC-Unit] reduction is a unit that contains the additional contract checks on its imports and exports. During its construction, the reduction renames the definitions inside the unit. It also replaces all uses of imports with appropriate guard expressions. Finally, it creates new definitions to guard the exports listed by the **unit/c** contract. The new unit is limited to those exports and to imports listed in the **unit/c** expression. Those imports that are listed in the latter, but not imported by the original unit are added to the import list with the contract that always succeeds.

4.5 Basic Properties of the Model

An operational model of contract checking should validate a basic property of contracts, namely, that they do not add unwanted behavior. To state this theorem, we need to define two simple operations. First, let $\|p\|$ be the program transformation that removes all blame strings and contracts from a given program. That means that if p is a program in the syntax from section 4.1, $\|p\|$ is a program in the syntax from section 2.2.

Second, let $\texttt{eval}(p)$ be the following function:

$$\texttt{eval}(p) = \begin{cases} n & \text{if } p \longmapsto^* (\textbf{program } d \ldots n) \\ b & \text{if } p \longmapsto^* (\textbf{program } d \ldots b) \\ s & \text{if } p \longmapsto^* (\textbf{program } d \ldots s) \\ \textbf{closure} & \text{if } p \longmapsto^* (\textbf{program } d \ldots (\lambda\ (x)\ e)) \\ \textbf{unit} & \text{if } p \longmapsto^* (\textbf{program } d \ldots u) \end{cases}$$

Given these definitions, the basic soundness theorem says that if a contracted program evaluates to a basic value, then its uncontracted counter-part evaluates to the same basic values. Contracts do not add behavior.

THEOREM 4.1. *For all closed programs p, if* $\texttt{eval}(\xi[\![p]\!]) = v$, *then* $\texttt{eval}(\|p\|) = v$.

The proof for theorem 4.1 proceeds by examining the reduction sequence for $\xi[\![p]\!]$ and relating it to the reduction sequence for $\|p\|$, showing that the extra computation steps added by guard expressions have no effect on the sequence as a whole, since the contracts are not broken.

A second theorem proves that a program either evaluates to a value, diverges, or runs into a well-defined error situation. Note that this "type soundness" theorem applies to both typed as well as untyped languages. The difference between the former and the latter is all about the possible set of errors; the untyped language may raise errors for cases where the typed language doesn't allow execution. Thus, if we use the Flatt and Felleisen [1998] type system on our programs, we can state a conventional type soundness theorem, claiming that the program either produces a value, runs into an infinite loop, or raises a run-time error.

THEOREM 4.2. *For all well-typed programs p,* $\texttt{eval}(\xi[\![p]\!]) = v$, $\xi[\![p]\!] \Uparrow$, *or* $\xi[\![p]\!] \longmapsto^* (\textbf{error}\ s)$.

The proof for theorem 4.2 uses the proof technique of Wright and Felleisen [1994]. Specifically, using the Flatt-Felleisen type system (extended to contracts), a progress lemma ensures that all typed programs can make progress or raise a contract error. Similarly, a preservation lemma validates that if a program reduces to another and the first one is well-typed, then so is the second one.

5. Implementation

While a model is a mathematically precise explanation of a language extension, it often fails to bring across how to add such extensions to an existing implementation. For example, Flatt and Felleisen [1998] explains units with a reduction semantics that copies the bodies of units at will, which would impose a huge overhead if translated naively into a compilation strategy, Instead, the unit compiler represents all units as grey boxes and implements unit linking and invocation via operations on these grey boxes.

In this section, we sketch how to implement our contract model in the context of a unit system such as the one available in PLT Scheme. Other dynamic module systems may need a different treatment, but we expect that the sketch here provides some insight on how to adapt the model to other contexts. The section starts with a brief, cursory explanation of how units are implemented, especially how linking is implemented without copying code. In the second subsection, we explain how to add contracts to such a unit implementation. Finally, in the last section, we discuss how to implement **unit/c** as a projection [Findler and Blume 2006].

5.1 Implementing Units

At a high-level of abstraction, the PLT Scheme compiles all units into thunks—parameter-less procedures—that produce two values:

- a mapping M from exports to reference cells; and
- a function f that consumes a mapping from imports to reference cells and runs the unit's body.

The compiler replaces uses of the imports in the unit body with appropriate accesses into the import mapping. It also adds assignment statements below each definition of an export that transfers the value into the expected reference cell of M.

Given a thunk-based representation of units, a unit invocation proceeds as follows. An application of the thunk yields two values, including a function that consumes an import mapping and runs the body of the unit. Since units are only invoked if their import list is empty, this function is applied to the empty import mapping and then executes the definitions and expressions of the given unit.

Similarly, the purpose of linking two (or more) units is to construct a new thunk from the given thunks. This new thunk returns M and f, which are constructed in a three-step process:

1. It applies the unit thunks for the constituent units, resulting in two export mappings (M_1, M_2) and two body functions (f_1, f_2).
2. Next a new export mapping M is constructed from the contents of M_1 and M_2.
3. In parallel, a new body function f is constructed. It
   (a) receives the import mapping I for the compound unit,
   (b) constructs an import mapping I_1 from I and M_2,
   (c) applies I_1 to f_1,
   (d) constructs an import mapping I_2 from I and M_1, and
   (e) applies I_2 to f_2.

Note that the process deals with the bodies of the constituent units as black boxes, copying not the code in these units but only pointers to the code.

5.2 Implementing Unit Contracts

Although our model of unit contracts assumes that linking may access the unit bodies, doing so for an implementation would radically alter the way PLT Scheme deals with units. Since we wish to fall back on a copying compiler, we add contracts only during linking, not to atomic units. Adding the appropriate contract guards in this manner poses challenges, however, because the values being guarded are set in the reference cells only after the unit bodies have been executed.

Our compiler therefore changes how export and import mappings work. Specifically, the body function for a unit fixes the import mappings such that when the imports are accessed, a value is received that has been appropriately wrapped with contracts. This wrapping must be delayed at least until the first time the import is accessed, to ensure it is not prematurely requested.

To allow for this delayed wrapping of imports, we alter the translation of units so that exports are mappings from names to thunks. The export thunks of atomic units are closed over a reference cell and simply return the value currently stored. Imports in a unit body are just translated to an access into the import mapping and application of the resulting thunk.

When the function f corresponding to a compound unit constructs the import mappings I_1 and I_2, it creates new thunks for names coming from the export mapping of the other constituent unit (M_2 and M_1, respectively). This thunk evaluates the original thunk from that export mapping, resulting in the exported value, and then wraps that value with the appropriate contracts. The contracts are those listed by both the exporting and importing units.

5.3 Implementing unit/c

Following Findler and Blume [2006], **unit/c** denotes a pair of projections. For function contracts, a projection consumes positive and negative blame labels and returns a function, which is like the given one, except that it uses the projections to enforce contracts.

The projection for a **unit/c** contract returns a function that

1. checks that its input is a unit;
2. checks that the unit imports a subset and exports a superset of those names listed in the **unit/c** form;
3. applies the unit thunk to get M and f;
4. and constructs a unit thunk that constructs a new export mapping M_n and function f_n and returns both.

The new export mapping M_n contains entries for those names listed in the **unit/c** form and maps them to new thunks. Each new thunk applies the thunk from M to get the exported value, applies the positive and negative blame labels to the projection corresponding to that name's contract, and then returns the application of the projection result to the exported value.

The new function f_n takes its import mapping argument I and performs almost the same alteration as for M_n to create I_n. The only difference is that the positive and negative blame are swapped when applied to the projections. When this step is finished, it applies f to I_n.

6. Related Work

Conceptually, the notion of contracts is due to Parnas [1972], who introduced it together with the notion of modules. Pragmatically, Meyer [1992] is responsible for the terminology of "design by contracts" and the popularization of the concept in the object-oriented community. A number of years ago, Findler and Felleisen [2002] began an exploration of contracts in the world of higher-order functional programming languages; numerous researchers expanded this exploration to a range of scenarios (lazy functions, lazy data types, lazy constructors, static analysis of module contracts, and theorem proving).

Technically our work follows two key pieces of research on contracts in higher-order functional programming languages. Findler and Felleisen [2002] introduced the idea of adding contracts to functional languages. Findler and Blume [2006] fleshed out the theoretical foundations of contracts by treating them as pairs of projections, which is currently the basis of implemented contract systems for higher-order languages.

All prior research on contracts assumes static boundaries between the parties of a contract and boundaries that are known at compile time. Our paper is the first to relax these assumptions and to provide a framework for checking contracts in a world of higher-order modules.

7. Conclusion

First-class modules are a useful feature in modern programming languages. From functors in ML, units in PLT Scheme, and components in Fortress to trait systems in Smalltalk and Scala, first-class components allow the programmer to code separate software components that can be linked together into various arrangements, facilitating code reuse. Until now, however, these features could not coexist with software contracts, which allow programmers to protect their components against abuse.

In this paper we present a design for adding contracts to a language with first-class modules. We also sketch how languages with first-class modules can implement these contracts, describe how to use the new features to gradually add contracts to a unit-based program, and present a contract combinator for units. Our contract system is part of the latest release of PLT Scheme, available at `http://www.plt-scheme.org/`.[6]

Acknowledgments

Thanks to Matthew Flatt and Scott Owens for discussions concerning the PLT Scheme unit system; to Ryan Culpepper, Carl Eastlund, and Sam Tobin-Hochstadt for discussions about various aspects of the theoretical model and the implementation of unit contracts; and to Christos Dimoulas for comments on a draft of the paper.

[6] Warning: Units in this paper use structural signature matching, while units in PLT Scheme rely on nominal matching, which affects many details at the syntax level, the contract model, and the implementation.

References

Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, 2008.

Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. In *IEEE Software*, pages 38–45, June 1999.

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

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. A functional I/O system, or fun for freshman kids. In *ACM SIGPLAN International Conference on Functional Programming*, page to appear, September 2009b.

Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In *Eighth International Symposium on Functional and Logic Programming*, volume 3945 of *LNCS*, pages 226–241. Springer, April 2006.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, October 2002.

Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, September 1998.

Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *LNCS*, pages 369–388. Springer, September 1997.

Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, June 1998.

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, January 1998.

Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems*, pages 270–289, November 2006.

Matthew Flatt et al. Reference: PLT Scheme. Reference Manual PLT-TR2009-reference-v4.1.4, PLT Scheme Inc., January 2009. `http://plt-scheme.org/techreports/`.

Paul T. Graunke, Shriram Krishnamurthi, Steve van der Hoeven, and Matthias Felleisen. Programming the Web with high-level programming languages. In *European Symposium on Programming*, volume 2028 of *LNCS*, pages 122–136. Springer, April 2001.

Xavier Leroy. Manifest types, modules, and separate compilation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122, January 1994.

Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10): 40–51, October 1992.

Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

Martin Odersky. *The Scala Language Specification*. Ecole Polytechnique Fédérale de Lausanne, 2009.

Scott Owens and Matthew Flatt. From structures and functors to modules and units. In *ACM SIGPLAN International Conference on Functional Programming*, pages 87–98, September 2006.

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

Gordon D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, pages 125–159, 1975.

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *European Conference on Object-Oriented Programming*, volume 2743 of *LNCS*, pages 248–274. Springer, July 2003.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, pages 38–94, 1994. First appeared as Technical Report TR160, Rice University, 1991.