# GRADUAL TYPING FOR FIRST-CLASS MODULES

Daniel Feltey

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

Department of Computer Science
Northeastern University
Boston, MA

May 2015

**NORTHEASTERN UNIVERSITY**
**GRADUATE SCHOOL OF COMPUTER SCIENCE**
**M.S. THESIS APPROVAL FORM**

*THESIS TITLE:* Gradual Typing for First-Class Modules

*AUTHOR:* Daniel Feltey

*M.S. Thesis Approved as an Elective towards the Master of Science Degree in Computer Science.*

| | |
|---|---|
| [illegible] | 7 May 2015 |
| *Thesis Advisor* | *Date* |
| [illegible] | 5/7/15 |
| *Thesis Reader* | *Date* |
| [illegible] | May 7, 2015 |
| *Thesis Reader* | *Date* |
| | |
| *Thesis Reader* | *Date* |

*GRADUATE SCHOOL APPROVAL:*

| | |
|---|---|
| [illegible] | 5/7/2015 |
| *Director, Graduate School* | *Date* |

*COPY RECEIVED IN GRADUATE SCHOOL OFFICE:*

| | |
|---|---|
| [illegible] | 5/7/2015 |
| *Recipient's Signature* | *Date* |

*Distribution: Once completed, this form should be scanned and attached to the front of the electronic Dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.*

# CONTENTS

# 1 INTRODUCTION

*If programmers sang hymns, some of the most popular would be hymns in praise of modular programming.* – D. L. Parnas

## 1.1 Gradual Typing and First-Class Components

Modularity is a prime concern in programming language design, and even dynamically-typed scripting languages such as Racket, Python, and Ruby support modular programming. In Parnas's words, module systems "allow one module to be written with little knowledge of the code used in an another module" and "allow modules to be reassembled and replaced without reassembly of the whole system" [10]. Languages like Racket [4] take full advantage of their dynamism to enable the latter with *units*—first-class modules that are available at run-time for flexible loading and linking. Their dynamic nature allows additional modules to be linked into the system while it is running without reassembly of the whole system. First-class components thus enable flexible uses such as dynamic plugin architectures. Furthermore, first-class components offer mutually-recursive linking, allowing even tightly coupled modules to be written separately. While the flexibility of Racket's units enable useful software development patterns, the lack of static type-checking means that the interfaces of these components are only loosely defined and enforced. This looseness hampers the ability to write reliable and independent modules.

Meanwhile, gradual type systems offer the ability to retrofit static type-based reasoning on top of existing dynamically-typed languages by allowing the programmer to selectively add type annotations to parts of their programs. The added annotations are statically checked, providing accurate documentation for the interfaces that the original developers wrote down in the comments or, in the worst case, only pictured in their minds. Therefore a gradual type system is a perfect match for first-class components, allowing programmers to retrofit stricter interfaces on existing components.

This brings me to my thesis:

*Gradual typing accommodates first-class modules.*

I demonstrate this through the design of a static type system for units and a corresponding implementation in Typed Racket. I show that programming with gradually typed first-class modules is practical by porting several unit-based programs from the Racket codebase to Typed Racket.

# 2 UNITS

## 2.1 Programming with Units

Large systems often necessitate the use of modules in order to reduce the complexity of individual components. When these components are tightly coupled, however, it becomes difficult to break up a program without introducing cyclic dependencies. Units mitigate this issue by explicitly allowing recursive linking, thus allowing large programs to be more readily broken down into logical components.

Consider the implementation of a typechecker for a language like Typed Racket [12]. Implementing a typechecker requires functions to typecheck each form in a language such as *if* expressions, *lambda* expressions, *let* expressions, and *applications*. Each typing rule should be implemented separately from all of the others, but due to the recursive nature of abstract syntax trees the process of typechecking these forms is highly interconnected. Using units allows typechecking for each form to be implemented independently of the others closely following the recursive nature of the problem. The typechecker for the full language is produced by linking together the units for each form.

```
(define-signature tc-expr^ (tc-expr))
(define-signature tc-if^ (tc/if))
(define-signature tc-lambda^ (tc/lambda))
(define-signature tc-let^ (tc/let))
(define-signature tc-app^ (tc/app))
```

Figure 1: Signatures in Typed Racket

Linking for Racket's units are mediated by *signatures*. Each signature provides a set of names that a unit will either agree to import or export. Figure 1 presents a subset of the signatures used in the actual implementation of Typed Racket. The *define-signature* form binds a signature name given a collection of names to be imported or exported. Since Racket links units in a *nominal* fashion, units that import a particular signature can only link with units that export that exact signature and vice versa. Signatures make programming with units manageable by providing static interfaces to support separate compilation and providing single identifiers to represent large collections of imports or exports. In structural unit systems, units must specify each imported and exported identifier which is impractical for units with large numbers of imports or exports. Many signatures used in the implementation of Racket contain long lists of identifiers. Using signatures allows simple reuse of these collections that is not subject to the error-prone lists of structural unit systems.

The definition of the *tc-expr@* unit below demonstrates the use of imports and exports. The *tc-expr@* unit exports the *tc-expr^* signature which requires the definition of *tc-expr*, the function that implements typechecking for expressions. Taking full advantage of the unit's imports, the *tc-expr* function pattern matches on its arguments and defers to the more specialized typechecking functions.

```
(define tc-expr@
  (unit (import tc-if^ tc-lambda^ tc-let^ tc-app^)
        (export tc-expr^)
        (define (tc-expr form)
          (syntax-parse form
            [(if test then else) (tc/if form)]
            [(#%plain-lambda formals . body)
              (tc/lambda form)]
            [(let ([name expr] ...) e) (tc/let form)]
            [(#%plain-app  . _) (tc/app form)]
            ...))))
```

The *tc-expr@* unit is not useful in isolation. It depends on units that implement its imported signatures. Before a unit's body can be evaluated the unit must satisfy its imports. The primary means of satisfying a unit's imports is to link it with other units that provide the necessary signatures. We can create a *typechecker@* unit for typechecking expressions by linking the *tc-expr@* unit with the other units necessary for typechecking.

```
(define typechecker@
  (compound-unit
    (import)
    (export TC-EXPR)
    (link (([TC-EXPR : tc-expr^])
           tc-expr@
           TC-IF TC-LAMBDA TC-LET TC-APP)
          (([TC-IF : tc-if^]) tc-if@ TC-EXPR)
          (([TC-LAMBDA : tc-lambda^]) tc-lambda@ TC-EXPR)
          (([TC-LET : tc-let^]) tc-let@ TC-EXPR)
          (([TC-APP : tc-app^])
           tc-app@
           TC-EXPR TC-LAMBDA TC-LET))))
```

The *compound-unit* form creates new units from existing units by linking them together. To produce a compound unit, the programmer provides import and export lists like an ordinary unit and a series of *linking clauses*. Each linking clause specifies the linking behavior of a particular unit value that participates in the linking. In each clause, the programmer writes down the exports of the unit, a reference to the unit value, and its imports. For example, the *tc-if@* unit exports the *tc-if^* signature which is bound to the name *TC-IF* in the body of the compound-unit form. It also imports the signature bound to the name *TC-EXPR* which represents the *tc-expr^* signature bound in the first linking clause. The *typechecker@* compound-unit recursively links together the units for each element of the typechecker. Despite the mutually recursive relationship between *tc-expr@* and the *tc-let@*, the linking behavior of units allows these components to be developed independently of one another.

A unit may provide definitions or values to its context via definitions or expressions in its body. These definitions or values can only be communicated to a unit's context when the unit is *invoked*. Furthermore, a unit can only be invoked when all of its imports have been satisfied, or when it imports no signatures, to ensure that the unit's body is well-defined.

Units are typically invoked using the *invoke-unit* form, which returns the result of evaluating the final expression in the unit's body. An invocation of a unit can also introduce definitions into the surrounding scope. For example, the *define-values/invoke-unit* form used below invokes a unit and introduces definitions for the unit's exports.

```
(define-values/invoke-unit typechecker@
                           (import)
                           (export tc-expr^))
(tc-expr #'((lambda ([x : Integer]) (+ x 5)) 12))
```

Since invoking units requires all imports to be satisfied, the programmer must usually provide units for all imports. Instead, since creating units can be a syntactic burden, many operations on units allow a unit's imports to be satisfied by bindings in its lexical context.

For example the *heap@* unit below is invoked without linking to a unit that exports the *compare^* signature. The *invoke-unit* form in this case uses the definition of *compare* in the current scope to satisfy the *heap@* unit's imports.

```
(define-signature compare^ (compare))
(define-signature heap^ (merge ...))
(define (compare n m) (< n m))
(define heap@
  (unit (import compare^)
        (export heap^)
        (define (merge h1 h2)
          (... (compare ...)))))
(invoke-unit heap@ (import compare^))
```

## 2.2 Tracking Dependencies

Invoking a unit in Racket may cause side effects such as printing to the screen or reading input from the user. The ability for units to contain side-effecting computation gives units expressive power, but it also adds a complication regarding how to manage these effects. If two effectful units are linked together using the *compound-unit* form, the first unit in the link clause executes its body first when the compound unit is invoked. If an effect in one unit depends on the other, then re-ordering the linking clauses in a compound unit may break the program.

Racket avoids this problem by allowing programmers to annotate units with *init-depend* clauses. These clauses specify which imports of a unit must be linked before the given unit. Failure to satisfy the dependency ordering results in an error at link-time.

As an example of the use of *init-depend* clauses, consider the following effectful units:

```
(define-signature data^ (data))
(define-signature results^ (results))
(define data@
  (unit (import)
        (export data^)
        (define data (read))))
(define results@
  (unit (import data^)
        (export results^)
        (init-depend data^)
        (define results
          (... data ...))))
```

the *data@* unit calls the *read* function to get input from the user and the *results@* unit uses the data to generate a results value. Since *results@* requires that the data is initialized before any processing can occur, it is annotated with an *init-depend* clause. The *init-depend* clause requires that the unit providing the *data^* signature must be linked in before *results@*. Trying to link the results unit before the data unit in a *compound-unit* expression will result in a runtime error.

## 2.3 Prior Work on First-Class Modules

While gradual typing for modular programming is novel, it builds on a long line of work on type systems for modules. In this section, I describe two lines of work that are particularly relevant: existing formalisms for units and ML modules.

### 2.3.1 *Units*

The first presentation of units by Flatt and Felleisen [3] had the goal of improving on module systems by supporting separate compilation, dynamic loading, and cyclic linking. The design I present in this thesis diverges from Flatt and Felleisen in order to support the pragmatics of the unit library as it exists in full-fledged Racket. In this section, I describe the key differences between the designs at a high-level. In subsequent chapters, I detail the differences in an informal design overview and with a formal model.

The Flatt and Felleisen model of units differs from Racket's units by using a structural linking mechanism as opposed to nominal linking. In Flatt's model, units import and export lists of individual identifiers rather than signatures. Flatt and Felleisen equipped their untyped model of units with a corresponding structural type system. Their type system forms the basis of the type system that I develop in this thesis.

Though Flatt and Felleisen's type system and this thesis both tackle notions of unit "dependencies", their purpose and implementation differ significantly. Dependency tracking in Flatt and Felleisen's model is used to track and prevent recursive type definitions through the linking of units. The initialization dependencies described in my model enforce the linking order specified by *init-depend* clauses. Furthermore, dependency information is tracked only with unit types, rather than with unit values, in the Flatt and Felleisen model. In Racket, the initialization dependencies are tracked in unit values.

Owens and Flatt [9] build on the Flatt-Felleisen model of units to include translucent and opaque types. Units that import and export opaque types allow the definition of units that can be parameterized over any type. This increases the expressivity of the unit system, but complicates the linking process. Consider the diamond import problem, in which two units that import and export an opaque type from a single source are subsequently linked together. When linking the two units together it is no longer possible to determine that the two opaque types are actually the same. The addition of translucent types to the unit system solves this problem. Units that can import and export types translucently allow for imported and exported types to refer to one another through type equations. Solving the diamond import problem is then a simple matter of tracing through type equations to determine when two types are equal.

Both of the unit models in the literature describe type systems with *structural linking*, meaning that units may link together if they export and import the same names. Meanwhile, the Racket implementation of units adopts a pragmatic view and conducts linking through statically-bound signatures. Thus, Racket uses *nominal linking*, meaning that units may only link together if their statically-bound signatures names agree. While the linking behavior is nominal, the type system that accommodates these units has a structural nature. In particular, a unit type is a subtype of another if it imports fewer or exports more signatures than another unit type. This structural subtyping rule comes from Flatt and Felleisen's original model.

### 2.3.2 *ML modules*

Unlike units, ML modules, known as structures, are not first-class values. This means that ML has no mechanism within the language to link structures at runtime like Racket's *compound-unit* form. Instead ML modules link together via *functor* application. Functors are functions from structures to structures that provide a way to combine ML modules. Since units in Racket are first-class values, there is no need for the notion of a functor, instead they can be simulated using functions and the *compound-unit* form.

Although recent implementations of Ocaml [7] include so-called first-class modules, they require the programmer to explicitly convert back and forth between modules and module values. Additionally, Ocaml includes a *module rec* form that allows recursive linking of modules, but it is limited compared to Racket's *compound-unit* form. The recursive linking form in Ocaml requires each linked module be given a type annotation and seems to prevent separate compilation of mutually recursive modules.

While units in Racket must be invoked in order to access their bindings, this is not the case in ML. ML modules allow access to the types and values they contain through a selection operator. Units in Racket use signatures and explicit invoking constructs to maintain abstractions in unit bodies. In ML this abstractions is maintained through module types. Module types in ML, called signatures, only allow access to members of the module that are mentioned in the type. This is similar to Racket's signatures which only allow access to bindings mentioned in imported signatures. Like the prior work on units, ML's type system for modules is structural. ML's module types, however, do not support subtyping.

# 3 DESIGN OVERVIEW

## 3.1 Criteria for Units in Typed Racket

The design of my type system is motivated by two points that tie into my thesis: (1) the type system should support the gradual evolution of module specifications, and (2) the type system should pragmatically support units.

I bootstrap my type system on Typed Racket's existing infrastructure since it already supports mature gradual typing for Racket. The two points above and Racket's existing design suggests that my gradual type system must contain the following elements:

- Nominal linking of units through signatures,
- Dependency tracking to order side effects in units, and
- Unit contracts to support typed-untyped interoperation.

The previous chapter introduced the distinction between signatures and units and Racket's nominal linking behavior. A type system for units must accommodate this distinction and reject invalid linking situations. Furthermore, since Racket allows the programmer to specify constraints on unit side-effects through *init-depend* clauses, the type system should track these constraints as well.

Typed Racket supports sound gradual typing by compiling type annotations into higher-order *contracts* [2] at the boundaries between compilation packages. Whenever a value flows across these boundaries, the contract system applies a check, and potentially a delayed check in the form of a wrapper, to the value. This check prevents untyped code from violating the type invariants of typed code. To soundly support units, I must extend Typed Racket with a translation from unit types to matching unit contracts. I base my target contracts off of Strickland and Felleisen's unit contracts [11] with a modified semantics that accounts for a unit's body expressions.

In the rest of this section, I explain how each design criterion is satisfied in my gradual type system design.

## 3.2 Signatures and Unit Types

The first-class nature of units and the fact that units may implement multiple signatures requires a distinct notion of a unit type. To illustrate this need, consider the following untyped implementation of a heap using units:

```
(define-signature heap^ (find-min insert merge))
(define-signature compare^ (compare))
(define h@
  (unit (import compare^)
        (export heap^)
        (define (find-min h) ...)
        (define (insert v h)
          (... (compare v ...) ...))
        (define (merge h1 h2) ...)))
```

To add type annotations to this program, we must assign a type to the variable *h@*. In ML-like languages in which modules are not first class values, the type of *h@* would be structural in terms of its imports and exports. This example corresponds roughly to a functor in ML that would accept a module of type *compare^* as an argument and return a module of type *heap^*. In Racket, where units are first class values, the distinction between functors and modules is blurred.

In my design, type annotations are added in two places. First, the programmer must annotate all signatures with type annotations for each name. Second, the programmer annotates expressions that evaluate to units with unit types. Each unit type mentions the imported and exported signatures as well as the type of the unit's body expressions. For example, a typed implementation of the heap program looks like this:

```
(define-signature heap^
  ([find-min : (-> Heap Integer)]
   [insert : (-> Integer Heap Heap)]
   [merge : (-> Heap Heap Heap)]))

(define-signature compare^
  ([compare : (-> Integer Integer Boolean)]))

(: h@ (Unit (import compare^)
            (export heap^)
            Void))
(define h@ (unit ...))
```

The type given to the *h@* unit above shows the symmetry between unit types and unit values. Declaring types in signature definitions provides a consistent interface for all units importing or exporting a given signature. Storing type information with signatures allows Typed Racket to support nominal linking in the type system for units.

## 3.3 Tracking Dependencies in Types

As described in section 2.2, expressions in unit bodies may have side-effects. To constrain these effects, programmers can specify the linking order of units using *init-depend* clauses. Recall the *results@* unit from earlier, reproduced below with matching type annotations:

```
(: results@ (Unit
              (import data^)
              (export results^)
              (init-depend data^)
              Void))

(define results@
  (unit (import data^)
        (export results^)
        (init-depend data^)
        (define results
          (... data ...))))
```

The type for *results@* specifies an additional signature in an *init-depend* clause that mirrors the clause in the unit expression. Tracking the dependencies in the unit type allows the type system to reject wrongly ordered linking clauses at compile-time.

## 3.4 Unit Contracts

In order to ensure safe interoperation between typed and untyped components, the runtime system must enforce the type systems's invariants when typed values flow into untyped contexts. Typed Racket uses Racket's contract system [2] to protect values flowing between typed and untyped modules. Strickland and Felleisen [11] introduced contracts for units that protect the values imported and exported from a given unit.

Simply checking the imports and exports of a given unit is sufficient for most uses of units in Racket because they only contain internal definitions and no body expressions. For units with body expressions, Strickland and Felleisen's contracts are insufficient. Unit contracts must also monitor the result of a unit's body when it is extracted via *invoke-unit*.

Strickland and Felleisen's implementation of unit contracts does not guard the result of invoking a unit with a contract that guarantees it maintains the expected invariants. The proposed solution to this oversight is to extend unit contracts to specify a contract that protects the result of invoking a unit. For Typed Racket, this enhancement is necessary to ensure sound interoperability between typed and untyped code. When units flow from typed to untyped code, unit contracts will ensure that the invariants of the type system hold, even when units are invoked.

## 3.5 Explicit Design Omissions

In the design of Typed Racket with units, some features present in earlier work on unit type systems were explicitly omitted. Owens and Flatt's [9] type system for units which includes opaque and translucent types was not considered for the design of Typed Racket with units. An analysis of the Racket codebase determined that very few Racket programs using units would benefit from opaque or translucent types. Most Racket programs tend to use units as replacements for modules and use units primarily to gain the advantages of cyclic linking and dynamic loading.

Furthermore, supporting opaque and translucent types in Typed Racket is challenging due to Typed Racket's lack of real opaque types that prevent reflective operations. The separation between units and signatures in Racket would be complicated by the addition of opaque and translucent types. Programmers would need to declare opaque types in signature definitions. Unit values importing and exporting signatures would then introduce type equations which must be managed in the type checking of *compound-unit* expressions and in compilation from unit types to unit contracts.

# 4 A MODEL OF UNITS

In this chapter, I present a formal model of my type system design. The formal model is encoded in the Redex modelling language [1] and is based on Flatt and Felleisen's original model [3] of units. This chapter starts with a basic, untyped language of units and then introduces a type system for that language. The model leaves out the formal details of typed-untyped interoperation and contracts.

## 4.1 The Untyped Language

Flatt and Felleisen [3] show that any language containing both *begin* and *letrec* forms can support units. Extending a language with units requires a sequencing form and a form which recursively binds names. Figure 2 presents the untyped core language that forms the basis for the semantics of units. The untyped core language is the untyped lambda calculus extended with operations on numbers and booleans and an *if* expression. The more interesting extensions in this core language are the *begin* and *letrec* forms, which are necessary when adding units to the core language.

```
e ::= v                              op ::= + | - | * | /
    | x                                   | = | > | <
    | (op e e)                            | and | or
    | (if e e e)                      v ::= boolean
    | (letrec ([x e] ...) e)              | number
    | (begin e ... e)
    | (λ (x ...) e)
    | (e e ...)
```

Figure 2: The Untyped Core Language

## 4.2 The Core Semantics

To support a simple expression of the semantics for the core language, the base language is extended to support a stateful *set!* operation as well as a *void* value, neither of which can occur in core language expressions, they only serve to simplify the presentation of the language's semantics.

Figure 3 shows the evaluation contexts, *C*, for the core language as well as the representations of the environment, *E*, and store, *S*. Environments map variable names such as *x* to locations, denoted by σ, and the store maps locations to values. Values in the language

$$
\begin{array}{rl}
v & ::= \;.... \mid \textsf{void} \mid (\lambda\ (x\ ...)\ e : E) \\
e & ::= \;.... \mid (\textsf{set!}\ x\ e) \\
E & ::= ([x : \sigma]\ ...) \\
S & ::= ([\sigma : v]\ ...)
\end{array}
\qquad
\begin{array}{rl}
C ::= & (op\ \textsf{val}\ ...\ C\ e\ ...) \\
 & \mid (\textsf{if}\ C\ e\ e) \\
 & \mid (\textsf{begin}\ C\ e\ e\ ...) \\
 & \mid (\textsf{val}\ ...\ C\ e\ ...) \\
 & \mid (\textsf{set!}\ x\ e) \\
 & \mid []
\end{array}
$$

Figure 3: Environments and Evaluation Contexts

are booleans, numbers, the *void* value, and closures consisting of a lambda and an environment.

The reduction rules for the core language are standard. Figure 4 presents only those reduction rules which are necessary to extend a language with units. The *begin* form forces the evaluation of its first subexpression to a value before evaluating subsequent expressions. A *begin* form containing only a single expression reduces to that expression. The *set!* rule evaluates its subexpression then updates the store and maps the location of the variable to its new value. The core language's semantics for *letrec* are standard. The environment is extended to include the variable names bound by the *leterec* expression and then each variable binding is evaluated in order before the body expression is finally evaluated.

$$
\begin{array}{lll}
(E\ S\ C[(\textsf{begin}\ v\ e_1\ e_2\ ...)]) & \longrightarrow & (E\ S\ C[(\textsf{begin}\ e_1\ e_2\ ...)]) \\
(E\ S\ C[(\textsf{begin}\ e)]) & \longrightarrow & (E\ S\ C[e]) \\
(E\ S\ C[(\textsf{set!}\ x\ v)]) & \longrightarrow & (E\ \textsf{extend}\ [\![S, (\sigma : v)]\!]\ \ C[\textsf{void}]) \\
 & & \text{where } \sigma = \textsf{lookup}\ [\![x, E]\!] \\
(E\ S\ C[(\textsf{letrec}\ ([x\ e_x]\ ...)\ e)]) & \longrightarrow & (\textsf{extend}\ [\![E, [x : \sigma_x], ...]\!]\ \ S\ C[(\textsf{begin}\ (\textsf{set!}\ x\ e_x)\ ...\ e)]) \\
 & & \text{where } (\sigma_x ...) \text{ fresh}
\end{array}
$$

Figure 4: Core Language Reduction Rules

## 4.3 A Language with Units

To model the functionality of Racket's unit system the core language is extended to include top-level definitions, a signature definition mechanism, and new expression forms for each of the fundamental unit operations: creating, compounding, and invoking units.

Signatures are created using the *define-signature* form which specifies the name of the signature, an optional parent signature and a list of identifiers required by the signature. Signature extensions include those variables from the base signature as well as any new identifiers. This notion of signatures is the greatest departure from prior models of units. Units in this model are linked based on the names of signatures rather than the actual identifiers imported and exported.

This restriction may seem limiting, but in most cases it is more practical than linking identifiers individually. Many signatures in Racket contain long lists of identifiers that must be defined by units exporting the signatures. Linking units exporting these signatures is made simpler by specifying signatures rather than long lists of identifiers.

The *unit* expression creates a new unit with the specified import, export, and init-depend signatures. Unit bodies contain a sequence of definitions followed by a single expression. Definitions are allowed to refer to one another and to values bound by imported signatures. Valid units must define all exported values. Imported values cannot be exported directly to prevent trivial circular linking.

The *invoke-unit* expression provides a way to evaluate the body of a unit. Only units which have satisfied their imports or import no signatures, may be invoked. Invoking a unit evaluates the unit's body expression in the context where the unit was defined extended with the unit's body definitions.

The *compound-unit* form allows the creation of compound units by specifying how one or more units will be linked together. Each unit expression in a compound expression must have its imports satisfied. Imports can be satisfied either by imports to the compound expression or by some linked units export. There are few restrictions on which signatures can be used to satisfied a units imports inside a compound expression. Recursive linking is allowed and is identical to linking between two distinct units. Initialization dependencies are the only means of disallowing certain linking scenarios in a compound-unit expression.

```
e ::= ....
   | (unit (import sig-clause ...)
           (export sig-clause ...)
           (init-depend sig ...)
           def ...
           e)                                    def ::= (define x e)
   | (invoke-unit e)                         sig-def ::= (define-signature sig (x ...))
   | (compound-unit                                    | (define-signature sig extends sig (x ...))
       (import [link : sig] ...)          sig-clause ::= (prefix x sig)
       (export link ...)
       (link (([link : sig] ...)
              e
              link ...)
             ...))
```

Figure 5: Extending the language with units

## 4.4 The Semantics of Units

Flatt and Felleisen [3] designed a semantics for units based on a translation of invoked units into *letrec* expressions and compound units into single unit expressions. The semantics of nominally linked units presented here closely follows Flatt and Felleisen's semantics for structural units. To express the semantics of units, the language is extended with a unit closure, which pairs a unit with an environment. Just as lambda expressions evaluate to closures, unit expressions reduce to values that close over the environment in which the unit is evaluated. This is necessary only to avoid complications associated with a substitution based semantics as presented in Flatt and Felleisen's original work on units.

```
v ::= .... | (unit (import sig-clause ...)
                   (export sig-clause ...)
                   (init-depend sig ...)
                   def ...
                   e : E)
```

```
C ::= ....
   | (invoke-unit C)
   | (compound-unit
       (import [link : sig] ...)
       (export link ...)
       (link (([link : sig] ...) v si ...) ...
             (([link : sig] ...) C link ...)
             (([link : sig] ...) e link ...)))
```

Figure 6: Evaluation Contexts for Units

The rule for evaluating an *invoke-unit* expression is nearly identical to the reduction presented in Flatt and Felleisen's model of units. Invoking a unit value is only allowed when that unit imports no signatures. In that case the unit reduces to a *letrec* expression which is evaluated in the environment of the unit value.

The evaluation of *compound-unit* expressions is not as simple as *unit* or *invoke-unit* expressions. Flatt and Felleisen's reduction rule for compound units generates a new unit by joining the unit body definitions from each unit being compounded and sequencing the expressions. This reduction is only correct under the side-condition that the definitions from each unit have no naming conflicts with one another. Structurally linked units impose the requirement that no imported identifier is exported. Nominal units, however, allow importing and exporting the same signature through the use of prefixes. A compound form that both imports and exports some signature cannot simply reduce to a unit value which merges the bodies of all linked units together. This can cause a naming conflict and lead to an invalid result. Therefore, the evaluation of a *compound-unit* form must perform a consistent renaming over all units being linked and handle the prefixing of imports and exports. The reduction rule defers to the *COMPOUND* metafunction which ensures that each linking clause is valid, init dependencies are satisfied, and all renaming is performed to return a single unit.

## 4.5 Types for Units

The typed language extends the untyped language with type annotations at binding sites, including signature definitions. The set of base and function types is extended with a *Unit* type for the the types of

```
(E S C[(unit (import sig_im ...)                ⟶ (E S C[(unit (import sig_im ...)
             (export sig_ex ...)                               (export sig_ex ...)
             (init-depend sig ...)                             (init-depend sig ...)
             def ...                                           def ...
             e)])                                              e : E)])

(E S C[(invoke-unit (unit (import)              ⟶ (E_unit S C[(letrec ([x e_d] ...)
                          (export sig ...)                       e_unit)])
                          (init-depend)
                          (define x e_d) ...
                          e_unit : E_unit))])

(E S C[(compound-unit                           ⟶ (E S C[(unit
        (import [link_cmp-im : sig_cmp-im] ...)              (import sig-clause_cmp-im ...)
        (export link_cmp-ex ...)                             (export sig-clause_cmp-ex ...)
        (link linking-clause                                 (init-depend)
              ...))])                                        def_cmp ...
                                                             (begin e_cmp ...)
                                                             : E_cmp)])
    where (unit                                  = (COMPOUND linking-clause ...),
            (import sig-clause_cmp-im ...)
            (export sig-clause_cmp-ex ...)
            (init-depend)
            def_cmp ...
            (begin e_cmp ...)
            : E_cmp)
          linking-clause = (([link_val-ex : sig_val-ex] ...)
                            (unit (import sig_im ...)
                                  (export sig_ex ...)
                                  (init-depend sig ...)
                                  def ...
                                  e : E_unit)
                            link_val-im ...)
```

Figure 7: Reduction Semantics of Unit Forms

unit expressions. Typechecking the typed unit language is straightforward. Non-unit expressions typecheck as in traditional models of the typed lambda calculus with subtyping. The *Unit* type allows a subtyping relation in which one unit type is a subtype of another if it imports fewer signatures, exports more signatures, or the body types are subtypes of one another.

Typechecking the unit language requires a new environment, labeled $\Sigma$, that maps signature names to the variables bound by that signature and their corresponding types. This environment could be merged with the rest of the type environment, but is kept separate to emphasize its role in typechecking units.

The typechecking rule for the *unit* form is structurally similar to the rule for typechecking *letrec*. A *unit* expression is well typed if its body definitions and expression are well typed in the context of its imported signatures. Other conditions of unit typechecking simply ensure that a unit is well-formed. Well formed units are those that do not import the same signature or identifier more than once, do not directly export an imported identifier, and define all expected exports with the correct types.

An *invoke-unit* expression is well-typed when the expression being invoked is a well-typed, import-free unit. This ensures that only units which have satisfied their imports can be invoked. The type of an *invoke-unit* expression is the body type of the unit subexpression.

Typechecking the *compound-unit* form is the most complex of the typechecking rules added to support typed units. The additional complexity comes from the need to track dependency information in the type system. The *Links* judgment operates over all the linking clauses of a *compound-unit* form to ensure that each link is valid. A given linking clause is valid when the type of the expression being linked is a unit type whose imports and exports are consistent with those declared in the linking clause and any *init-depend* signatures mentioned in the type must have appeared above the current linking clause. This guarantees that units can be linked together safely according to user-specified init dependencies.

$$
\frac{\begin{array}{c}
\{\mathit{sig\text{-}id}_{id}, \ldots\} \subseteq \{\text{sig-name}\llbracket \mathit{sig}_{im} \rrbracket, \ldots\} \\
\text{bindings}\llbracket \Sigma, (\mathit{sig}_{im} \ldots) \rrbracket \cap \text{bindings}\llbracket \Sigma, (\mathit{sig}_{ex} \ldots) \rrbracket = \varnothing \\
\text{distinct}(\Sigma, (\text{sig-name}\llbracket \mathit{sig}_{im} \rrbracket \ \ldots)) \\
\text{distinct}(\Sigma, (\text{sig-name}\llbracket \mathit{sig}_{ex} \rrbracket \ \ldots)) \\
\text{distinct}(\Sigma, (\mathit{sig\text{-}id}_{id} \ldots)) \\
(\mathit{sig\text{-}id}_{im} \ldots) = (\text{sig-name}\llbracket \mathit{sig}_{im} \rrbracket \ \ldots) \\
(\mathit{sig\text{-}id}_{ex} \ldots) = (\text{sig-name}\llbracket \mathit{sig}_{ex} \rrbracket \ \ldots) \\
\Gamma_{ext} = \text{extend}\llbracket \Gamma, ([d : \tau_d] \ldots), \text{sig-elts}\llbracket \Sigma, \mathit{sig}_{im} \rrbracket, \ldots \rrbracket \\
\Gamma_{ext} \mid \Sigma \vdash \mathit{expr}_d : \ \tau_{de} \quad \ldots \\
\Gamma_{ext} \mid \Sigma \vdash \tau_{de} \leq \tau_d \quad \ldots \\
\Gamma_{ext} \mid \Sigma \vdash \mathit{expr} : \ \tau \\
\Gamma_{exports} = \text{extend}\llbracket \text{sig-elts}\llbracket \Sigma, \mathit{sig}_{ex} \rrbracket, \ldots \rrbracket \\
\text{bindings}\llbracket \Sigma, (\mathit{sig}_{ex} \ldots) \rrbracket \subseteq (d \ldots) \\
\Gamma \mid \Sigma \vdash \tau_d \leq \text{lookup}(d, \Gamma_{exports}) \quad \ldots
\end{array}}{\begin{array}{l}
\Gamma \mid \Sigma \vdash (\text{unit } (\text{import } \mathit{sig}_{im} \ldots) \\
\qquad\qquad (\text{export } \mathit{sig}_{ex} \ldots) \\
\qquad\qquad (\text{init-depend } \mathit{sig\text{-}id}_{id} \ldots) \\
\qquad\qquad (\text{define } d : \tau_d \ \mathit{expr}_d) \ldots \\
\qquad\qquad \mathit{expr}) : \\
(\text{Unit } (\text{import } \mathit{sig\text{-}id}_{im} \ldots) \\
\qquad (\text{export } \mathit{sig\text{-}id}_{ex} \ldots) \\
\qquad (\text{init-depend } \mathit{sig\text{-}id}_{id} \ldots) \\
\qquad \tau)
\end{array}}
$$

$$
\frac{\begin{array}{l}
\Lambda_{links} = ([\mathit{link\text{-}id}_{im} : \mathit{sig\text{-}id}_{im}] \ldots \\
\qquad\qquad [\mathit{link\text{-}id}_{unit\text{-}ex} : \mathit{sig\text{-}id}_{unit\text{-}ex}] \ldots \ldots) \\
(\mathit{sig\text{-}id}_{cmp\text{-}exp} \ldots) = (\text{lookup}(\mathit{link\text{-}id}_{ex}, \Lambda_{links}) \ldots) \\
\Gamma \mid \Sigma \vdash \mathit{expr}_{unit} : \ \tau_{unit} \quad \ldots \\
\text{Links}\llbracket \Gamma, \Sigma, \Lambda_{links}, \\
\qquad (\mathit{link\text{-}id}_{im} \ldots), \\
\qquad (((\mathit{link\text{-}id}_{unit\text{-}ex} \ldots) \ \tau_{unit} \ (\mathit{link\text{-}id}_{unit\text{-}im} \ldots)) \ldots), \\
\qquad \mathit{any}, \\
\qquad (\mathit{sig\text{-}id}_{cmp\text{-}id} \ldots), \\
\qquad \tau \rrbracket
\end{array}}{\begin{array}{l}
\Gamma \mid \Sigma \vdash (\text{compound-unit} \\
\qquad (\text{import } [\mathit{link\text{-}id}_{im} : \mathit{sig\text{-}id}_{im}] \ldots) \\
\qquad (\text{export } \mathit{link\text{-}id}_{ex} \ldots) \\
\qquad (\text{link } (([\mathit{link\text{-}id}_{unit\text{-}ex} : \mathit{sig\text{-}id}_{unit\text{-}ex}] \ldots) \\
\qquad\qquad \mathit{expr}_{unit} \\
\qquad\qquad \mathit{link\text{-}id}_{unit\text{-}im} \ldots) \ldots)) : \\
\\
\qquad (\text{Unit } (\text{import } \mathit{sig\text{-}id}_{im} \ldots) \\
\qquad\qquad (\text{export } \mathit{sig\text{-}id}_{cmp\text{-}exp} \ldots) \\
\qquad\qquad (\text{init-depend } \mathit{sig\text{-}id}_{cmp\text{-}id} \ldots) \\
\qquad\qquad \tau)
\end{array}}
$$

$$
\frac{\begin{array}{c}
\Gamma \mid \Sigma \vdash \mathit{expr} : \ \tau_{unit} \\
\Gamma \mid \Sigma \vdash \tau_{unit} \leq (\text{Unit } (\text{import}) \ (\text{export}) \ (\text{init-depend}) \ \text{Any}) \\
(\text{Unit } (\text{import}) \ (\text{export } \mathit{sig\text{-}id} \ldots) \ (\text{init-depend}) \ \tau) = \tau_{unit}
\end{array}}{\Gamma \mid \Sigma \vdash (\text{invoke-unit } \mathit{expr}) : \ \tau}
$$

Figure 8: Typechecking Rules for Units

# 5 IMPLEMENTING TYPED UNITS

## 5.1 Overview

This chapter gives an overview of the implementation of units in Typed Racket. I introduce the notion of *trampolining* macros to implement the syntactic support for typed units. To support typed-untyped interaction, this chapter also explains the details of type to contract compilation for unit types. The chapter closes with a discussion of the limitations of Typed Racket's implementaion of units.

Implementing units for Typed Racket presents a syntactic challenge that requires more than just translation the typing judgements from the previous chapter into code. Since Typed Racket accommodates the full range of syntactic extensions that come with Racket, it typechecks a source program by first fully expanding it into the core language forms [13]. This allows Typed Racket to accommodate most syntax extensions without extending the type system. Since this core language is fully desugared, high-level information about complex syntactic forms may be lost in the expansion process.

Typechecking complex, high-level macros such as units requires cooperation from the macro implementation so that the typechecker can recover the information it needs to typecheck the form. Concretely, this means that Typed Racket needs to provide a wrapper unit macro that annotates the unit expression with typechecking annotations. The typechecker looks at these annotations to recover the information needed by the unit typechecking rules.

## 5.2 Racket's Unit Library

Racket implements units through a large collection of macros. This implementation includes signatures, the core unit forms introduced in section 4.3, and many variations on these core forms that reduce the syntactic burden of programming with units. The nominal nature of Racket's units requires an implementation that binds variables which are not lexically apparent in unit expressions. Specifically, it is impossible to distinguish from the syntax of a unit expression whether a variable referenced in the unit's body is bound by an imported signature or the outer context of the unit expression. Thus, Racket's unit macro is necessarily unhygienic [6].

Racket's unit macro performs a renaming on all imported and exported variables as well as internal definitions. This avoids possible naming conflicts that can occur when linking units together. Unfortunately, this renaming strategy complicates the implementation of macros that must cooperate with Racket's unit form.

## 5.3 A Failed Attempt at Typed Units

The unhygienic nature of the unit macro poses a significant challenge to annotating the syntax of unit expressions for typechecking. Once syntax is annotated with the appropriate information, implementing typechecking for units is a direct translation of the typing judgments from section 4.5 into code. Annotating unit syntax requires the ability to map external signature names to their internal renaming. Recovering the internal names is necessary to correctly associate the types of signature-bound variables in the body of a unit.

The naive attempt to recover internal names by inserting the signature-bound identifiers into the body of the unit, fails. Inserting these identifiers directly into a unit expression gives them an incompatible context, resulting in a different internal renaming. Furthermore, attempting to annotate unit body definitions and expressions with information for typechecking by performing local expansion on each definition and expression in the unit body also results in unbound identifier errors. Local expanding sub-expressions of a unit's body modifies the context of the sub-expressions and corrupts lexical binding within the unit. This causes identifiers that should be bound within the unit body to trigger unbound identifier errors. Overall, an attempt to annotate unit syntax without cooperation from the untyped unit macro will not succeed.

## 5.4 Trampolining Macros

The process of annotating a unit expression for typechecking must be able to handle the renaming and variable binding schemes that the unit macro creates. The complex and unhygienic expansion of the unit macro that occurs during compilation must be replicated by the annotation process in order to maintain the correct binding structure. To solve this problem, I adapt the style of trampolined programs [5] to the design of macros. Trampolining macros are series of two or more macros that cooperate with one another by allowing control of the macro expansion process to bounce back and forth between each involved macro.

Using a trampolining style, the typed unit macro becomes substantially less complex than the implementation that manually iterates through each expression in the unit body and performs local expansion. The typed unit macro, depicted in figure 9, becomes a nearly trivial wrapper over its untyped equivalent. The macro inserts a table that is used to map external signature names to their internal

```
(define-syntax (typed-unit stx)
  (syntax-parse stx
   [(_ (import im-sig ...)
       (export ex-sig ...)
       body ...)
     ; Defer to the untyped unit macro
     #`(unit (import im-sig ...)
             (export ex-sig ...)
             ; The table mapping external to internal names
             #,(import/export-table stx)
             ; Wrap the unit body with the annotate macro
             (annotate body ...))]))

(define-syntax (annotate stx)
  (syntax-parse stx
    [(_) #'(begin)]
    [(_ e)
      (syntax-parse (local-expand #'e ...)
        [(begin b ...) #'(annotate b ...)]
        [(define-syntaxes (d ...) body)
          ; Leave macro definitions unchanged
          #'(define-syntaxes (d ...) body)]
        [(define-values (d ...) body)
          ; Mark definitions for later typechecking
          #`(define-values (d ...)
              #,(tag-unit-body-def
                  #'(#%expression
                      (begin
                        (void (λ () d) ...)
                        body))))]
        ; Mark other expressions for later typechecking
        [_ (tag-unit-body-expr ...)])]
    [(_ e ...) #'(begin (annotate e) ...)]))
```

Figure 9: Trampolining Macros

renamings and wraps the entire unit body in a macro that marks each definition or expression for typechecking.

Expansion of the typed unit macro begins by yielding control to the untyped unit macro. The untyped unit macro first expands the table, leaving syntax that the typechecker uses to recover the internal names of signature-bound variables. After the table has expanded, the trampolining process begins and the unit macro cedes control to the annotating macro. The trampolining process then bounces control of expansion back and forth between the unit macro and the annotating macro. The annotating macro only needs to head-expand expressions and definitions to mark them for typechecking. An added benefit of the trampolining implementation is that macro definitions in unit bodies do not need to be treated specially by the annotating macro. Instead, Racket's macro expansion process handles the expansion of macros in unit bodies.

To better understand the trampolining process consider a simple use of the typed unit macro below:

```
(typed-unit (import)
            (export)
  (define x 5)
  (+ x 12))
```

This unit imports and exports no signatures and contains only a single internal definition and body expression. Since the unit has no imports or exports the table that maps signature names to their renamings is unnecessary for this example. Following the trampolining process described above the *typed-unit* macro expands into the *unit* macro and wraps the unit body with the *annotate* macro.

```
(unit (import)
      (export)
  (annotate
   (define x 5)
   (+ x 12)))
```

The untyped *unit* macro cannot immediately perform anymore expansion so it yields control of expansion to the *annotate* macro.

```
(unit (import)
      (export)
  (annotate (define x 5))
  (annotate (+ x 12)))
```

The expansion of the *annotate* macro wraps each subexpression and splices it into the body of the unit. Control of expansion returns the the *unit* macro, but it immediately yields control back to the first instance of the *annotate* macro.

```
(unit (import)
      (export)
  (@ (define-values (x) 5))
  (annotate (+ x 12)))
```

When the *annotate* macro encounters a definition or an expression it performs head-expansion, expanding Racket's *define* form into the core *define-values* form. Additionally, the *annotate* macro leaves a mark that tells the typechecker to typecheck the given expression. In the example, this mark is denoted by the @ symbol.

```
(unit (import)
      (export)
  (@ (define-values (x) 5))
  (@ (#%plain-app + x 12)))
```

Finishing off the example, the trampolining process continues until each form in the unit's body has been annotated. This result is the unit above with its body expanded and marked for typechecking.

## 5.5 Implementing Type to Contract Translation

For sound interoperation, Typed Racket must compile unit types to unit contracts at compilation package boundaries. While I already explained the design of unit contracts, I did not describe the compilation strategy. Consider again the implementation of a heap with typed units:

```
(define-signature compare^
  ([compare : (-> Integer Integer Boolean)]))
(define-signature heap^
  ([insert : (-> Integer Heap Heap)]
   ...))

(: h@ (Unit (import compare^)
            (export heap^)
            Void))
(define h@
  (unit (import compare^)
        (export heap^)
        (define (insert n h) ...)
        ...))
```

When *h@* is exported, Typed Racket needs to generate a contract that guards its invocations in other compilation contexts. Assuming that *heap/c* is bound to a contract combinator that corresponds to heap types, Typed Racket will produce the following contract for *h@*:

```
(unit/c
  (import
   (compare^ [compare (-> integer? integer? boolean?)]))
  (export
   (heap^ [insert (-> integer? heap/c heap/c)] ...))
  void?)
```

The signatures in both imports and exports are expanded and each type in each signature is converted to a contract. Since signatures are expanded during the type to contract compilation, contract generation only applies to unit values and unit types. Signatures may be exported and imported without any contract intervention. After all, signatures are not first-class values.

## 5.6 Implementation limitations

Due to implementation challenges, several features of the base unit library are still left unsupported in Typed Racket. In this section, I detail the scope of these limitations and future plans to rectify them.

### 5.6.1 *Signature Forms*

Racket's unit system includes many syntactic forms to make programming with units convenient. This includes linking and invocation forms which infer the signature specifications required for linking and invocation. These forms use Racket's macro system to associate static information with identifiers that refer to units. Typed Racket's implementation of units is able to use this static information to support typechecking these forms. Racket also allows signature definitions to contain more than simple lists of identifiers. Besides identifiers, signature definitions can contain macro definitions, variable definitions for unit's importing the signature, and variable definitions for units exporting the signature among others.

```
(define-signature sig^
 (x y z
 (struct tree (val left right))
 (define-values (a b c) ...)
 (define-values-for-export (d e f) ...)
 (define-syntaxes (s t) ...)))
```

Figure 10: A Racket Signature

Figure 10 shows many of the signature forms that Racket supports. Typed Racket should accommodate the idioms of unit programming in untyped Racket, but it is unclear how to incorporate all features of signatures in Racket. The *define-values* and *define-values-for-export* signature forms pose a particular problem for typechecking. These definition forms bind variables first from the signature definition and then from the environment in which the signature definition occurs. This complex binding structure makes typechecking these definitions difficult. The Typed Racket typechecker operates only on fully expanded programs, but signature definitions are processed and registered before other top level definitions are typechecked. Signature definitions, therefore, do not have full information to typecheck these definitions during signature processing. The implementation of units as macros could allow typechecking these signature definitions along with unit typechecking. This strategy, however, is likely to produce unhelpful error messages that refer to unit expressions rather than signature definitions. Additionally, if the types of definitions are not known inside of a signature then compiling unit types into unit contracts can no longer guarantee the safety of typed-untyped interactions. For these reasons, Typed Racket disallows variable definitions that occur inside *define-signature* forms.

Typed Racket does not currently support the *struct* form in signature definitions or struct definitions in unit bodies. Racket's struct definitions are generative meaning that each struct definition creates a new struct-type even if it shares the same name as another struct. Since units may be invoked multiple times, a unit body containing a struct definition would create multiple incompatible structure definitions. The generativity of structs prevents Typed Racket from being able to typecheck struct definitions that occur in internal definition contexts. For example, a unit that defines and exports a struct definition may be invoked multiple times. The typechecker must assign a single type to the unit's body, but each invocation of the unit would create incompatible structure values. The result is a program that typechecks, yet fails with a runtime type error.

### 5.6.2 *Parametricity*

The *Pairing Heap* case study required a partial rewrite of the original program. This rewrite required monomorphizing the program and specializing it to integers. The original program could create heaps of any "type" that supported an implementation of the *compare^* signature. This suggests that there are programs that would benefit from a type system that allows parameterizing over the types in a signature definition. Specifically, such a type system would allow a direct porting of the *Pairing Heap* program to Typed Racket, without the need to monomorphize the program.

The top of Figure 11 gives the definitions of the *compare^* signature in the monomorphized version of the Pairing Heap program and a unit intending to compare integers. The bottom of the figure displays a hypothetical syntax for signatures which can be parameterized over types in a style similar to ML modules. Typed Racket cannot currently deal with this sort of parametrization due to a lack of true opaque types. A unit that imports a signature containing a variable with an opaque type should not be able to introspect and refine the type of that variable. Furthermore, the interaction between typed and untyped code is unclear in the presence of these parametric signatures. Owens and Flatt [9] presented a type system which handles this sort of parametricity for structural units. This thesis has

```
(define-signature compare^
  ([compare : (-> Any Any Boolean)]))
(define int-compare@
  (unit (import)
        (export compare^)
        (define (compare x y)
          (and (exact-integer? x)
               (exact-integer? y)
               (<= x y)))))
```

---

```
(define-signature compare^
  #:opaque (t)
  ([compare : (-> t t Boolean)]))
(define int-compare@
  (unit (import)
        (export (compare^ #:with ([t Integer])))
        (define (compare x y) (<= x y))))
```

---

Figure 11: Monomorphic vs. Parametric Signatures

demonstrated that the Flatt and Felleisen model of units extends to support Racket's nominal units. Since Owens and Flatt's primary extension to the Flatt-Felleisen unit model is the addition of opaque and translucent types, this should extend to the type system for Racket's nominal units as well.

# 6 EVALUATION

## 6.1 The Goal

The thesis of gradual typing is that programmers can ease the maintenance and evolution of their existing code bases by adding type annotations. This implies that a gradual type system needs to support the programmer's idioms and the necessary annotations are not too onerous. Therefore, to evaluate my gradual type system for units we need a two-pronged approach with these two evaluation criteria:

- Typed Racket with units supports common Racket idioms, and
- the syntactic overhead of adding type annotations is reasonable

To investigate whether gradually typed units meet these criteria, I ported several unit-based programs from Racket to Typed Racket. The porting process reveals what idioms are used in practice and what kinds of type annotations are necessary for unit-based programs.

## 6.2 Cases

My evaluation corpus consists of three case studies. Two of the case studies, *Gobblet* and *Paint by Numbers*, are part of the *Games* package distributed with Racket. The third is a user-contributed package from the PLaneT [8] package distribution system. The three programs were chosen to illustrate three distinct use cases of units. *Gobblet* breaks up the implementation of the game over several files that each implement a unit. The units are linked together and invoked to run the game. *Paint by Numbers* is a puzzle game which implements each of its puzzle instances as a unit. Puzzle instances are loaded from text files containing the problem description using Racket's *include* macro then linked together through a function that performs a fold over the list of units. The *Pairing Heap* program demonstrates the parametric features of units by defining a heap unit that depends on a user specified comparison function.

## 6.3 The Process

Adding types to untyped code is usually a straightforward process. The programmer needs to add annotations to top-level bindings and function definitions Often the programmer can determine the types of expressions from documentation comments or behavioral contracts on expressions. In poorly documented programs, the programmer must deduce the appropriate type.

The similarity between unit bodies and Racket compilation packages simplifies the porting process, the programmer only needs to keep in mind the types of identifiers bound by imported signatures. In order to add type annotations to units that import signatures, those signatures must first declare the types of their member variables. Without context, it is impossible to determine the types of variables contained in a signature. The programmer can only decipher the types of signature variables by observing uses of those variables in units that import or export a given signature. Thus, the process of porting units and signatures to Typed Racket requires a two-way refinement process in which types determined from unit bodies aid in determining types in signatures which subsequently feeds back into the process of typing unit bodies. Without external specifications, porting signatures and units to Typed Racket must be performed in tandem with one another.

| Program | Gobblet | Paint by Numbers | Pairing Heap |
|---|---|---|---|
| Lines | 1785 | 141 | 164 |
| % Increase | 13 | 22 | 44 |
| Signature ann. | 36 | 8 | 11 |
| Useful ann. | 80 | 10 | 17 |
| λ: ann. | 76 | 0 | 6 |
| Other ann. | 18 | 0 | 6 |
| Type def. | 9 | 1 | 2 |
| Assert/cast | 27 | 1 | 0 |
| Ann./100L | 12 | 13 | 24 |

Figure 12: Case Studies

## 6.4 Results

The table in figure 12 presents an analysis of results of porting the three case studies to Typed Racket. For each case study we report the the number of lines in the ported program as well as the percentage increase in number of lines from the original untyped program. We report these added annotations in several disjoint categories. The *Signature annotations* row is the number of identifiers in signatures that were assigned types. *Useful annotations* are those annotations declared for top-level and unit-body identifiers and the fields of structure definitions. These are the annotations which add information that is useful to the typechecker and to programmers as a form of documentation. The *λ: annotations* are the types added to variables bound as arguments to anonymous functions. Finally, the *other annotations* category includes all other annotations in the fully typed program.

The *Type definition* row counts the number of type definitions that were added to the program. In some circumstances, even in a fully annotated program, the type checker cannot prove that an expression has the correct type. In these cases the programmer can *assert* or *cast* the expression to satisfy the typechecker by adding a runtime check. The number of assertions and casts needed in a given program is collected in the *Assert/cast* row. The final row in the table reports the number of annotations needed per hundred lines of code.

## 6.5 Discussion

Overall the process of porting programs to Typed Racket with Units incurred reasonable syntactic overhead. The increase in number lines over all three programs was 14%, which is on par with a similar experiment conducted for Object-Oriented Typed Racket. One of the main causes of an increase in the number of lines is due to typed signature definitions. In Racket many signature definitions list all identifiers on a single line of code, but this becomes unreadable when signatures contain type declarations. This leads to splitting a typed signature definition over multiple lines. For smaller programs, like the *Pairing Heap*, this is especially noticeable.

The *Pairing Heap* program posed additional challenges. It was originally written in MzScheme, a predecessor of Racket, and required a partial rewrite into Racket before it could be ported to Typed Racket. Additionally, the program's *pairing-heap@* unit is intended to be parameterized over the implementation of a comparison function. Since Typed Racket does not currently support signatures with opaque types, the program was ported to work only with integers.

There are at least two possible future directions for further evaluation. First, a performance study to determine the overhead of unit contracts in partially ported programs would further address the overhead imposed by gradually typed units. Second, a soundness evaluation is necessary to provide a proof that unit contracts soundly mediate interactions between typed and untyped units.

### 6.5.1 *Threats to Validity*

As possible threats to the validity of the experiment we note that only the *Pairing Heap* program was ported in its entirety. *Gobblet* is a very large program of approximately 4000 lines of code, for this experiment we ported only the non-gui portion of the program which still measured nearly 2000 lines. *Paint by Numbers* contained only two files that made any use of units, so this experiment ported only those two files. The *Paint by Numbers* game, however, has one of the more interesting uses of units in the Racket codebase, a function which folds over a list of units linking them together.

# BIBLIOGRAPHY

[1] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2010.

[2] R. B. Findler and M. Felleisen. Contracts for Higher-Order Functions. In *Proc. ICFP*, pp. 48–59, 2002.

[3] M. Flatt and M. Felleisen. Cool Modules for HOT Languages. In *Proc. PLDI*, pp. 236–248, 1998.

[4] M. Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. `http://racket-lang.org/tr1/`

[5] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined Style. In *Proc. ICFP*, pp. 18–27, 1999.

[6] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic Macro Expansion. In *Proc. LFP*, pp. 151–161, 1986.

[7] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system (release 4.02): Documentation and user's manual. 2014. `http://caml.inria.fr/pub/docs/manual-ocaml/`

[8] J. Matthews. Component Deployment with PLaneT: You Want it Where? In *Proc. SFP*, 2006.

[9] S. Owens and M. Flatt. From Structures and Functors to Modules and Units. In *Proc. ICFP*, pp. 87–98, 2006.

[10] D. L. Parnas. On the criteria to be used in decomposing systems into modules. In *Proc. CACM*, 1972.

[11] T. S. Strickland and M. Felleisen. Contracts for First-Class Modules. In *Proc. DLS*, pp. 27–38, 2009.

[12] S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *Proc. POPL*, pp. 395–406, 2008.

[13] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as Libraries. In *Proc. PLDI*, pp. 132–141, 2011.