

Written Response For The Qualifying Examination

Caner Deric

Department of Computer Science
Indiana University

August 24, 2018

1) Modeling Linklets Using PLT Redex

Model the basics of the new Racket "linklet" module system using PLT Redex. Use random testing to confirm that the model produces the same answers as both the existing implementation and the implementation that you have produced using the Pycket JIT.

2) Adding Pairs To The Denotational Model Of The Untyped CBV Lambda Calculus

Extend Jeremy's denotational model of the untyped lambda calculus with pairs. Extend the proof of correspondence with the operational semantics in Isabelle to handle pairs.

3) Measuring The Performance Of The Linklet System On Pycket

Measure the performance of your changes to the Pycket JIT on the existing benchmarks. For at least 3 performance differences, propose a hypothesis that explains the difference.

1 Modeling Linklets Using PLT Redex

Model the basics of the new Racket "linklet" module system using PLT Redex. Use random testing to confirm that the model produces the same answers as both the existing implementation and the implementation that you have produced using the Pycket JIT.

```
(linklet [[imported-id/renamed ...] ...]
         [exported-id/renamed ...]
         defn-or-expr ...)

imported-id/renamed = imported-id
                    | (external-imported-id internal-imported-id)

exported-id/renamed = exported-id
                    | (internal-exported-id external-exported-id)
```

Fig. 1. Linklet Grammar

1.1 A Step Back : Racket on Chez Scheme

The previous implementations of Racket (before 7.0) were relying heavily on the C implementation of the core parts, such as the module system, macro system, etc, which posed a lot of difficulties for code maintenance, porting and performance improvements. Having a major part of Racket being implemented in Racket would make it easier for everyone to contribute, and enable Racket to be easily ported to other platforms. Therefore, in the pursuit of having large portion of Racket written in Racket, the core parts such as the module system and the macro system have been re-written in Racket by Matthew Flatt in 2016. Unfortunately, he reported a performance slowdown up to a factor of two in time and around %25 in space. In order to address this, Racket decided to adopt Chez Scheme as its runtime, rather than trying to improve its own VM. [1]

The Racket on Chez Scheme endeavor proved to be a promising means towards improving Racket's portability, and its implementation in general. One of the first steps was to separate the expansion and compilation, to enable a better transition to a different compiler and a runtime system. To this end, the compiler is made to work with "linklets", which are "lambda-like blocks of code that import and export variables", that live below the abstractions like modules and syntax objects. Having linklets as the new unit of compilation, Racket builds its implementation of modules, macros, and top-level evaluation, all on linklets. [2]

1.2 Linklets

As shown in Figure 1, a linklet has imported variables, exported variables, definitions and Racket core expressions. It may export variables with different names, and similarly

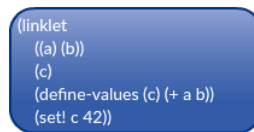


Fig. 2. An Example Linklet

it may rename any variable that's imported from another linklet. The *defn-or-expr* form is an expression of a restricted version of fully expanded programs. So the linklet body contains Racket core forms like *define-values*, *set!*, *lambda*, *let-values*, etc. A linklet can import and/or define variables, manipulate variables, and export them for other linklets to use. Figure 2 shows an example linklet, that imports variables *a* and *b* and defines the variable *c* and exports it.

The action starts when a linklet is “instantiated”. The instantiation has two possible outcomes; either a *linklet instance*, or a value. A linklet instance is essentially a container that exports variables for other linklets to use.

Instantiation begins with the linklet receiving a set of linklet instances, each exporting the variables that the linklet (being instantiated) wants to import. The resulting linklet instance then becomes ready to be used in other instantiations. Figure 3 shows the instantiation of the linklet shown above in Figure 2.

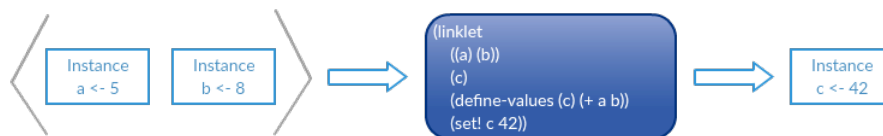


Fig. 3. Basic Linklet Instantiation

Running a linklet

As mentioned above, the second possible outcome of an instantiation is a value. That happens when a linklet is instantiated with an extra input linklet instance, namely the “target instance” (or just the target, for short).

When a target is provided to a linklet instantiation, the body of the linklet is evaluated just like in the regular instantiation, however the result of the instantiation is not an instance, but it's the result of the last expression in the linklet's body (similar to a *begin* form). Although it resembles a function application, the difference is that the linklet being instantiated can use the variables defined in the target, modify them and/or add

new variables to the target, so the target instance will have the new versions when its re-used again in another instantiation (either as an import or again as a target). Moreover, the linklet may export a variable that doesn't have a corresponding definition in its body (effectively defining it as uninitialized) and may use the value provided by the target for that variable (instead of setting the target's variable to uninitialized). This essentially builds the foundation on which the core support for the top-level evaluation is going to be established, "where variables may be referenced and then defined in a separate element of compilation". [3]

In the remainder of this section, we are going to introduce the Redex model of the linklets, which models the semantics of the instantiation and the interactions between the instances. We will also talk about how the model is tested both against Pycket's test suite and the Redex's random testing.

```

e ::= x | v | (e e ...) | (if e e e) | (p1 e) | (p2 e e)
    | (set! x e) | (begin e e ...)
    | (lambda (xl ...) e) | (let-values (((xl) e) ...) e)
    | (raises e)
v ::= n | b | c | (void)
c ::= (closure x ... e Σ)
n ::= number
b ::= true | false
x ::= variable-not-otherwise-mentioned
p1 ::= add1
p2 ::= + | * | <
o ::= p1 | p2
E ::= [] | (v ... E e ...) | (o v ... E e ...) | (if E e e)
    | (begin v ... E e ...) | (set! x E)
    | (let-values (((x) v) ... ((x) E) ((x) e) ...) e)
Σ ::= ((x any) ...)
σ ::= ((x any) ...)

```

Fig. 4. RC language grammar

1.3 Developing the Redex Model

Since the linklets contain Racket core forms in their bodies, we first start by developing the Redex model, "Racket Core", which models the evaluation of the Racket core forms and expressions. Next we move on to developing the "Linklet" model, that models the linklets, and uses the Racket Core model in instantiation to evaluate the expressions in the linklet body. We keep the interaction between the two models as minimal as possible to make the semantics clear. In other words, the Racket Core model has no idea about the semantics of linklet instantiation, and similarly the Linklet model doesn't know (or care) about evaluating Racket forms.

The Racket Core Model

As shown in the grammar in Figure 4, we have core forms like *begin*, *let-values*, *if*, *lambda*, along with some unary and binary primitives; and also numbers, booleans, closures and (*void*) as values and the Σ as the lexical environment. We also have mutation implemented with *set!* and σ as the store, so the Σ maps variables to cells (i.e. references to store) and σ maps cells to values.

Figure 5 shows the standard reduction relation for the Racket Core model. δ handles the primitive applications, βv handles the function application, and others are pretty straightforward. Note that we also have basic exception handling, where the *raise* form will discard the evaluation context and will be turned into the keyword *stuck*. As we will talk about random testing in Section 1.4, error handling is often useful to have when demonstrating the agreement between the Redex model and the actual implementation. We want the model to signal an exception whenever the actual implementation produces an error.

$$\begin{array}{ll}
[E[(\text{raises } e)] \Sigma \sigma] \longrightarrow [(\text{raises } e) \Sigma \sigma] & [\text{error}] \\
[E[x] \Sigma \sigma] \longrightarrow [E[\text{lookup}[\sigma, x_i]] \Sigma \sigma] & [\text{lookup}] \\
\quad \text{where } x_i = \text{lookup}[\Sigma, x] & \\
[E[(\text{lambda } (x \dots) e)] \Sigma \sigma] \longrightarrow [E[(\text{closure } x \dots e \Sigma)] \Sigma \sigma] & [\text{closure}] \\
[E[(\text{set! } x \ v)] \Sigma \sigma] \longrightarrow [E[(\text{void})] \Sigma \text{extend}[\sigma, (c_i), (v)]] & [\text{set!}] \\
\quad \text{where } (\text{lookup}[\Sigma, x] \neq (\text{raises } x)), c_i = \text{lookup}[\Sigma, x] & \\
[E[(\text{begin } v_1 \dots v_n)] \Sigma \sigma] \longrightarrow [E[v_n] \Sigma \sigma] & [\text{begin}] \\
[E[(\text{let-values } (((x) \ v) \dots) e)] \Sigma \sigma] \longrightarrow [E[e] \text{extend}[\Sigma, (x \dots), (x_2 \dots)] \text{extend}[\sigma, (x_2 \dots), (v \dots)]] & [\text{let}] \\
\quad \text{where } (x_2 \dots) = (\text{variables-not-in } e \ (x \dots)) & \\
[E[(\text{if } v_0 \ e_1 \ e_2)] \Sigma \sigma] \longrightarrow [E[e_1] \Sigma \sigma] & [\text{if-true}] \\
\quad \text{where } (v_0 \neq \text{false}) & \\
[E[(\text{if false } e_1 \ e_2)] \Sigma \sigma] \longrightarrow [E[e_2] \Sigma \sigma] & [\text{if-false}] \\
[E[(\delta \ v_1 \ v_2 \dots)] \Sigma \sigma] \longrightarrow [E[\delta[(\delta \ v_1 \ v_2 \dots)]] \Sigma \sigma] & [\delta] \\
[E[(\text{closure } x \dots_n e \ \Sigma_i \ v \dots_n)] \Sigma_2 \sigma] \longrightarrow [E[e] \text{extend}[\Sigma_i, (x \dots), (c_2 \dots)] \text{extend}[\sigma, (c_2 \dots), (v \dots)]] & [\beta v] \\
\quad \text{where } (c_2 \dots) = (\text{variables-not-in } e \ (x \dots)) &
\end{array}$$

Fig. 5. Racket Core standard reduction relation

The evaluation is done by repeatedly applying the standard one-step reduction, effectively implementing the reflexive-transitive closure, as shown in Figure 6. Note that the *run-rc* recognizes the *raises* form and reduces it to the keyword “stuck” to essentially model raising an exception. Otherwise, it will either return values (numbers, booleans, closures, (void)), or repeatedly apply the reduction. Note that we output only the keyword “closure” for any closure values, since there is no way of comparing Racket closures with the ones we output, as the Racket closures don’t have explicit representations.

$\text{run-rc}[(n \ \Sigma \ \sigma)]$	$= n$
$\text{run-rc}[(b \ \Sigma \ \sigma)]$	$= b$
$\text{run-rc}[(c \ \Sigma \ \sigma)]$	$= \text{closure}$
$\text{run-rc}[(\text{void}) \ \Sigma \ \sigma]$	$= (\text{void})$
$\text{run-rc}[(\text{raises } e) \ \Sigma \ \sigma]$	$= \text{stuck}$
$\text{run-rc}[any_i]$	$= \text{run-rc}[any_{again}]$
where $(any_{again}) = (\text{apply-reduction-relation } \rightarrow_{\beta s} (\text{term } any_i))$	
$\text{run-rc}[any_i]$	$= \text{stuck}$

Fig. 6. Racket Core evaluator

Linklets Model

As discussed above, linklets are like *lambda* forms with additional exported variables and Racket Core forms in the body, while the linklet instances are containers of variables along with the export information. The model grammar can be seen in Figure 7. One interesting top-level form for a linklet body is *define-values*. The Racket Core model essentially models the expressions that a linklet contains in its body. However, defining a variable only makes sense when there's a context around the expression, which is precisely the reason why *define-values* is not a part of the RC language, but a part of Linklets language, since a linklet body acts like a top-level context, where we can define and use new variables.

One other interesting form in this grammar is the *program* form. Recall that the linklets are the most primitive units of compilation. While they're essentially valid objects from the perspective of the compiler itself, which has its own context to define and manipulate linklets, there's not a form in the language to simply act like a context, take some linklets and start the computation. We couldn't use the Racket Core forms like *let-values* to basically define some linklets and instantiate them in the body, because then we would need to make a linklet an expression in the Racket Core language, which is wrong not only from the perspective of the Racket Core, but also from the Linklets' perspective, since it would also mean that a linklet could have another linklet in its body. Therefore, we created the form *program*, that will define some linklets in the *use-linklets* part, and have a *begin*-like body that models a top-level context where we can define some instances using the *let-inst* form and "run" some linklets (i.e. instantiate with targets). Note that the body of a *program* may also have other Racket Core expressions like $(+ \ 1 \ 2)$.

So we have two different evaluation contexts, one for evaluating the *program* body and another for evaluating the linklet body. Hence the evaluation contexts, *EP* and *EL*.

Recall that in the RC language we had the environment Σ and the store σ , mapping variables to cells and cells to values respectively. In the Linklets language we have an additional environment for the linklets and the linklet instances, Ω , that maps symbols to either linklets or linklet instances. The reason to keep an additional environment is

```

L ::= (linklet ((imp-id ...) ...) (exp-id ...) l-top ...)
LI ::= (linklet-instance (exp-id ...) (x C) ...)
p ::= (program (use-linklets (xL L) ...) p-top ... final-expr) | (raises e)
l-top ::= d | e
p-top ::= I | T | (let-inst x I) | IV | (raises e)
d ::= (define-values (x) e)
final-expr ::= n | b | (void) | stuck | IV | T | (raises e)
imp-id ::= x | (x x)
exp-id ::= x | (x x)
I ::= LI | stuck | (instantiate linkl-ref inst-ref ...)
T ::= v | stuck | (instantiate linkl-ref inst-ref ... #:target inst-ref)
IV ::= (instance-variable-value inst-ref x)
linkl-ref ::= x | L | (raises e)
inst-ref ::= x | LI | (raises e)
Ω ::= ((x any) ...)
EP ::= []
      | (program (use-linklets) V ... (let-inst x EL) p-top ... final-expr)
      | (program (use-linklets) V ... EL p-top ... final-expr)
      | (program (use-linklets) V ... EL)
EL ::= []
      | (instantiate EL inst-ref ...)
      | (instantiate L LI ... EL inst-ref ...)

      | (instantiate EL inst-ref ... #:target inst-ref)
      | (instantiate L LI ... EL inst-ref ... #:target inst-ref)

      | (instance-variable-value EL x)

```

Fig. 7. Linklets model grammar

clarity. We're essentially threading the top-level environment through multiple instantiations, so there's no need for the Racket Core model to look through the linklets and instances in the environment when it is trying to look up for a value of variable in an expression. Additionally, currently we keep both linklets and linklet instances in the same Ω environment, so it's not possible to give the same name to both a linklet and a linklet instance (which is presumably not a very good idea in the first place). Although, as a future work, to make the model clearer we plan to either have an additional environment to keep them separate, or have tagged names, or both.

The evaluation in the linklet model happens similarly with the Racket Core model, by repeatedly applying the standard reduction relation. Figure 8 shows the standard reduction relation for the linklet model. One thing to notice is that we have more error cases than actual reductions. The reason for this is the random testing, which we will discuss in detail in Section 1.4.

Aside from the error cases, the ones for the *linklet-lookup*, *let-inst* and the *instance-variable-value* are pretty straightforward. The instantiation happens in *instantiate-linklet* and *eval-linklet* cases, where it begins with calling the meta-function *instantiate-entry*.

The *instantiate-entry* sets the stage for instantiation by collecting the imports, setting the exported but not defined variables to uninitialized, and also if we're instantiating

$[EP[(\text{raises } e)] \Omega \Sigma \sigma] \longrightarrow [(\text{raises } e) \Omega \Sigma \sigma]$	[error]
$[EL[(\text{raises } e)] \Omega \Sigma \sigma] \longrightarrow [(\text{raises } e) \Omega \Sigma \sigma]$	[error in EL]
$[EP[x] \Omega \Sigma \sigma] \longrightarrow [EP[\text{lookup}[\Omega, x]] \Omega \Sigma \sigma]$	[linklet-lookup]
$[EP[(\text{instance-variable-value } LI \ x)] \Omega \Sigma \sigma] \longrightarrow [EP[\text{get-var-val}[LI, x, \sigma]] \Omega \Sigma \sigma]$	[instance variable value]
$[EP[(\text{instance-variable-value } L \ x)] \Omega \Sigma \sigma] \longrightarrow [(\text{raises instance-expected}) \Omega \Sigma \sigma]$	[instance variable value error]
$[EP[(\text{let-inst } x \ LI)] \Omega \Sigma \sigma] \longrightarrow [EP[(\text{void})] \text{extend}[\Omega, (x), (LI)] \Sigma \sigma]$	[let-inst]
$[EP[(\text{instantiate } L \ LI \ \dots)] \Omega \Sigma \sigma] \longrightarrow [EP[LI_1] \Omega_1 \Sigma_1 \sigma_1]$ where $(LI_1 \ \Omega_1 \ \Sigma_1 \ \sigma_1) = \text{instantiate-entry}[\Omega, \Sigma, \sigma, L, LI, \dots]$	[instantiate linklet]
$[EP[(\text{instantiate } L \ LI \ \dots \ \#:\text{target } x_{\text{ao}})] \Omega \Sigma \sigma] \longrightarrow [EP[e_i] \Omega_i \Sigma_i \sigma_i]$ where $(e_i \ \Omega_i \ \Sigma_i \ \sigma_i) = \text{instantiate-entry}[\Omega, \Sigma, \sigma, L, LI, \dots, \#:\text{target}, x_{\text{ao}}]$	[eval linklet]
$[EP[(\text{instantiate } L \ LI \ \dots)] \Omega \Sigma \sigma] \longrightarrow [(\text{raises } e) \Omega \Sigma \sigma]$ where $(\text{raises } e) = \text{instantiate-entry}[\Omega, \Sigma, \sigma, L, LI, \dots]$	[error in instantiation]
$[EP[(\text{instantiate } L \ LI \ \dots \ \#:\text{target } x_{\text{ao}})] \Omega \Sigma \sigma] \longrightarrow [(\text{raises } e) \Omega \Sigma \sigma]$ where $(\text{raises } e) = \text{instantiate-entry}[\Omega, \Sigma, \sigma, L, LI, \dots, \#:\text{target}, x_{\text{ao}}]$	[error in evaluation]

Fig. 8. Linklets model standard reduction relation

with a target then putting the target into the environment as the current target instance to be used in evaluating the linklet body expressions. If no target is given in the entry, then *instantiate-entry* will create an empty instance and inserts it into Ω as the new target of evaluation, as it will be modified by the expressions in the linklet body and returned as the result of instantiation. Additionally if there's an error, such as when less than expected instances are given for the imports, or a given import instance doesn't export the desired variable, then *instantiate-entry* will immediately reduce to *raise*. Otherwise after setting the stage for instantiation, the *instantiate-entry* will pass the control to the *instantiate-loop*, which will start evaluating the expressions in the linklet body.¹

As can be seen in the definition in Figure 9, the *instantiate-loop* uses *rc-api* to pass the expression to the Racket Core model for evaluation, while threading the Ω (the linklet environment), Σ (the lexical environment) and σ (the store) across the evaluations of the expressions in the body, thereby passing all the side effects to variables and the modifications to the target further down the instantiation. The *rc-api* is a slightly modified version of the *run-rc* shown in Figure 6. The *run-rc* evaluates an RC expression and produces only the result. However, since we have effects in the language, while using the RC model from within the linklet model we need to get in addition to the result the environment and the store after evaluating the given expression. That's precisely what *rc-api* does, namely to return a 3-tuple of the result, the environment and the store.

There are two essential points in the instantiation, *i*) the linklet's expressions use of the variables provided by the given target instance, and *ii*) the modifications on the target.

¹ For clarity and space, we will display the codes of only the essential functions and relations in the model. With everything included the model contains over 20 meta-functions. The models, along with the tests can be found in <https://github.com/cderici/linklets-redex-model>

$\text{instantiate-loop}[(\Omega, \Sigma, \sigma, \#:\text{target}, x_T, \#:\text{last-val}, v, \#:\text{result}, \text{instance})]$	$= (\text{lookup}[(\Omega, x_T) \Omega \Sigma \sigma])$
$\text{instantiate-loop}[(\Omega, \Sigma, \sigma, \#:\text{target}, x_T, \#:\text{last-val}, v, \#:\text{result}, \text{value})]$	$= (v \Omega \Sigma \sigma)$
$\text{instantiate-loop}[(\text{define-values } (x_0) e \text{ l-top } \dots), \Omega, \Sigma, \sigma, \#:\text{target}, x_T, \#:\text{last-val}, v, \#:\text{result}, v/i]$	$= \text{instantiate-loop}[(\text{l-top } \dots), \Omega_i, \text{extend}[(\Sigma_i, (x_0), (cell_i))], \text{extend}[(\sigma_2, (cell_i), (v_i))], \#:\text{target}, x_T, \#:\text{last-val}, (\text{void}), \#:\text{result}, v/i]$
$\text{where } (v_i \Sigma_i \sigma_i) = \text{rc-api}[(e \Sigma \sigma)],$ $(\Omega_i \sigma_2) = \text{modify-target}[(\Sigma, \sigma_i, x_0, cell_i, v_i, x_T, \Omega)],$ $cell_i \text{ fresh}$	
$\text{instantiate-loop}[(e \text{ l-top } \dots), \Omega, \Sigma, \sigma, \#:\text{target}, x_T, \#:\text{last-val}, v, \#:\text{result}, v/i]$	$= \text{instantiate-loop}[(\text{l-top } \dots), \Omega, \Sigma_i, \sigma_i, \#:\text{target}, x_T, \#:\text{last-val}, v_i, \#:\text{result}, v/i]$
$\text{where } (v_i \Sigma_i \sigma_i) = \text{rc-api}[(e \Sigma \sigma)]$	
$\text{instantiate-loop}[(e \text{ l-top } \dots), \Omega, \Sigma, \sigma, \#:\text{target}, x_T, \#:\text{last-val}, v, \#:\text{result}, v/i]$	$= (\text{raises } e_i)$
$\text{where } ((\text{raises } e_i) \Sigma_i \sigma_i) = \text{rc-api}[(e \Sigma \sigma)]$	

Fig. 9. Instantiation loop

Ordinarily, when the Racket Code model encounters a variable while evaluating an expression, it looks it up in the top level environment in which we cell and put any “defined” variable in the linklet and signals an error if the variable is undefined. However, as we discussed before, a linklet may export a variable that it doesn’t define and have it in its body as uninitialized, thereby having expressions in its body containing basically a free variable, which is supposed to be bound by the target’s definition during the instantiation. Therefore in this case the Racket Core model, while looking for the variable in the top level environment (and failing), needs to have a pointer to the target instance to look for the variable’s binding. This is essentially the reason why the *instantiate-entry* puts the target instance to the environment before starting the instantiation. So the essential idea is, that if the linklet defines a variable, then it doesn’t matter if the target has a definition for it or not, the linklet will use its own definition, otherwise the target will be consulted for that particular variable.

The second point in the instantiation is, the modification on the target instance by the expressions in the linklet. In both modes of the instantiation (whether any target is provided or not), there’s a target instance used in the instantiation loop (i.e. when evaluating the linklet body), which is either the one that’s given to the entry (i.e. targeted instantiation) or the instance we create ourselves to be returned as a result of instantiation (i.e. regular instantiation). When evaluating the linklet’s expressions, this target’s

variables are modified and/or extended. Note that, this happens only when a linklet is defining a variable with a *define-values* form, as can be seen in Figure 9. Below are the summary of the semantics of these modifications:

For any linklet L , (target) instance T and a variable v that L defines,

1. If T doesn't have a variable named v , then it will be added to T unconditionally.
2. If L exports v , then it will be added to T (see item (4)). If T already has a variable named v , then it will be overwritten.
3. If L doesn't export it, and T has a variable named v , then it will be left unmodified.
4. If both L and T defines and exports v , then it's an error.

Note that in the evaluation of the *define-values* form, the defined variable is added to the top level environment, along with its value. This ensures that for any linklet defined variable v , any other expression in the linklet body that is using v will regularly use the value in the top level environment, instead of the value in the target.

The modifications on the target instance is implemented by passing the target along the function calls as a reference and threading the Ω that contains all the linklets and instances, including the target.

After evaluating all the expressions in the linklet body, as can be seen in Figure 9, a value or the prepared instance is returned, according to the “#:result” parameter, which is set at the beginning by the meta-function *instantiate-entry*. Note that the parameter “#:last-val” is kept during the instantiation, and updated at every expression evaluation, to return the value of the last expression in the linklet body in case that the “#:result” parameter is “value”.

1.4 Testing the Model

Testing is performed in two parts: with individual test cases initially designed for Pycket, and random testing.

Using Pycket Tests

When we were implementing linklets on Pycket, we manually crafted a test suite that has over a 100 individual tests for distinct cases of linklet interaction scenarios. Those tests were running against both Racket and Pycket implementations to ensure the correctness of the semantics.

For the Redex model, we adapted those test cases one-by-one to the model's *program* form, and ran them against three systems at the same time, Racket, Pycket & The Redex model. Happily, all the tests pass.

```

(test-equal
 (term
  (eval-prog
   (program (use-linklets
              [l1 (linklet () (x) (define-values (x) 1))]
              [l2 (linklet ((x)) (y g)
                           (define-values (y) 10)
                           (define-values (g) (lambda (p) (+ x y)))
                           (set! y 50))]
              [l3 (linklet () (y g)
                           (set! y 200)
                           (g -1))])
            (let-inst t1 (linklet-instance ()))
            (let-inst l1 (instantiate l1))
            (instantiate l2 l1 #:target t1) ; fill in the target
            (instantiate l3 #:target t1))) 201)

```

Fig. 10. An example test case

Figure 10 shows one of those test cases as an example, where the linklet l2 exports a closure that has a free variable “y” inside, defined as 10 at the time of closure creation. We start by filling the target t1 with the variables that l2 exports, including the closure “g”, and use the same target t1 in instantiating the linklet l3. The linklet l3 exports the undefined variable “y”, and because that the target t1 now has the variable “y”, it uses the target’s variable. l3 sets the variable to 200 and invokes the closure. Because of the lexical scope, the closure uses the environment that it captures at the time of its creation, which had the value 10 for the variable “y”. However that variable “y” is now modified to be 200, therefore the result is 201, instead of 11. Note that the value for “y” inside the target has been modified too, so *(instance-variable-value t1 'y)* would give us 200.

Random Testing

The PLT Redex provides comprehensive functionalities for random testing, where it generates random terms based on a given grammar and tests those terms against a given predicate. It essentially tries to find a counterexample that will fail the predicate. We used the random testing for both Racket Core and Linklet models separately, where we essentially asked the question “Does Racket agree with the given model about this term?” for both.

In particular, we tested these two predicates, implemented as meta-functions on the languages RC and Linklets respectively:

- *eval-rc=racket-core*
- *eval-prog=racket-linklets*

eval-rc=racket-core is a meta-function that takes a term (in RC language) as input and applies the racket evaluator and the eval-rc (i.e. RC evaluator) at the same time and checks if they produce the same answer (on an empty environment).

One interesting issue was that the *redex-check* generates random terms based on the given grammar, which in this case the grammar for RC language (and also the Linklets grammar for random testing of the Linklets model). However, these languages include objects that doesn't have explicit representations in Racket, such as *closure*. A closure in both Racket and RC has formal parameters, body and an environment that it closes on. However, it's not possible to syntactically produce closures in Racket, since the environment doesn't have explicit representation. Therefore, a syntactically valid term according to the RC grammar such as “(*closure* *x* (+ *x* *y*) ((*y* *cell123*)))” doesn't have a corresponding term in Racket. As a result, any randomly generated term that has an explicit closure in it will raise an exception on the Racket evaluator, while successfully being run on the RC evaluator.

To address this issue, and decrease the number of errors caused by randomly generated such cases, we restricted the grammar a bit for *redex-check* to generate terms that will have corresponding terms in the Racket world. Figure 11 shows the restricted expression for the RC. Note that in this case the only difference is that we don't allow the closures (the non-terminal “c”) to explicitly appear in terms. We have a little more complicated restricted setup for the Linklets, as we will discuss below.

$$\begin{aligned}
 e\text{-test} ::= & x \mid n \mid b \mid (\text{void}) \\
 & \mid (e\text{-test } e\text{-test } \dots) \mid (\text{lambda } (x_i \dots) e\text{-test}) \mid (\text{if } e\text{-test } e\text{-test } e\text{-test}) \\
 & \mid (p2 \ e\text{-test } e\text{-test}) \mid (p1 \ e\text{-test}) \mid (\text{set! } x \ e\text{-test}) \mid (\text{begin } e\text{-test } e\text{-test } \dots) \\
 & \mid (\text{let-values } (((x) \ e\text{-test}) \dots) \ e\text{-test}) \mid (\text{raises } e\text{-test})
 \end{aligned}$$

Fig. 11. Restricted RC expressions

With 1000 randomly generated terms (freshly generated each time we ran the tests), we successfully established that our Racket Core model agrees with the actual Racket implementation.

Testing the Linklets model was a little bit more elaborate than testing the Racket Core model, for two reasons:

1. Because that the terms are randomly generated, in the Linklets model we had to deal with almost every corner case there is to evaluate linklets. Imagine randomly generating Racket modules based on a full Racket grammar and running them. They would be syntactically valid, but one would expect a plethora of runtime errors from reference to undefined identifiers to arity errors in function applications, or type errors like trying to add a number with a string, etc.
2. Terms of the RC language are syntactically valid for Racket language too, so we could easily feed the terms to both Racket and RC evaluators without making any changes. However, this is not possible for the Linklets language, since Racket doesn't know about the *program* form.

```

eval-prog[(program (use-linklets (xL L) ...) p-top ...)] = run-prog[(program (use-linklets (xL L) ...) p-top ...) () () ()]
where (and (term check-free-varss[L, ...])
           (term no-exp/imp-duplicates[L, ...])
           (term no-export-rename-duplicates[L, ...])
           (term no-non-definable-variables[L, ...])
           (term no-duplicate-binding-names[L, ...])
           (term linklet-refs-check-out[
             (p-top ...),
             (xL ...),
             get-defined-instance-ids[(p-top ...), ()])])
eval-prog[p] = stuck

```

Fig. 12. Initial checks before the evaluation begins

To deal with the former case, where we have lots of semantically invalid test inputs, we had two options. First, to use mechanism like *#:prepare* in *redex-check*, to ensure that a generated term is semantically valid, discard otherwise, without applying the predicate. Second, add checks to the Linklets evaluator to catch and produce errors for the corner cases. While the former is a perfectly good solution for the testing per se, we decided to choose the second option to get the model closer to the actual implementation. This, unfortunately left us with the majority of the test cases being rejected by both systems, as opposed to having them being interesting with respect to the instantiation semantics. However, we expect that to be solved when we further improve the testing with the first option as well. As can be seen in Figure 12 below, we have ahead of time checks for a bunch of error cases. The reason for these checks to be done ahead of time instead of the runtime is that the Racket catches these errors in the function *compile-linklet*, where it constructs linklet objects from s-expressions, while the Linklet model clearly doesn't have the *compile-linklet* since the model is syntax-driven, i.e. the representation **is** the object itself.

To address the second issue, where we can not invoke the Racket evaluator on *program* forms, we implemented a basic converter to rewrite the *program* forms to use the Racket forms like *let-values*, etc. Figure 13 below shows such a conversion as an example.

```

> (term (to-actual-racket
      (program (use-linklets
        [l1 (linklet () ())]
        [l2 (linklet ((b)) () (define-values (a) 5) (+ a b)]]
        [l3 (linklet () (b) (define-values (b) 3)]]))
      (let-inst t3 (instantiate l3))
      (let-inst t1 (instantiate l1))
      (instantiate l2 t3 #:target t1)))
'(let ((l1 (compile-linklet '(linklet () ())))
      (l2 (compile-linklet '(linklet ((b)) () (define-values (a) 5) (+ a b)))))
  (l3 (compile-linklet '(linklet () (b) (define-values (b) 3)))))
(define t3 (instantiate-linklet l3 (list)))
(define t1 (instantiate-linklet l1 (list)))
(instantiate-linklet l2 (list t3) t1))

```

Fig. 13. Converting the *program* form into Racket

As discussed above for testing the Racket Core model, another issue with the random testing is generating terms that doesn't have corresponding terms in the Racket language. As we have shown in Figure 11, we solved this issue by restricting the grammar to generate meaningful terms for the Racket evaluator. We also have a restricted grammar for the Linklets language, as shown in Figure 14.

```

linkl-ref-test ::= x | L
p-test ::= (program (use-linklets (xi L) ...) p-top-test ... final-expr-test)
p-top-test ::= T-test | n | b | (void) | stuck | (let-inst x I-test)
I-test ::= (instantiate linkl-ref-test x ...)
T-test ::= (instantiate linkl-ref-test x ... #:target x)
final-expr-test ::= n | b | (void) | stuck | (instance-variable-value x x) | T-test

```

Fig. 14. Restricted Linklet *programs*

We again removed the explicit closures as values and also removed the explicit terms representing the linklet instances. Because a linklet instance is a mapping from variables to cells, and just like in the case of *closure* forms, a cell identifier has no corresponding term in Racket.

Just as the case for Racket Core model, with an increased number of 2000 each time freshly generated random terms, we successfully demonstrated that our Linklets model agrees with the actual Racket linklet implementation (i.e. *racket/linklet*)).

1.5 Conclusion

In this section, we introduced a PLT Redex model for the linklets in Racket, demonstrated how the semantics work, and discussed how we used both individual test cases and random testing to confirm that both the existing Racket implementation and the Pycket implementation of linklets agree with the introduced Linklets PLT Redex model.

References

1. Flatt, M. 2017
<https://groups.google.com/d/msg/racket-dev/2BV3ElyfF8Y/4RSd3XbECAAJ>
2. Racket-on-Chez Status: January 2018
<http://blog.racket-lang.org/2018/01/racket-on-chez-status.html>
3. Linklets and the Core Compiler
<http://docs.racket-lang.org/reference/linklets.html>

2 Adding Pairs To The Denotational Model Of The Untyped CBV Lambda Calculus

Extend Jeremy's denotational model of the untyped lambda calculus with pairs. Extend the proof of correspondence with the operational semantics in Isabelle to handle pairs.

In this section we introduce the changes we made to add the pairs to the denotational model of the untyped lambda calculus, along with the changes for keeping the validity of the soundness and completeness proofs of the denotational semantics with respect to the operational semantics.

The semantics and the proofs we are working with are based on the paper “*Revisiting Elementary Denotational Semantics*”, and the Isabelle proofs are from the proof archive named “*Declarative Semantics for Functional Languages*”, both by Jeremy Siek. [1] [2]

For the sake of clarity and space, we show and discuss here only the essential parts of the changes. A list of changes to the individual Isabelle files are listed in Section 2.4.

All the codes can be found in <https://github.com/cderici/denotational-semantics-LC-plus-pairs>.

$x \in \mathbb{X}$	variables
$n \in \mathbb{Z}$	integers
$\oplus \in \{+, \times, -, \dots\}$	arithmetic operators
$e \in \mathbb{E} ::= n \mid e \oplus e \mid x \mid \lambda x. e \mid e e \mid \text{if } e \text{ then } e \text{ else } e$	
$\langle e, e \rangle \mid \text{car } e \mid \text{cdr } e$	expressions

Fig. 15. Syntax of a call-by-value λ calculus with integer arithmetic, and pairs.

2.1 Extending the Operational & Denotational Semantics

The first task is to add the pairs to the operational semantics. Therefore we start by adding pairs to the grammar of λ calculus, as shown in Figure 15.

The completeness of the denotational semantics wrt operational semantics is proved using the small-step semantics, and the soundness is proved using the big-step semantics. Therefore the next step is to extend both the small-step and the big-step semantics.

We start by extending the small-step semantics with pairs. Firstly, we defined the substitution rules for the pairs (in the function *subst*), and added the rule for pairs to be values.

$$\text{isval } \langle e_1, e_2 \rangle \iff \text{isval } e_1 \wedge \text{isval } e_2$$

After adding inductive cases for pairs to the proof of the substitution lemma (*subst_fv_aux*), which shows that the substitution works, we finally defined the reduction rules for pairs in the standard reduction relation, namely *reduce*, to complete the small-step semantics with pairs.

$$\begin{array}{ll}
\llbracket (\neg \mathbf{isval} \ e_1) \wedge e_1 \longrightarrow e'_1 \rrbracket \Longrightarrow & \langle e_1, e_2 \rangle \longrightarrow \langle e'_1, e_2 \rangle \\
\llbracket (\neg \mathbf{isval} \ e_2) \wedge e_2 \longrightarrow e'_2 \rrbracket \Longrightarrow & \langle e_1, e_2 \rangle \longrightarrow \langle e_1, e'_2 \rangle \\
\llbracket e \longrightarrow e' \rrbracket \Longrightarrow & \mathbf{car} \ e \longrightarrow \mathbf{car} \ e' \\
\llbracket e \longrightarrow e' \rrbracket \Longrightarrow & \mathbf{cdr} \ e \longrightarrow \mathbf{cdr} \ e' \\
\llbracket (\mathbf{isval} \ v_1) \wedge (\mathbf{isval} \ v_2) \rrbracket \Longrightarrow & \mathbf{car} \ (\langle v_1, v_2 \rangle) \longrightarrow v_1 \\
\llbracket (\mathbf{isval} \ v_1) \wedge (\mathbf{isval} \ v_2) \rrbracket \Longrightarrow & \mathbf{cdr} \ (\langle v_1, v_2 \rangle) \longrightarrow v_2
\end{array}$$

For the big-step semantics, we start by adding pairs to the values (*bval*), and defined their big-step evaluation rules, along with the corresponding inductive elimination rules. Note that, $\langle v, v \rangle_{\mathbf{b}}$ denotes a pair value in big-step semantics (**bval** in Isabelle):

$$\begin{array}{ccc}
\frac{\rho \vdash e_1 \Downarrow v_1 \quad \rho \vdash e_2 \Downarrow v_2}{\rho \vdash \langle e_1, e_2 \rangle \Downarrow \langle v_1, v_2 \rangle_{\mathbf{b}}} & \frac{\rho \vdash e \Downarrow \langle v_1, v_2 \rangle_{\mathbf{b}}}{\rho \vdash \mathbf{car} \ e \Downarrow v_1} & \frac{\rho \vdash e \Downarrow \langle v_1, v_2 \rangle_{\mathbf{b}}}{\rho \vdash \mathbf{cdr} \ e \Downarrow v_1}
\end{array}$$

We also added the congruence rules and (rather long) cases for pairs to complete the proof of soundness of the big-step semantics with respect to the small-step semantics, thereby made ready the operational semantics for our work on the denotational model. Before starting to extend the denotational semantics and the correspondence proofs with the pairs, we first need to add the pairs to the value set, and make sure that all the properties and relations defined on them still work.

$$val ::= \mathbf{vnat} \ nat \mid \mathbf{vfun} \ "(val \times val) \ fset" \mid \mathbf{vpair} \ val \ val$$

We add the pairs to the values, as shown above, and extend the denotational semantics as shown in Figure 16. Note that we also extend the definition of the ordering relation \sqsubseteq on the domain \mathbb{D} . As a notation side-note, we will use $\langle val, val \rangle_{\mathbf{v}}$ to denote the pair values, instead of using **vpair**.

We have one more task before we can move on to the soundness and completeness proofs. We need to make sure that the lifting of the ordering relation to the environment still works with the added pairs. For this, we add the cases for pairs in the proof of the *Subsumption* lemma, where we show the ordering relation is downward closed (the Proposition 1 in the paper). Before proving each inductive case for pairs, we stated the introduction and elimination rules and proved for each pair expression, which are shown in Figure 17. After adding these rules, we proved each inductive case for the pairs in the subsumption proof.

$$\begin{array}{l}
\mathcal{E}[\lambda x. e]\rho = \{t \mid \forall (d, d') \in t. d' \in \mathcal{E}[e]\rho(x:=d)\} \\
\mathcal{E}[e_1 \ e_2]\rho = \left\{ d \mid \begin{array}{l} \exists t d_1 d'_1 d_2. t \in \mathcal{E}[e_1]\rho \wedge d_2 \in \mathcal{E}[e_2]\rho \\ \wedge (d_1, d'_1) \in t \wedge d_1 \sqsubseteq d_2 \wedge d \sqsubseteq d'_1 \end{array} \right\} \\
\mathcal{E}[x]\rho = \{d \mid d \sqsubseteq \rho(x)\} \\
\mathcal{E}[n]\rho = \{n\} \\
\mathcal{E}[e_1 \oplus e_2]\rho = \{n_1 \oplus n_2 \mid n_1 \in \mathcal{E}[e_1]\rho \wedge n_2 \in \mathcal{E}[e_2]\rho\} \\
\mathcal{E}[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\rho = \left\{ d \mid \begin{array}{l} \exists n. n \in \mathcal{E}[e_1]\rho \wedge (n \neq 0 \implies d \in \mathcal{E}[e_2]\rho) \\ \wedge (n = 0 \implies d \in \mathcal{E}[e_3]\rho) \end{array} \right\} \\
\mathcal{E}[\langle e_1, e_2 \rangle]\rho = \{p \mid \exists v_1 v_2. v_1 \in \mathcal{E}[e_1]\rho \wedge v_2 \in \mathcal{E}[e_2]\rho \wedge p = \langle v_1, v_2 \rangle_v\} \\
\mathcal{E}[\text{car } e]\rho = \{v_1. \exists v_2. \langle v_1, v_2 \rangle_v \in \mathcal{E}[e]\rho\} \\
\mathcal{E}[\text{cdr } e]\rho = \{v_2. \exists v_1. \langle v_1, v_2 \rangle_v \in \mathcal{E}[e]\rho\} \\
\\
n \sqsubseteq n \quad \frac{t \sqsubseteq t'}{t \sqsubseteq t'} \quad \frac{v_1 \sqsubseteq v'_1 \wedge v_2 \sqsubseteq v'_2}{\langle v_1, v_2 \rangle_v \sqsubseteq \langle v'_1, v'_2 \rangle_v} \quad \boxed{d \sqsubseteq d}
\end{array}$$

Fig. 16. A new elementary semantics for CBV λ -calculus.

2.2 Soundness of the Denotational Semantics wrt. the Operational Semantics

We're finally ready to extend the proof of soundness (*DenotSoundFSet.thy*). We first extend and prove for pairs the preservation and progress lemmas, which state “*reduction preserves denotation*”, and “*if there is a denotation for an expression, then it's either a value, or there exists an expression we can reduce to*”, respectively. Recall that in the soundness proof the big-step operational semantics is used. Therefore these two lemmas are not used in the soundness proof (i.e. remove them, and the soundness still works), however they are quite essential to the completeness proof (Lemma 7 & 8 in the paper), as it uses the small-step semantics.

We believe that the most important modification in this step is to extend the logical relation \mathcal{G} that relates \mathbb{D} to the sets of syntactic values \mathbb{V} (i.e. **isval**).

$$\begin{array}{l}
\mathcal{G} : \mathbb{D} \rightarrow \mathcal{P}(\mathbb{V}) \\
\mathcal{G}(n) = \{n\} \\
\mathcal{G}(t) = \left\{ \langle \lambda x. e, \varrho \rangle \mid \begin{array}{l} \forall (d_1, d_2) \in t. \forall v_1. v_1 \in \mathcal{G}(d_1) \implies \\ \exists v_2. \varrho(x:=v_1) \vdash e \Rightarrow v_2 \wedge v_2 \in \mathcal{G}(d_2) \end{array} \right\} \\
\mathcal{G}(\langle v_1, v_2 \rangle_v) = \left\{ vp \mid \begin{array}{l} \exists bv_1, bv_2. bv_1 \in \mathcal{G}(v_1) \wedge \\ bv_2 \in \mathcal{G}(v_2) \wedge vp = \langle bv_1, bv_2 \rangle_b \end{array} \right\}
\end{array}$$

With this addition, we were able to prove the Lemma 2 in the paper (with the addition of inductive cases for pairs), which states “ \mathcal{G} is downward closed”. Then we moved on to proving the pair cases for the termination argument (the Lemma 4 in the paper), and from that the soundness proof easily followed.

$$\begin{aligned}
\mathbf{cons}[intro] &= \llbracket v_1 \in \mathcal{E}[e_1]\rho \wedge v_2 \in \mathcal{E}[e_2]\rho \wedge v = \langle v_1, v_2 \rangle_{\mathbf{v}} \rrbracket \implies v \in \mathcal{E}[\langle e_1, e_2 \rangle]\rho \\
\mathbf{car}[intro] &= \llbracket (\langle v_1, v_2 \rangle_{\mathbf{v}}) \in \mathcal{E}[e]\rho \wedge v = v_1 \rrbracket \implies v \in \mathcal{E}[\mathbf{car} \ e]\rho \\
\mathbf{cdr}[intro] &= \llbracket (\langle v_1, v_2 \rangle_{\mathbf{v}}) \in \mathcal{E}[e]\rho \wedge v = v_2 \rrbracket \implies v \in \mathcal{E}[\mathbf{cdr} \ e]\rho \\
\mathbf{cons}[elim] &= \llbracket v \in \mathcal{E}[\langle e_1, e_2 \rangle]\rho \\
&\quad \bigwedge v_1 v_2. \llbracket v_1 \in \mathcal{E}[e_1]\rho \wedge v_2 \in \mathcal{E}[e_2]\rho \wedge v \sqsubseteq \langle v_1, v_2 \rangle_{\mathbf{v}} \rrbracket \implies P \rrbracket \implies P \\
\mathbf{car}[elim] &= \llbracket v \in \mathcal{E}[\mathbf{car} \ e]\rho \\
&\quad \bigwedge v_1 v_2. \llbracket \langle v_1, v_2 \rangle_{\mathbf{v}} \in \mathcal{E}[e]\rho \wedge v \sqsubseteq v_1 \rrbracket \implies P \rrbracket \implies P \\
\mathbf{cdr}[elim] &= \llbracket v \in \mathcal{E}[\mathbf{cdr} \ e]\rho \\
&\quad \bigwedge v_1 v_2. \llbracket \langle v_1, v_2 \rangle_{\mathbf{v}} \in \mathcal{E}[e]\rho \wedge v \sqsubseteq v_2 \rrbracket \implies P \rrbracket \implies P
\end{aligned}$$

Fig. 17. Introduction and elimination rules for the $\langle e, e \rangle$, **car** and **cdr** expressions

2.3 Completeness of the Denotational Semantics wrt. the Operational Semantics

Our final task is to make the completeness proof work with the pairs. The proof starts with showing that “*the reverse substitution preserves meaning*”, which involves an extension to the definition of the *join* operation defined on \mathbb{D} .

$$n \sqcup n = n \quad t \sqcup t' = t \cup t' \quad \frac{v_1 \sqcup v'_1 = v''_1 \quad \wedge \quad v_2 \sqcup v'_2 = v''_2}{\langle v_1, v_2 \rangle_{\mathbf{v}} \sqcup \langle v'_1, v'_2 \rangle_{\mathbf{v}} = \langle v''_1, v''_2 \rangle_{\mathbf{v}}} \boxed{d \sqcup d}$$

Note that, *join* is now inductively defined, and the join on pairs is only defined when both of the pairwise joins are defined, both of which reflect in the further proofs involving the \sqcup operation. Two important such proofs are *Proposition 1* that says “*join is the least upper bound of \sqsubseteq* ”, and the *Lemma 1* that says “ *\mathbb{D} is closed under join on values*”. For the first one, we worked on two lemmas:

- *le_union1* : $v_1 \sqcup v_2 = v_{12} \implies v_1 \sqsubseteq v_{12}$
- *le_union2* : $v_1 \sqcup v_2 = v_{12} \implies v_2 \sqsubseteq v_{12}$

We turned the proof of *le_union1* lemma into an inductive proof and added the cases for pairs. For the proof of *le_union2*, we just added the lemma “ $v_1 \sqcup v_2 = v_2 \sqcup v_1$ ” (*join_commutes*), and the proof trivially followed the proof of *le_union1*.

The final tasks of the completeness proof are extending the lemmas “*reverse substitution preserves meaning*” (Lemma 5 in the paper), and the “*reverse reduction preserves meaning*” (Lemma 6 in the paper). Recall that the final goal in completeness proof is to show that $e \longrightarrow^* n \implies \mathcal{E}[e]\emptyset = \mathcal{E}[n]\emptyset$. So the reverse substitution comes from decomposing the equality in $e \longrightarrow e' \implies \mathcal{E}[e]\rho = \mathcal{E}[e']\rho$ into $\mathcal{E}[e]\rho \subseteq \mathcal{E}[e']\rho$ and $\mathcal{E}[e']\rho \subseteq \mathcal{E}[e]\rho$. The forward direction is the preservation, and the backwards direction is still the base of the reverse substitution after adding the pairs, since the backwards direction itself is straightforward to prove for pairs by induction. However, the proof

of these two lemmas themselves required us to prove the inductive cases for pairs, so we extended those proofs with pairs too, and the proof of completeness with pairs then trivially followed.

2.4 List Of Files Changed

Here are the list of Isabelle theory files that are changed (in the order of bottom to top in the import hierarchy), along with a small description on what exactly changed.

- *Lambda.thy* : The grammar is extended with the pairs, and the pair cases are added to the functions *FV* and *BV*.
- *SmallStepLam.thy* : Added pairs to the *subst*, *is_val* and *reduce*. Also added the pair cases to the proof of the lemma *subst_fv_aux*.
- *BigStepLam.thy* : Extended the *bval* and *eval*, and added rules for pairs to *bs_val*, *psubst*. Also stated and proved the congruence rules for pairs as lemmas, and added the cases for pairs to the lemma *big_small_step*, along with a couple of small additions like *bs_observe*.
- *ValuesFSet.thy* & *ValuesFSetProps.thy* : Extended the *val* definition, and also the relation *val_le* (\sqsubseteq). Added the inductive elimination rules for pairs for the \sqsubseteq relation.
- *DeclSemAsDenotFSet.thy* : Extended the definition of the function *E* with appropriate cases for pairs.
- *ChangeEnv.thy* : Added the introduction and elimination rules for pairs, and extended the proof of *change_env_le*.
- *DenotSoundFSet.thy* : Added rules for the *substitution*, *preservation* and the *progress* lemmas. Extended the logical relation *good*, and the *sub_good* lemma (the Lemma 2, \mathcal{G} is downward closed). Finally extended the *denot_terminates* lemma.
- *DenotCompleteFSet.thy* : Extended the definition of *join*, and added pairs to the proofs of the lemmas *combine_values*, *le_union1* and *le_union2* (stated and proved an additional lemma, namely *join_commutes*). Finally, extended the proof of *reverse_subst_pres_denot* lemma.

References

1. Siek, J., Revisiting Elementary Denotational Semantics, 2017
<https://arxiv.org/abs/1707.03762>
2. Siek, J., Declarative Semantics for Functional Languages, Archive of Formal Proofs, 2017
https://www.isa-afp.org/entries/Decl_Sem_Fun_PL.html

3 Measuring the Performance of the Linklet System On Pycket

Measure the performance of your changes to the Pycket JIT on the existing benchmarks. For at least 3 performance differences, propose a hypothesis that explains the difference.

In this section, we start by demonstrating and discussing the differences between the old and the new Pycket utilizing the linklets, and move on to measuring the performance of the new Pycket on the existing benchmarks, and finally we talk about the behaviors we observe in the experiment results by explaining some of the performance issues that are caused by the new linklet implementation on Pycket.

3.1 A Step Back : Old Pycket vs New Pycket

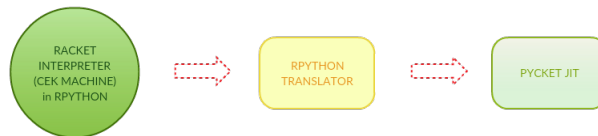


Fig. 18. Pycket

Pycket is a tracing JIT compiler that is generated by the RPython meta-tracing framework (originally designed for PyPy) that automatically generates tracing JIT compilers from interpreters written in RPython. [2][3] Before moving on to the new Pycket version that utilizes the linklets, let's recall how the Pycket works in its original design. As shown in the Figure 19, given a Racket program source, Pycket runs the Racket binary to expand the given program into Racket core forms (see fully expanded programs), and then uses its own expander (*expand.py*) to turn fully-expanded Racket code into a Pycket AST and evaluates it. [1]

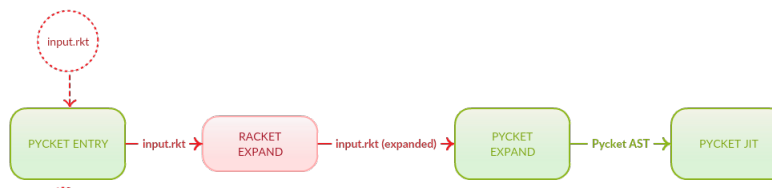
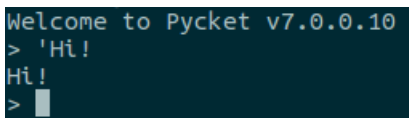


Fig. 19. Old Pycket

As we discussed a bit in the Section 1.1, a couple of fundamental changes have happened in the Racket's core that made the Racket developers to consider moving onto a new runtime, namely the Chez Scheme. To make this transition easier (and correspondingly to make Racket more portable) Racket implements a layer called the “*rumble*”, which gives to any runtime that implements it the capability of hosting other layers that are written in Racket. [2] The way that it works is, that Racket exports four layers in the form of serialized independent linklets that are used to bootstrap Racket using the *rumble* implementation of the runtime system. These *bootstrapping* linklets implement the *expander*, *thread*, *io*, and *regex* layers. Therefore, any VM that implements *rumble* can import these linklets and get the Racket's module system, macro system and many more for free.

Having most of the *rumble* layer already implemented within, implementing the linklets enabled the Pycket to exploit this change in the Racket and use the exported bootstrapping linklets. Currently, the new version of Pycket loads the “*expander*” linklet into its runtime at boot, thereby gets (among others) the entire module system, the functions *read*, *expand* and *eval* for free. It uses Racket's own module system to *load*, *expand* and *eval* Racket modules (e.g. *racket/base*), consequently running entirely independent from the Racket binary. With this setup, Pycket even has a jitted Racket REPL, as shown in Figure 20 below.



```
Welcome to Pycket v7.0.0.10
> 'Hi!
Hi!
> 
```

Fig. 20. A jitted Racket REPL on Pycket

3.2 Performance

The new implementation with the linklets fundamentally changed how the evaluation works in Pycket. In the old version, Pycket used to handle resolving, expanding and evaluating the modules by itself. The new Pycket, on the other hand, uses Racket's module system through the *expander* linklet. Therefore on the high level, the entire execution is in the continuation of the evaluation (instantiation) of the expander linklet. In the low level, because that the expander represents everything in terms of linklets, the only way to start an evaluation is by instantiating a linklet as we discussed in Section 1.2,

Although the use of linklets provided us with lots of cool Racket features for free, such as REPL, it came with the cost of losing most of the control in evaluating core Racket library modules in general. Pycket has to blindly follow the implementation provided in the bootstrap linklets. This is good from the perspective of performance

comparison between Pycket and Racket, since both systems evaluating exactly the same implementation of the module system, macro system, and all the others that are exported with the bootstrap linklet. However, since Pycket doesn't have any static access to the Racket codes, it is hard (if not impossible) to communicate hints to the JIT for the execution of any Racket function that we received from the exported linklet (e.g. using decorators like `@jit.unroll_safe`). Because of this, and because the JIT is automatically generated by the RPython framework, it is not a trivial task to customize and specialize the JIT specifically for Racket modules implemented on the linklets at the high level (e.g. the macro system or the module system).

Because of the implementation differences between the old Pycket and the new Pycket, and also the developments on Racket, currently the old Pycket is not able to work with the latest version of Racket (7.0.0.10), and the new Pycket is not able to work with the old Racket (6.12) that the old Pycket was working on. Therefore, in this section, we're going to separately introduce with our benchmarks

- the difference between the old Racket (6.12) and the new Racket (7.0.0.10), and
- the difference between the old Pycket and the new Pycket.

to identify interesting points where the new Pycket might got slower against the old Pycket, while the new Racket got faster against the old one.

Setup We're basing our experiments on Pycket's 2015 ICFP paper [1], in which the old Pycket was being compared to the Racket 6.1. Here we're using Racket 6.12, because that some of the developments on Racket between 6.1 and 6.12 and the corresponding developments on Pycket rendered the old Pycket unable to run on 6.1.

We use the *Larceny Cross-platform Benchmarks*, each in `#lang racket/base`. We performed the experiments on IU's Karst cluster, where each node is an IBM NeXtScale nx360 M4 server equipped with two Intel Xeon E5-2650 v2 8-core processors, each node having 32 GB of RAM, running Red Hat Enterprise Linux (RHEL) 6 with kernel 2.6.32-754.2.1.el6.x86_64.

Every benchmark was run 100 times uninterrupted at highest priority in a new process. The execution time was measured in the program and, hence, does not include start-up (neither loading and expanding the *racket/base*); however, warm-up was not separated, so all times include JIT compilation. Analysis is performed using the ReBench's analysis tool.²

All of the benchmark sources, including the script generators for the Karst cluster can be found in the <https://github.com/cderici/pycket-performance>

² <https://github.com/smarr/ReBench>

3.3 Experiment Results

We start this section by showing the results for the old Pycket vs Racket 6.12, as a sanity check. Since the Pycket’s ICFP paper runs Pycket against the Racket 6.1, we wanted to see if there is a major change in Pycket’s performance between Racket 6.1 and 6.12. As can be seen in Figure 21, nothing major has changed, as the results are pretty much aligned with what we see in the paper. [1]

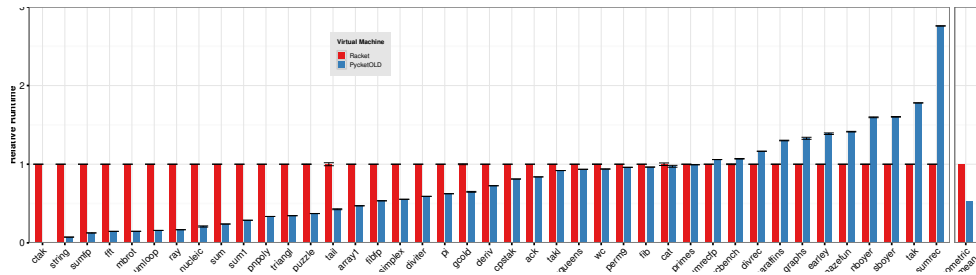


Fig. 21. Old Pycket vs Racket 6.12 - normalized to Racket, lower is better

Next, we show the results of Racket 6.12 vs. Racket 7.0.0.10 in Figure 22 below. The first thing to notice is, while for almost all the other benchmarks the performance is more or less the same, for the benchmark *sum1*, we observe a 3x slowdown in the new Racket. This is because that the *sum1* benchmark calls *read* at each iteration, and in Racket 6.12 the *read* is implemented in C, while in Racket 7.0.0.10 it is in Racket. All the other benchmarks are below the point of being an interesting difference between the two Rackets. However, while we evaluate Pyckets’ performances we use the information here in a binary fashion, where we look for benchmarks for which the Racket 7.0.0.10 got (with any difference) faster and the new Pycket got slower.

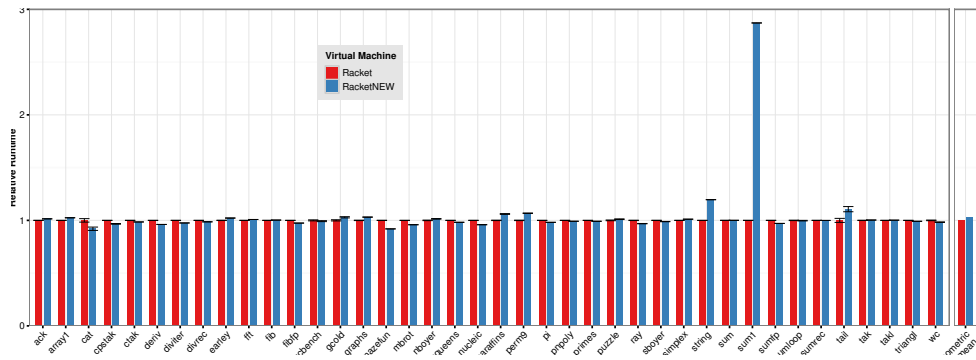
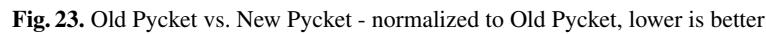


Fig. 22. Racket 6.12 vs. Racket 7.0.0.10 - normalized to Racket 6.12, lower is better

After an examination of the results, we identify *mazefun*, *deriv*, *nqueens*, *nucleic*, *sumrec* and *triangl* as our cases of study, where the new Racket got faster, but the new Pycket got slower.



To evaluate the results, for all the benchmarks we extracted the JIT back-end logs and looked at individual loops that are traced during the execution of the benchmarks for both Pyckets, focusing on the several benchmarks we identified before as interesting.

³ Note that the timing of the benchmarks are done in the program itself, so they don't include the time spent on neither loading the Racket library modules nor the expansion of the benchmark being run.

	Old Pycket	New Pycket
Tracing time:	0.262688	28.287384
Backend time:	0.188901	5.581448
Total # of loops:	43	893
Total # of bridges:	250	6985
Total # of guards:	21401	3603638

Table 1. Differences in JIT summaries between Old Pycket and New Pycket for the *mazefun* benchmark

run. The get around this issue and identify the interesting loop, for each benchmark we generated the JIT back-end counts that includes the information on how many times a trace has been used. Looking at the mostly used trace for each benchmark showed us the interesting loop we are interested in.

Looking at the individual traces for all the benchmarks and identifying the interesting loops, we observed that the traces are mostly the same in both versions of Pycket, with some minor but essential differences that made the difference in performance results. This enabled us to clearly identify the additional overheads caused by the linklet implementation, and form some hypotheses on the behavior of our benchmarks.

Recall that, as we discussed in the Redex model in Section 1.3, the interpreter needs to have a handle to the current target instance to lookup for any variable that couldn't be found in the top level environment (i.e. not defined by the linklet). For this reason, we put the current linklet (target) instance to the environment to communicate it with the interpreter. Also recall that in the model (as well as in the Pycket implementation) we put Cells in the environment instead of just the values, in other words we put a pointer to the linklet variable. That's because any modifications on the variable (i.e. via a set! or target overwrite) need to be reflected on any instance that has that variable (as well as, for example, any closure that captured an environment that includes the variable).

Another interesting point is, in Racket (and Chez Scheme) implementation of linklets, the variables have *constance* information that's being used in various optimizations. Although Pycket's linklet variables have the *constance* information, we use it only as a check when a linklet variable is being modified. If a "constant" linklet variable is being modified, we signal an error.

These points about keeping the "current linklet instance" in the environment, using cells, and the linklet variables' constance play a major role in the overheads we observe in the traces.

3.5 Hypotheses for the Current Performance Overheads

The Figures 24 and 25 below demonstrate a one to one difference between typical traces we observe in the old and new Pycket across all benchmarks. The example is a cleared and simplified loop traced for the *sumrec* benchmark. In the Figure 25, additional lines within the inner loop are marked with ++. Here are the main observations and hypotheses to explain the current performance behaviors of Pycket with linklets, based on the individual traces we examined for almost every benchmark:

One observation that's visible in all the traces is, because of the additional *current_linklet_instance* field in our environment, we have an additional 8 bits (pointer) on each environment cell we create.

As discussed above, there are two ways to lookup a variable, *i*) top level environment, and *ii*) the *current_linklet_instance*. This indirection has a couple of effects on the traces:

- Whenever a variable is got from the *current_linklet_instance*, JIT has to put a guard to check if the instance and the value is received non-null. This is visible in all of the benchmarks.
- The inner loops have additional parameters for the *current_linklet_instance* value and (for some unclear reason) the top level environment.
- For any variable that's passed to the inner loop which is originally received from the *current_linklet_instance*, there's a guard within the inner loop to check whether the variable is null. Because of a currently unclear reason, the optimizer is not able to optimize away those guards, leaving the loop with an extra check in each iteration. We hypothesize that this is the primary reason for the benchmarks that have a lot of indirections in them, such as *mazefun*, since we observe many interconnected inner loops with additional parameters being checked at each iteration.
- Because of the extra field in the environment for the *current_linklet_instance*, whenever we create an environment cell there is an extra *setfield_gc* operation.

The handling of the linklet variables also causes some negative effects that are clearly visible on the traces. First of all, as we discussed above, because that we're not communicating the *constance* information with the JIT, the optimizer is not able to remove the guards that are clearly removed from the corresponding trace for the old Pycket. This is observed on the traces for *nqueens*, *deriv*, *sumrec*, *triangl* benchmarks.

The combination of these issues/hypotheses are visible in all the benchmark traces, and enough to explain the performance behaviors, as these are the *only* differences between the traces of the two versions of Pycket. Aside from these observations the most used traces of both systems are identical.

References

1. S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. *Pycket: A tracing JIT for a functional language*. In Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. ACM, 22–34, 2015.

2. C. F. Bolz, A. Cuni, M. Fijakowski, and A. Rigo. *Tracing the meta-level: PyPy's tracing JIT compiler*. In Proc. ICPOOLPS, pages 18–25, 2009.
3. C. F. Bolz and L. Tratt. *The impact of meta-tracing on VM design and implementation*. Science of Computer Programming, 2013.

```

Outer Loop
    i11 = getfield_gc_i(p0, ConsEnvSize1Fixed.inst_vals_fixed_0)
Inner Loop
    1. label(i11, p3, p0, p1)
    2. guard_not_invalidated()
    3. i18 = int_lt(i11, 0)
termination check
    4. guard_false(i18)
    5. i20 = int_sub(i11, 1)
construct a new environment cell
    6. p21 = new_with_vtable()
    7. p22 = new_with_vtable()
    8. setfield_gc(p21, i11)
    9. setfield_gc(p22, p3)
    10. setfield_gc(p21, ConstPtr(ptr24))
    11. setfield_gc(p22, p0)
    12. setfield_gc(p22, p1)
    13. setfield_gc(p22, ConstPtr(ptr25))
jump back to loop header
    14. jump(i20, p3, p21, p22)

```

Fig. 24. Optimized trace for sumrec inner loop on old Pycket

```

Outer Loop
  p11 = getfield_gc_r(p1, Cont.inst_env 16 pure>)
  guard_class(p11)
  p30 = getfield_gc_r(p11, ConsEnv.inst_current_linklet_instance)
  guard_nonnull(p30)
  p31 = getfield_gc_r(ConstPtr(ptr29), W_Cell.inst_w_value)
  guard_nonnull_class(p31, W_CellIntegerStrategy)
  i11 = getfield_gc_i(p31)
  guard_value(i11)
Inner Loop
  1. label(p11, p30, i11, p3, p0, p1)
  2. guard_not_invalidated()
  3. i18 = int_lt(i11, 0)
  ++ guard_value(i11)
termination check
  4. guard_false(i18)
  5. i20 = int_sub(i11, 1)
construct a new environment cell
  6. p21 = new_with_vtable()
  7. p22 = new_with_vtable()
  8. setfield_gc(p21, i11)
  9. setfield_gc(p22, p3)
  ++ setfield_gc(p21, ConstPtr(ptr30))
  10. setfield_gc(p21, ConstPtr(ptr24))
  11. setfield_gc(p22, p0)
  12. setfield_gc(p22, p1)
  13. setfield_gc(p22, ConstPtr(ptr25))
jump back to loop header
  14. jump(p11, p30, i20, p3, p0, p1)

```

Fig. 25. Optimized trace for `sumrec` inner loop on new Pycket