Macro-embedding Compiler Intermediate Languages in Racket

WILLIAM J. BOWMAN, University of British Columbia, Canada

We present the design and implementation of a macro-embedding of a family of compiler intermediate languages, from a Scheme-like language to x86-64, into Racket. This embedding is used as part of a testing framework for a compilers course to derive interpreters for all the intermediate languages. The embedding implements features including safe, functional abstractions as well as unsafe assembly features, and the interactions between the two at various intermediate stages.

This paper aims to demonstrate language-oriented techniques and abstractions for implementing (1) a large family of languages and (2) interoperability between low- and high-level languages. The primary strength of this approach is the high degree of code reuse and interoperability compared to implementing each interpreter separately. The design emphasizes modularity and compositionality of an open set of language features by local macro expansion into a single host language, rather than implementing a language pre-defined by a closed set of features. This enables reuse from both the host language (Racket) and between intermediate languages, and enables interoperability between high- and low-level features, simplifying development of the intermediate language semantics. It also facilitates extending or redefining individual language features in intermediate languages, and exposing multiple interfaces to the embedded languages.

1 INTRODUCTION

Recently, we faced an engineering challenge: we wanted to design and implement interpreters for a family of 126 languages. These languages share considerable structure and features sets, and many are nearly idententical; changes to one language are not independent of any other. Some are high level, functional, safe languages, while some are low level, unsafe and equivalent to a subset of x86-64 assembly (x64). Most are somewhere in between—intermediate languages in which safe, high-level features interoperable with unsafe, low-level features. The language definitions may change as the specification evolves, so code maintence is a primary concern.

Our users could use these interpreters to test an optimizing nanopass compiler (Ghuloum 2006; Keep and Dybvig 2013) from a small Scheme-like language to x64. At the time, the users relied on a combination of fragile syntactic tests and end-to-end tests of the entire compiler. But syntactic tests would often yield false test failure when the compiler produced an equivalent but synctically different program, and end-to-end tests failed to support debugging since a test failure could not be attributed to a particular compiler pass. Interpreters that provide the ground truth value of a program for each compiler pass would enable resiliant tests and better debugging. The users should not need to know anything about the interpreters other than their interface: they take a (possibly valid) program, and produce its value or an error. They would also need debugging output and metrics such as reasonable error messages and the ability to distinguish whether the provided program was invalid or the test failed because the value produced was unexpected.

Our context is the University of British Columbia's undergraduate compilers course (CPSC 411).[1] Our users are the students and the course staff, who use the interpreters to test the compilers and provide feedback. The nanopass structure of the compiler means the compiler includes many passes, up to about 30 by the end of the course, and each pass has separate intermediate languages specifying the source and target of the pass. Students develop the compiler incremenetally in 10 milestones, with new languages and changes to existing languages each milestone.

[1]For additional context, the course material is publically available online at https://github.com/cpsc411/, with the most recent spring 2022 iteration deployed at https://www.students.cs.ubc.ca/~cs-411/2021w2/.

Author's address: William J. Bowman, Computer Science, University of British Columbia, Vancouver, British Columbia, Canada, wjb@williamjbowman.com.

Our first question is an engineering question: how should we implement these interpreters? The constraits described above prevent us from doing the obvious thing: writing each interpreter as a function that folds over the syntax of the language and computes the value of the input program. Language-oriented programming (Felleisen et al. 2018; Felleisen et al. 2015; Tobin-Hochstadt et al. 2011) could provide a suitable design for answering our first question, *if* the existing abstractions apply to low-level languages as well as high-level languages. But this yields a second question: (how) can we implement a family of compiler intermediate languages, from low- to high-level languages, as a macro-embedding using language-oriented programming? This paper answers this latter question, demonstrating how to embed such low-level abstractions using language-oriented programming, particularly in Racket, so that we can easily develop and maintain our family of interpreters.

Before considering our case study, let's consider the small example language below.

$$
\begin{array}{rcl}
p & ::= & (\textbf{begin}\ \textit{effect effect} \ldots) \\
\textit{effect} & ::= & (\textbf{set!}\ \textit{reg word64}) \mid (\textbf{set!}\ \textit{reg}_1\ (\textit{binop reg}_1\ \textit{word32})) \\
\textit{reg} & ::= & \textbf{rax} \mid \ldots \mid \textbf{r12} \mid \textbf{r13} \mid \textbf{r14} \mid \textbf{r15} \\
\textit{word32} & ::= & \textit{int32} \mid \textit{reg} \\
\textit{word64} & ::= & \textit{word32} \mid \textit{int64} \\
\textit{binop} & ::= & + \mid - \mid *
\end{array}
$$

The language is a simple subset of x64 with an s-expression syntax, described by the grammar above. A program begins with **begin** and is followed by a non-empty list of effects. Each effect can modify a register, either setting it to a value or to the result of a binary operation. We assume every program ends with **rax** set to the result.

We could implement a whole-program environment-passing interpreter as below.

```
> (define (interp-x64-v1 p)
    (define (interp-binop op)
      (match op
        ['+ +]
        ['- -]
        ['* *]))

    (define (interp-opand opand env)
      (match opand
        [(? integer?) opand]
        [(? symbol?) (dict-ref env opand)]))

    (define (interp-effect effect env)
      (match effect
        [`(set! ,reg (,binop ,reg ,opand))
         (dict-set env reg ((interp-binop binop)
                            (interp-opand reg env)
                            (interp-opand opand env)))]
        [`(set! ,reg ,opand)
         (dict-set env reg (interp-opand opand env))]))

    (match p
      [`(begin ,list-of-effect ...)
       (interp-opand 'rax (for/fold ([env empty])
                                    ([effect list-of-effect])
                            (interp-effect effect env)))]))      #lang racket
```

This interpreter is a fold over the list of effects, which ends by dereferencing the value of the register 'rax from the final environment. Initially, the environment is empty. Each effect is interpretered by updating the environment, which maps registers to values.

```
> (interp-x64-v1 '(begin (set! rax 15)))
15                                                              #lang racket
```

The interpreter is straightforward to implement, once, for a single small language. But we need a different approach to scale to developing and maintaining hundreds of interpreters. Many of the intermediate languages include registers, assignment statements to registers, integer operands, etc—we do not want to copy/paste, but must design for sharing the implementation of features between interpreters.

The key change we make to our design is to macro-embed each *language feature* into a common host language, rather than implementing each *language* separately. We then derive an interpreter for any given language using the host language's eval, with the set of languages features imported.

For the above example language, we could start by implementing the set! feature as follows.

```
> (define env (make-hash))
> (define-syntax (set! stx)
    (syntax-parse stx
      [(set! reg (binop _ opand))
       #`(dict-set! env 'reg (binop (interp-opand 'reg)
                                    (interp-opand opand)))]
      [(set! reg opand)
       #`(dict-set! env 'reg (interp-opand opand))]))           #lang racket
```

In this example, we define as a macro the set! language feature, without the rest of the language. We first define the environment as a mutable hash table at run time in the host language. Since the syntax of the program is traversed implicitly in the macro expander, we cannot pass additional arguments during the syntax traversal, so we use a mutable environment. The macro set!, defined using define-syntax, is implicitly called during compile time by the macro expander whenever it encounters the defined identifier set!. Each macro transforms syntax into new syntax at compile time, so we require slightly different abstractions to work over syntax objects rather than over quoted lists. The macro's definition receives as input a single syntax object, representing the entire syntax of the call to the macro, including its own name. We use syntax-parse (Culpepper 2012; Culpepper and Felleisen 2010) for pattern matching over syntax objects, and use quasisyntax (written #`) to construct syntax objects. We destructure the input syntax expecting set! in operator position (which is unused in the definition), a register on the left-hand side, and either a binary operation or an operand on the right-hand side. The syntax template constructed with #` can refer to bound syntax pattern variables, whose bound names are automatically replaced by their value in the template without the need to unquote.

The definition of set! is essentially similar to the earlier interpreter, implementing assignment as an update to the environment. The key difference is that we generate a run-time expression that updates the environment, rather than immediately executing the update to the environment. We must explicitly transform the register operand from a raw piece of syntax into a symbol by quoting it in the generated syntax. But we do not need to explicitly transform binary operations such as +, since they are already defined and have sufficiently similar semantics in the host language.

The implementation relies on interp-opand, defined below, to expand operands.

```
> (define-syntax (interp-opand stx)
    (syntax-parse stx
      [(_ reg:id)
       #`(dict-ref env 'reg)]
      [(_ int:integer)
       #`int]))                                       #lang racket
```

interp-opand has two cases: the pattern reg:id matches a raw identifier and binds the syntax pattern variable reg, while the pattern int:integer matches an integer literal and binds the syntax pattern variable int. Identifiers are assumed to be registers and expand to dereference from the environment. The register is quoted, transforming the raw identifier into a symbol. Integers expand to themselves since the object language and the host language share integer literal syntax. Following a common convention, we use _ to bind the irrelevant name of the operator (interp-opand) in the patterns.

To get a full interpreter from this feature, we need to combine it with the other features (begin, +, etc), and provide an entry point that provides the value interpretation of programs by dereferencing 'rax from the environment as the final operation. We can reuse the host language begin and binary operations, as their semantics are close enough, and add the following entry point.

```
> (define-syntax (interp-x64-v2 stx)
    (syntax-parse stx
      [(_ stx)
       #`(begin stx (dict-ref env 'rax))]))
> (interp-x64-v2
   (begin
     (set! rax 15)))
15                                                    #lang racket
```

Our entry point, interp-x64-v2, is a macro that uses the host language begin to sequentially execute the original syntax and then dereference the symbol 'rax from the environment. The macro expander proceeds outside-in, so it will expand all the macros in stx after expanding a call to interp-x64-v2. Unlike when we used the procedure interp-x64-v1, we do not need to explicitly quote the input program (begin (set! rax 15)) since the expander syntax-quotes the syntax of the call automatically before passing that syntax object to the macro.

This is the basic idea behind language-oriented design (Felleisen et al. 2018; Felleisen et al. 2015; Tobin-Hochstadt et al. 2011). Rather than writing the language as a closed set of features, a language is a collection of individual features open to extension. We implement the features by embedding into a common host language, then we can use the module system to export, import, extend, redefine and mix the features into a desired language. If we're careful about how we design the embedding, we can easily support interoperability and extension. The above implementation is not particularly careful. We completely redefine set!, so we're unable to interoperate with Racket's set!, and registers are only embedded as part of set!, not as their own feature, so do not work correctly in other contexts. We explore a better implementation in the rest of the paper.

Language-oriented programming in Racket has been used to implement general-purpose languages such as Typed Racket (Tobin-Hochstadt and Felleisen 2008, 2011) and domain-specific languages such as Scribble (Flatt et al. 2009). However, these canonical examples are high-level languages.

This approach is not obviously suited to macro-embedding assembly language, with its global state, labeled code, and unstructured control flow. However, Racket's language-extension facilities—namely the module system, the static interposition points, run-time interposition features, macros designed for extension, and `eval` parameterized by a namespace—support this use case and our engineering constraints well. All interpreters share a single ~600 LOC file that implements the embedding. The embedding is largely modular, so individual language *features* can be changed once, locally, for all languages, with some exceptions. Embedding most high-level features is straightforward by borrowing from Racket. Embedding into a shared host language simplifies implementing intermediate languages, where low- and high-level features blend together.

This language-oriented approach has added benefits beyond achieving our engineering goals. It simplifies using the intermediate languages in Racket's toolchain (such as the REPL and IDE) and enables interoperating with Racket (useful for in-class demonstrations and typesetting notes). Trying to define macros locally and compositionally—instead of, *e.g.*, analyzing, transforming, or executing a whole program—exposed some bad design decisions in our intermediate languages, enabling us to improve the languages.

In this article, we walk through the design and implementation of embedding key interesting language features, and of the interfaces to our interpreters. We describe which parts of the Racket language-extension API are particularly useful, and discuss limitations in our current design, implementation, and Racket abstractions.

The implementation of our embedding can be installed locally using the following command.[2]

```
raco pkg install https://github.com/cpsc411/cpsc411-pub.git?path=cpsc411-lib#hashlang-x64
```

We typeset all interactive examples run in a REPL in a grey box, with a language name indicating in which language the examples must be run. Typeset code that does not appear in a box is an excerpt from the implementation of the languages, is not expected to run on its own, and may be simplified to elide irrelevant details. Most of these excerpts are from a single file, found online at https://github.com/cpsc411/cpsc411-pub/blob/hashlang-x64/cpsc411-lib/cpsc411/langs/base.rkt.

2 X64

We start with the embedding of the lowest-level features from x64, in particular, global state (registers and memory), pointer arithmetic, and labels and jumps. These features are present not only in the target language, but in many of the intermediate languages.

The syntax of our target language, a parenthesized subset of x64, is given below.

$$
\begin{array}{rcl}
\mathit{p} & ::= & (\textbf{begin}\ \mathit{effect}\ \ldots) \\
\mathit{effect} & ::= & (\textbf{set!}\ \mathit{addr}\ \mathit{word32}) \mid (\textbf{set!}\ \mathit{reg}\ \mathit{word64}) \mid (\textbf{set!}\ \mathit{reg}_1\ (\mathit{binop}\ \mathit{reg}_1\ \mathit{word32})) \\
 & \mid & (\textbf{set!}\ \mathit{reg}_1\ (\mathit{binop}\ \mathit{reg}_1\ \mathit{addr})) \mid (\textbf{with-label}\ \mathit{label}\ \mathit{effect}) \mid (\textbf{jump}\ \mathit{trg}) \\
 & \mid & (\textbf{compare}\ \mathit{reg}\ \mathit{opand}) \mid (\textbf{jump-if}\ \mathit{relop}\ \mathit{label}) \\
\mathit{reg} & ::= & \textbf{rsp} \mid \textbf{rbp} \mid \textbf{rax} \mid \ldots \mid \textbf{r12} \mid \textbf{r13} \mid \textbf{r14} \mid \textbf{r15} \\
\mathit{addr} & ::= & (\mathit{reg}\ +\ \mathit{int32}) \mid (\mathit{reg}\ +\ \mathit{reg}) \mid (\mathit{rbp}\ -\ \mathit{int32}) \\
\mathit{word32} & ::= & \mathit{int32} \mid \mathit{label} \mid \mathit{reg} \\
\mathit{word64} & ::= & \mathit{word32} \mid \mathit{int64} \mid \mathit{addr} \\
\mathit{opand} & ::= & \mathit{int64} \mid \mathit{reg} \\
\mathit{trg} & ::= & \mathit{reg} \mid \mathit{label}
\end{array}
$$

[2]The software is archived at https://doi.org/10.5281/zenodo.7051910.

Every program is a sequence of statements, which have a one-to-one correspondence with some x64 instruction. Labels must be symbols whose string representation is accepted by the assembler. The displacement- and index-mode offsets in addresses must be divisible by 8, as we assume all addresses are word aligned. We use the small-code model, so labels are 32-bit words. The register operand of a binary operation must be the same on the right-hand side and the left-hand side, to suit x64, but we duplicate the operand in our syntax to simplify the interpretation of `set!` expressions as a Racket `set!`.

Our run-time system follows various conventions from the System V ABI[3], which we introduce as they become relevant. The program must also end by jumping to the special done label with the result in the register `rax`.

We can write everyone's favourite function, factorial, in `#lang cpsc411/hashlangs/base` (the language that exposes the majority of the macro embedding on which all the interpreters are based) as follows.

```
> (begin
    (set! r15 5)
    (set! r14 1)
    (with-label fact
      (compare r15 0))
    (jump-if = end)
    (set! r14 (* r14 r15))
    (set! r15 (+ r15 -1))
    (jump fact)
    (with-label end
      (set! rax r14))
    (jump done))
120                                       #lang cpsc411/hashlangs/base
```

Programs in this `#lang` are embedded into Racket, so we can interoperate at run time, although these interoperability semantics are not well documented nor the primary goal.

```
> (require (only-in racket/base displayln define))
> (displayln rax)
120
> (define (fact x)
    (begin
      (set! r15 x)
      (set! r14 1)
      (with-label fact
        (compare r15 0))
      (jump-if = end)
      (set! r14 (* r14 r15))
      (set! r15 (+ r15 -1))
      (jump fact)
      (with-label end
        (set! rax r14))
      (jump done)))
```

[3] https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf

```
> (fact 6)
720                                                  #lang cpsc411/hashlangs/base
```

2.1 Registers

Registers should be modelled as global variables, provided by the language. However, Racket does not allow a variable provided by a module to be modified by another module using `set!`, so we cannot define registers as in `(define rax (void))`. We could define the registers in the user's module by non-hygienically introducing them in `#%module-begin` (the module-level interposition point), but breaking hygiene is complicated and fragile so should be avoided, and introducing module-level state that is supposed to be global doesn't allow interoperation with Racket.

Instead, we define each register as a `box`. The `box` is exported, and the contents of the `box` is modified instead of the variable itself. Normally, a user creates a box as in `(define x (box 0))`, assigns to a box using `(set-box! x 1)`, and dereferences a box explicitly using `(unbox x)`. To embed registers as boxes, we want to transparently transform any use of `set!` on a register, and any use of a register, into one of these operations.

Thankfully, Racket's `set!` was designed with extension in mind. It cooperates with any identifier that is defined as a variable-like transformer using `make-variable-like-transformer`. Such macros have two definitions attached: one that is used when they appear on their own (like a reference to a variable), and one that is used by `set!` during macro expansion when the variable-like macro appears on the left-hand side of a `set!`. For example, the definition of the register `rax` is given below.

```
(define _rax (box (void)))

(define-syntax rax
  (make-variable-like-transformer
   #`(unbox _rax)
   (lambda (stx)
     (syntax-parse stx
       [(set! _ v)
        #`(set-box! _rax v)]))))
```

For each register, we create two definitions: a run-time representation, and a compile-time transformer for references of and assignments to the surface syntax name for the register. The run-time representation is prefixed by an underscore, and is defined as a `box`, initialized either to `(void)` or a value define by the run-time system. The compile-time transformer uses `make-variable-like-transformer` to transform references, using the first argument, and assignments, using the second argument.

When the defined syntax, `rax` in this case, appears as a reference on its own and not in operator position, the transformer unconditionally expands to the first argument, in this case generating the syntax object representing the run-time expression `(unbox _rax)`.

When the syntax appears as the left-hand side of a `set!`, the expander delegates to the second argument of `make-variable-like-transformer` to expand the entire `set!` expression. We know that the syntax of the expression must be `(set! rax v)` for this transformer to be called, and we only need the syntax of `v`. We rewrite the syntax `(set! rax v)` to the syntax object representing the run-time expression `(set-box! _rax v)`.

Now, all references to `rax` as a variable implicitly expand to `(unbox _rax)`, and all instances of `(set! rax v)` expand to `(set-box! _rax v)`. Further, the embedding of this register is local—we only define the language feature `rax`, and did not need to modify any existing definitions.

make-variable-like-transformer is not necessary to implement registers, but is a useful abstraction of the pattern, and is already designed to cooperate with set!. We could alternatively redefine set!, as we did in Section 1, but using variable-like transformers instead means we can implement the register as its own feature, separate from set!, isolating the embedding of the register rather than spreading it across two definitions (the register as a reference, and a separate extension of set!).

2.2 Memory and Pointer Operations

To model memory, we create two vectors: stack and memory. Similar to registers, we must extend the behaviour of set! to support setting memory addresses, and define the interpretation of dereferencing addresses. Unfortunately, since addresses of the form (rbp - offset) are compound rather than identifiers, we cannot use make-variable-like-transformer. This suggests the need for a new abstraction, which we return to in Section 6.

We start with the stack, since it has more structure. The stack pointer is required to live in rbp[4], and this register is only allowed to be used in a particular, stack-like manner, specified by the compiler. rbp can be decremented by a word-aligned integer-literal offset. Its value can be read directly, or a value can be read from it at a word-aligned offset. The stack pointer starts at the very end of the stack, and stack allocation should grow backwards towards the front.

Like with other registers, we create a box _rbp to hold the current stack pointer. However, unlike the other registers, we define rbp as a custom macro, rather than as a variable-like transformer.

```
(define-syntax (rbp stx)
  (syntax-parse stx
    [:id
     #'(unbox _rbp)]
    [(rbp (~datum -) offset:integer)
     #`(vector-ref stack (- (unbox _rbp) offset))]))
```

The first clause only matches when rbp appears by itself as an identifier, not in operator position, and binds no pattern variable. The second clause matches rbp in operator position, followed by a displacement-mode offset, indicated by the literal symbol - and an integer literal. Referencing rbp in non-operator position produces the current stack pointer, but the syntax (rbp - offset) (when not used as the left-hand side of a set!) dereferences the address at the current stack pointer minus the offset.

```
> (require (only-in racket/base vector-length))
> (vector-length stack)
12960
> (rbp - 0)
'uninit
> rbp
12959
> (rbp + 0)
eval:10:0: rbp: expected the literal symbol `-'
  at: +
  in: (rbp + 0)
> (rbp - rax)
eval:11:0: rbp: expected integer
```

[4]Actually, it holds the frame base pointer, but we haven't introduced the frame yet, so they're the same for now.

```
    at: rax
    in: (rbp - rax)
                                              #lang cpsc411/hashlangs/base
```

When dereferencing the address at `(rbp - 0)`, we get an uninitialized value (since we never set that address). But when referencing `rbp`, we get the address of the end of the stack. Other syntax for `rbp` is invalid. We have not spent much effort on error reporting yet, but `syntax-parse` occassionally makes good error reporting easy.

This macro is only used when the address appears as a value; we implement assigning to the stack separately. The implementation for assignment to the stack is entirely in `set!`, which must detect what kind of left-hand side is being used, and expand depending on the kind of address.

There are three cases in `set!` for stack addresses: the stack pointer is incremented, the stack pointer is updated, or some stack address is assigned.

```
(define-syntax (set! stx)
  (syntax-parse stx
    ; Stack pointer increment
    [(set! (~literal rbp) (binop (~literal rbp) v))
     #`(set-box! _rbp (binop rbp v))]
    ; Stack pointer update
    [(set! (~literal rbp) v)
     #`(set-box! _rbp v)]
    ; Stack slot update
    [(set! ((~literal rbp) - offset) v2)
     #`(vector-set! stack (- (unbox _rbp) offset) v2)]
    ....))
```

This definition uses the syntax class `~literal` to match only when the left-hand side is or includes the bound identifier `rbp`; these clauses are only triggered when `set!` is used with the stack pointer, not an arbitrary other register. The first two cases only use `rbp` as the left-hand side, so could be supported by `make-variable-like-transformer`. These clauses expand to update `rbp`, modifying the current stack pointer. The third clause uses a compound left-hand side. Since we need this compound left-hand side (as well as the earlier compound dereferences), we cannot use `make-variable-like-transformer`, and must spread our implementation of `rbp` between `set!` and a custom tranformer for `rbp`. The final clause expands to modify a location on the stack, computed as the current stack pointer minus the offset.

```
> (set! (rbp - 0) 5)
> (rbp - 0)
5
                                              #lang cpsc411/hashlangs/base
```

To implement the heap, we use another vector, called `memory`. Unlike the stack, memory addresses can end up in any register, so we cannot implement the register as its own transformer or special case any particular register like we did with `rbp`. Instead, we must detect the general form of the address operands in the implementation of `set!`. There are two cases to consider: assigning to memory (the left-hand side has the form `(reg binop offset)`), or reading from memory (the right-hand side has the form `(reg binop offset)`). In each case, we generate an instruction using either `vector-set!` or `vector-ref` accessing `memory` at the location base plus or minus the offset.

```
(begin-for-syntax
  (define-syntax-class addr-op
    (pattern (~or (~datum +) (~datum -)))))
  (define-syntax-class not-rbp
    (pattern (~not (~literal rbp)))))

(define-syntax (set! stx)
  (syntax-parse stx
    ....
    ; Assign to memory
    [(set! (base:not-rbp op:addr-op offset) value)
     #`(vector-set! memory (op base offset) value)]
    ; Read from memory
    [(set! v1:id (base:not-rbp op:addr-op offset))
     #`(r:set! v1 (vector-ref memory (op base offset)))]
    ; Else use Racket's set!
    [(set! v1 v2)
     #`(r:set! v1 v2)]))
```

We use two syntax classes, which extend the pattern language of `syntax-parse`, to detect the index-mode operand. The base must not be the identifier bound to the stack pointer register, and there should be an address operator (either the literal syntax `+` or `-`) between the base and the offset. We reimport Racket primitives that get redefined by our embedding with the prefix `r:`, so `r:set!` refers to Racket's implementation of `set!`.

```
> (vector-length memory)
10000
> (vector-ref memory 0)
'unalloced
> (set! (r12 + 0) 5)
> (set! rax (r12 + 0))
> rax
5
> (vector-ref memory 0)
5                                                   #lang cpsc411/hashlangs/base
```

By default, our run-time system puts a pointer to unallocated heap memory in `r12`, which the compiler uses to implement allocation.

2.3 Labels and Jumps

We model labelled instructions as procedures with no arguments that are immediately called, and jumps as procedure calls that must not return.

All statements appear under a `begin`. To implement labels, we redefine `begin` to transform the sequence of instructions into a `letrec`, binding each label to a procedure.

This definition is the first we've seen that uses syntax quasiquotation with unquoting. The syntax unquote `#,` splices into a syntax template a value computed by running a compile-time expression. This works analogously to unquote `,` over quasiquoted lists `` ` ``. Similarly, syntax splicing unquote `#,@` splices a list of values into the template.

```
  (begin-for-syntax
    (define (labelify defs effects)
      (match effects
        ['() (values defs '())]
        [(cons effect effects)
         (syntax-parse effect
           #:literals (with-label begin)
           [(with-label label effect)
            (let-values ([(defs effects^)
                          (labelify defs (cons #'effect effects))])
              (values
               (cons #`[label (lambda () (r:begin #,@effects^))] defs)
               (list #`(jump label))))]
           [(begin effects1 ...)
            (labelify defs (append (attribute effects1) effects))]
           [effect
            (let-values ([(defs effects^) (labelify defs effects)])
              (values defs #`(effect #,@effects^)))])]))

  (define-syntax (begin stx)
    (let-values ([(defs effects) (labelify '() (cdr (syntax->list stx)))])
      #`(letrec #,defs (r:begin #,@effects))))
```

The macro `begin` does not immediately pattern match on its input, but deligates to a compile-time procedure `labelify`. This procedure takes the list of statement as syntax objects (created by converting the syntax object to a list of syntax objects using `syntax->list`), without the leading `begin` operator (dropped by calling `cdr`). The procedure traverses the list of instructions `(cons effect effects)`, collecting a list of definitions to bind with `letrec` and a list of instructions to execute in the body of the `letrec`. When `effect` is a labeled instruction `(with-label label effect^)`, we create a binding for a procedure definition `` #`[label (lambda () (begin effects^ ...))] ``, where `effects^` are the instructions that remain after recursively labeling the rest of the list of effect, starting with the labelled one, `(cons #'effect^ effects)`. We prepend the new definitions to those returned by the recursive call, and replace the instruction sequence with a jump to the new definition ``(list #`(jump label))``. When we encounter a nested `begin`, we prepend all the instructions in the body to the current `effects` and recur. All other instructions are left in place. At the end of the loop, we bind all the labels in the scope of the list of instructions, expanding to (roughly) ``#`(letrec (defs ...) (r:begin effects^ ...))``.

To embed conditional jumps, we create a mutable `eq?`-comparable hash table mapping each comparison procedure to a flag. The `compare` instruction is modelled as a procedure, which updates each flag using each comparison procedure, and `jump-if` jumps when the comparison procedure's flag is set.

```
  (define flags
    (make-hasheq
     `((,!= . #f) (,= . #f) (,< . #f) (,<= . #f) (,> . #f) (,>= . #f))))

  (define (compare v1 v2)
    (for-each (lambda (cmp) (hash-set! flags cmp (cmp v1 v2)))
              (list != = < <= > >=)))

  (define (jump-if cmp-proc d) (when (hash-ref flags cmp-proc) (d)))
```

This implementation uses each procedure pointer as a key in the table. This can be fragile, as the pointer for a procedure can be unpredictable, but this local use should not be a problem. We could avoid the fragility with extra indirection, using symbols as keys in the flag table and mapping those symbols to the correct procedure, or use `eval`. However, in either case, `jump-if` would need to be a macro that quotes ones of its arguments. The other solutions also introduce more code and indirection to reason about.

The implementation of jumps assumes that jumps never return. To ensure they never return, we rely on the program always explicitly exiting by jumping to `done`, as required by the run-time system. To implement `done`, we store an escape continuation at the start of the module, and `done` invokes that continuation, ending the whole computation with the value of `rax`. The return-address register is initialized to `done`, so intermediate languages that implement procedures that follow the calling convention also exit properly.

```
(define exit-cont (box (lambda _ (r:error "halt not initialized"))))

(define (done) ((unbox exit-cont) (unbox _rax)))

(define-syntax (boundary stx)
  (syntax-parse stx
    [(_ term)
     #`(let/ec k
         (set-box! exit-cont k)
         term)]))
```

The `boundary` form must be installed around any term that uses labels and jumps. The `boundary` is implicitly installed around terms at the module level. We must instead use `let/cc` around interactive terms via `#%top-interaction` (the interposition point that lets our language be used from Racket's REPL) since the continuation lives past the local scope during REPL interactions. The `boundary` form is also exposed to enable Racket interoperability. Without an explicit boundary, terms may not return to Racket properly. The `boundary` is similar to the interoperability semantics by Patterson et al. (2017) enabling inline typed assembly in a typed functional language. Their semantics enable the value of the return register to flow to the function language when the `halt` instruction is delimited by an explicit boundary term.

```
> (define (f1 x)
    (begin (set! rax x) (jump done))
    (displayln "After boundary")
    rax)
> (f1 5)
5
> (define (f2 x)
    (boundary
      (begin (set! rax x) (jump done)))
    (displayln "After boundary")
    rax)
> (f2 5)
After boundary
5                                                   #lang cpsc411/hashlangs/base
```

The boundary term delimits the halt continuation used for the run-time system. A better implementation might use delimited countinuations explicitly.

3 INTERMEDIATE LANGUAGE ABSTRACTIONS AND METADATA

While interpreting the low-level abstractions is one major challenge, another major challenge occurs at the various intermediate language points where source-like abstractions must interoperate with target-like abstractions. Two examples that make interesting use of Racket's macro system are abstract locations and frame variables. We provide a grammar for one of the intermediate languages that includes these features.

$$
\begin{array}{rcl}
\mathit{p} & ::= & (\textbf{module}\ \mathit{info}\ (\textbf{define}\ \mathit{label}\ \mathit{info}\ \mathit{tail})\ \ldots\ \mathit{tail}) \\
\mathit{info} & ::= & ((\textbf{assignment}\ ((\mathit{aloc}\ \mathit{rloc})\ \ldots))) \\
\mathit{frame} & ::= & (\mathit{aloc}\ \ldots) \\
\mathit{tail} & ::= & (\textbf{jump}\ \mathit{trg}\ \mathit{loc}\ \ldots) \mid (\textbf{begin}\ \mathit{effect}\ \ldots\ \mathit{tail}) \\
\mathit{effect} & ::= & (\textbf{set!}\ \mathit{loc}\ \mathit{triv}) \mid (\textbf{set!}\ \mathit{loc}_1\ (\mathit{binop}\ \mathit{loc}_1\ \mathit{opand})) \mid (\textbf{begin}\ \mathit{effect}\ \ldots\ \mathit{effect}) \\
\mathit{opand} & ::= & \mathit{int64} \mid \mathit{loc} \\
\mathit{triv} & ::= & \mathit{opand} \mid \mathit{label} \\
\mathit{loc} & ::= & \mathit{rloc} \mid \mathit{aloc} \\
\mathit{rloc} & ::= & \mathit{reg} \mid \mathit{fvar} \\
\mathit{trg} & ::= & \mathit{label} \mid \mathit{loc}
\end{array}
$$

In this intermediate language, the **module** and procedure definitions are annotated with metadata *info* describing where abstract locations have been assigned so far. The **assignment** is a map from abstract locations to physical locations, which is updated in stages during register allocation as variables are assigned either to register or frame locations, depending on various factors.

Prior to register allocation, each language uses abstract locations *aloc*, which are variables uniquely (within the scope of a procedure) identified by their name, but whose physical location is handled by the compiler. Abstract locations look like an arbitrary variable name followed by a "." and a numeric suffix, such as `x.1` in `(set! x.1 5)`. In the intermediate languages *during* (as it takes several nanopasses to complete) and just *after* register allocation, some or all abstract locations should be aliases for physical locations. However, prior to register allocation and when not yet assigned, abstract locations must act like undeclared local variables. Being able to run programs in these intermediate states between register allocation nanopasses enables tests to fail early.

In many intermediate languages, we abstract away from the stack *per se*, and expose frame variables. Frame variables allow most of the compiler to avoid some tedium when dealing with stack locations. A frame variable is an identifier named "fv" with a numeric suffix, such as the `fv1` in `(set! fv1 5)`. Frame variables are allocated in the enclosing procedure's stack frame in the word-sized slot indicated by the suffix. For example, `fv1` is roughly equivalent to `(rbp - 8)`. However, the exact location on the stack depends on the details of how procedures and frames are implemented; `rbp` normally points to the base of the enclosing procedure's stack frame, but changes during a non-tail call to point to the callee's frame. The intermediate languages must map frame variables to stack locations, so that explicit stack manipulation implicitly manipulates frame variables and procedure calling conventions are interpreted correctly.

For abstract locations, we can use `make-variable-like-transformer` if the abstract location has been assigned a physical location. However, the macro must be locally bound, not globally bound, so we use `let-syntax` instead of `define-syntax`. If an abstract location has not been assigned a physical location, we must also bind it in a local scope. We use a local analysis implemented with `local-expand` (which enables a macro to locally change expansion order) to detect abstract locations that appear in `set!` and bind them as local variables during expansion.[5] We can implement frame variables using `make-variable-like-transformer`, although we must break hygiene to get them globally bound. However, we must also keep track of some uses of the frame base pointer `rbp` to provide a consistent interpretation of frame variables, so this implementation must interoperate at run time with assignments to `rbp`.

3.1 Abstract Locations

By default, Racket does not let us assign to an undefined variable, so `(set! x.1 5)` is invalid if `x.1` is not in scope.

```
> (set! x.1 5)
set!: assignment disallowed;
 cannot set variable before its definition
  variable: x.1
  in module: top-level
> (define x.1 (void))
> (set! x.1 5)                                                    #lang racket
```

We could tell Racket to do it anyway, but the scope of the assigned variable may be unexpected.

```
> (compile-allow-set!-undefined #t)
> y.2
y.2: undefined;
 cannot reference an identifier before its definition
  in module: top-level
> (set! y.2 5)
> y.2
5
> (let () (set! z.3 5))
> z.3
5                                                                 #lang racket
```

Abstract locations can be freely generated by the compiler and have no defined or sequential pattern, so we cannot create them ahead of time like we do registers and frame variables. Abstract locations are not global variables; they are unique to a procedure's scope. Two procedures can refer to `x.1`, and those are different variables. As we see in the grammar, programs do not declare what abstract locations are in scope. So allowing `set!` to undefined variables would not implementat abstract locations correctly.

We must first bind abstract locations. We want to avoid writing an explicit traversal over the entire program. This would require either fixing the syntax, which we don't want to do since we want to reuse this solution across many languages, or writing an abstract tree traversal, which is silly since we already have a macro expander. We implement an analysis by locally changing expansion order to fully expand an expression that references abstract locations, and instrumenting `set!` to collect the set of abstract locations that appear on the left-hand side during expansion. This is sufficient since programs must be well defined—an abstract location must be assigned before it is referenced.

[5]This analysis would be unnecessary if the intermediate language declared local variables, suggesting a better language design.

The analysis form do-bind-locals is defined below, accompanied by some compile-time definitions for collecting a scope-aware set of identifiers.

```
(begin-for-syntax
  (require syntax/id-set racket/set)
  (define gathered-locals (mutable-free-id-set))

  (define (reset-locals!)
    (set-clear! gathered-locals))

  (define (collect-local! id)
    (set-add! gathered-locals id))

  (define (get-locals)
    (set->list gathered-locals)))

(define-syntax (do-bind-locals stx)
  (syntax-parse stx
    [(_ body except ...)
     (reset-locals!)
     (define b (local-expand #'body 'expression '()))
     #`(let (#,@(for/list ([l (get-locals)]
                           #:unless (set-member? (immutable-free-id-set
                                                   (attribute except))
                                                  l))
                  #`[#,(syntax-local-introduce (format-id #f "~a" l)) (void)]))
         #,b)]))
```

We wrap the body of any scope-introducing form, particularly **define** and **module**, with do-bind-locals. The possibly empty sequence #'(except ...) declares a set of variables that should not be bound, such as abstract locations that have been assigned physical locations. We first fully expand the body using local-expand. During this local expansion, all assigned abstract locations are collected into the set gathered-locals. Each is bound within the scope of the fully expanded body b, using format-id with #f to clear the original scope from the identifier, and using syntax-local-introduce to bind it in the current scope.

To collect the abstract locations, we make a modification to set!.

```
(define-syntax (set! stx)
  (syntax-parse stx
    ; Read from memory
    [(set! v1:id (base:not-rbp op:addr-op offset))
    ; Collect used abstract location
     (when (aloc? (syntax->datum #'v1))
       (collect-local! #'v1))
     #`(r:set! v1 (vector-ref memory (op base offset)))]
    ; Else use Racket's set!
    [(set! v1 v2)
    ; Collect used abstract location
     (when (aloc? (syntax->datum #'v1))
       (collect-local! #'v1))
     #`(r:set! v1 v2)] ....))
```

Abstract locations are written to either when reading from memory, or when the set! does not involve a memory address at all. In both cases, if the left-hand side is an abstract location, we collect it.

For an abstract location that is already assigned a physical location, we exclude it from binding in do-bind-locals and instead make it a variable-like transformer to its assigned physical location. For interpreting programs, it isn't necessary to redirect abstract locations to their assigned physical location. However, doing so aids debugging by triggering failing tests earlier. If a bug occurs in one of the register allocation nanopasses, the interpretation of the program changes as soon as the metadata changes, and not only once the program text has been rewritten to replace all abstract locations with their assigned physical locations.

To implement these assigned abstract locations, we use the following compile-time procedures. make-aloc-transformer is an abstraction of make-variable-like-transformer for abstract locations that have been assigned the physical location rloc, and bind-assignments generates bindings based on the metadata info for the program tail.

```
(begin-for-syntax
  (define (make-aloc-transformer rloc)
    (make-variable-like-transformer
     rloc
     (lambda (stx)
       (syntax-parse stx
         [(set! bla v)
          #`(r:set! #,rloc v)]))))

  (define (bind-assignments info tail)
    #`(let-syntax #,(for/list ([assignments (dict-ref info 'assignment '())])
                      (with-syntax ([aloc (car assignments)]
                                    [rloc (cadr assignments)])
                        #`[aloc (make-aloc-transformer #'rloc)]))
        #,tail)))
```

bind-assignments is unusal in that it generates syntax with new macro definitions, *i.e.*, with new compile-time code that must be further expanded. We use let-syntax to bind each abstract location as a local macro. This is important since the same abstract location can appear in multiple scopes. Being defined as a macro, reads and writes to the abstract location are statically rewritten to the physical location that has been assigned. The call to make-aloc-transformer appears as syntax, not as a value spliced into the syntax. We generate a call to the compile-time procedure instead of inlining the make-variable-like-transformer call in the syntax template to avoid code duplication. If t were inlined in the syntax template, the code would be duplicated during expansion time, allocating a new closure for each binding, and causing performance problems during macro expansion.

This helper is used in the implementations of scope-introducing forms with attached metadata, such as in the following.

```
(define-syntax (module stx)
  (syntax-parse stx
    [(_ info defs ... body)
```

```
      (define info-dict (infostx->dict #'info))
      #`(boundary
         (begin
           #,(bind-assignments
              info-dict
              #`(local [defs ...]
                  (do-bind-locals tail
                                 #,@(get-info-bound-vars info-dict))))))]))
```

We see the implicit insertion of the boundary mentioned earlier. First bind-assignments binds any abstract locations that have been assigned, then do-bind-locals locally expands the body and binds all other abstract locations. This order doesn't matter, since do-bind-locals excludes binding based on the info metadata.

3.2 Frame Variables

The intermediate languages contain an unbounded number of global frame variables corresponding to locations on the current procedure's stack frame. Frame variables have the form "fv*n*", where *n* is some natural number.

The first challenge is binding all these variables. We cannot bind an unbounded number of variables apriori. For the moment, we cheat: we bind an arbitrary large number, 1620, of them.[6] We discuss alternatives in Section 6.

To bind them, we break hygiene, using syntax-local-introduce to bind frame variables in the right scope, as we don't want to list out all 1620 of them in the interpreter. We bind each frame variable to a macro that performs the offset calculation into the stack.

```
(begin-for-syntax (define CURRENT-FVARS 1620))

(define-syntax (define-fvars! stx)
  (syntax-parse stx
    [(_)
     #`(begin
         #,@(for/list ([i (in-range 0 CURRENT-FVARS)])
              (with-syntax ([fvar (syntax-local-introduce
                                   (format-id #f "fv~a" i))]
                            [offset (* i 8)])
                #`(define-syntax fvar (make-fvar-transformer offset)))))]))

(define-fvars!)
```

The define-fvars! form generates frame variables 0 through 1619, and binds each to a make-fvar-transformer (introduced shortly) at the offset corresponding to its index into the frame. We use format-id with #f, indicating no scope, to create the identifier then use syntax-local-introduce to prevent the macro from introducing a fresh scope. (We thought that providing stx as the lexical context on the identifier would have avoided the use of syntax-local-introduce, but that doesn't seem to be the case.) As before, we define and call make-fvar-transformer rather than inline it in the syntax template.

[6]We would have choosen 42, but needed more than about 50, as students did use about 50 frame variables in some test programs. This number is similar to 42, but large enough.

make-fvar-transformer is a wrapper around make-variable-like-transformer that performs the offset calculation into the stack, and otherwise behaves like rbp.

```
(begin-for-syntax
  (define (make-fvar-transformer frame-offset)
    (with-syntax ([stack-index #`(- (unbox _rbp) (+ (unbox current-fvar-offset)
                                                    #,frame-offset))])
      (make-variable-like-transformer
       #`(vector-ref stack stack-index)
       (lambda (stx)
         (syntax-parse stx
           [(set! _ v) #`(vector-set! stack stack-index v)]))))))
```

The offset for a frame variable is in two parts: the offset given by the statically known index on the frame variable, plus a dynamic offset determined by the run-time modifications to the frame base pointer rbp. The expression (unbox current-fvar-offset) occurs under the syntax template, generating a run-time calculation, while frame-offset is an compile-time value spliced into the template. In the intermediate languages, rbp is modified by a constant offset when a new frame is allocated for a non-tail call, moving the frame base pointer to point to the callee's frame. But a frame variable might be referenced, after this, as an argument for the call, and must refer to the caller's frame. The stack-index calculates the offset into the stack in this situation, using the value current-fvar-offset which is updated whenever rbp is incremented.

As an example, consider the following low-level implementation of a non-tail call.

```
> (define identity (lambda () (begin (set! rax fv0) (jump r15))))
> (begin
    (set! fv0 41)
    (set! fv1 1)
    (set! rbp (- rbp 16))
    (return-point l
      (begin
        (set! r15 l)
        (set! fv2 fv0)
        (jump identity)))
    (set! rbp (+ rbp 16))
    (set! r10 fv1)
    (set! rax (+ rax r10))
    (jump done))
42                                            #lang cpsc411/hashlangs/base
```

We perform a non-tail call to identity with the argument 41, which is passed on the stack instead of in a register. We also have one variable live across the call, fv1. return-point expands to, essentially, let/cc, and represents the start of a non-tail call that returns to the continuation l. All frame variables are relative to the caller's frame. However, the frame is allocated just before the non-tail call, so the references to fv0 would be to some uninitialized memory on the callee's frame if we did not correct the interpretation of frame variables with respect to frame allocation. This requires expanding *fvars* to include a dynamic calculation into the stack.

We modify the earlier definition of set! to keep track of current-fvar-offset as follows.

```
(define-syntax (set! stx)
    (syntax-parse stx
      ; Stack pointer increment
      [(_ (~literal rbp) (binop (~literal rbp) v))
       #`(begin
          (set-box! current-fvar-offset (binop (unbox current-fvar-offset) v))
          (set-box! _rbp (binop rbp v)))]
      ....))
```

Any time the stack pointer is adjusted, we generate a run-time update to `current-fvar-offset` to cooperate with the implementation of frame variables, in addition to the run-time modification of the stack pointer.

4 CUSTOM PROCEDURES

So far, we primarily used Racket's macro system for compile-time interposition on syntactic forms to implement our embedding. However, one feature relies on impersonators (Strickland et al. 2012), a feature for run-time interposition on some datatype.

Our compiler implements procedures as a tagged datatype that stores its arity, to implement dynamic checks, and as a closure, so the label is stored with the values of all variables free in the procedure's body. This requries a different procedure interface than Racket's to support additional operations in intermediate languages. But, we would like our procedures to expand to Racket procedures, so application continues to work as expected, and so we can interpret each language primitive as Racket's equivalent procedure.

To impersonate a procedure, Racket provides `impersonate-procedure`, which takes an underlying procedure implementation, an optional wrapper around the result, and a list of optional properties that attach additional information to the procedure. Our implementation of procedures is given below.

```
  (define (make-procedure label arity env-size)
    (impersonate-procedure label #f
                           proc-label:prop label
                           proc-env:prop (make-vector env-size)
                           proc-arity:prop arity))
```

In the compiler, procedure datatypes are built from this primitive form `make-procedure`. The `label` is an assembly label to the code for the procedure (modelled as a Racket procedure). The `arity` describes the arity of the procedure from the user's perspective, *i.e.*, the number of parameters the user specified in the definition. The `env-size` declares the size of the procedure's environment. We embed our procedures as a Racket procedure impersonator, implemented by the underlying `label`, with three properties attaching information needed by intermediate language interfaces to procedures.

We define the impersonater properties below.

```
  (define-values (proc-label:prop proc-label:prop? unsafe-procedure-label)
    (make-impersonator-property 'procedure-label))

  (define-values (proc-env:prop proc-env:prop? unsafe-procedure-env)
    (make-impersonator-property 'procedure-env))

  (define-values (proc-arity:prop proc-arity:prop? unsafe-procedure-arity)
    (make-impersonator-property 'procedure-arity))
```

```
(define (unsafe-procedure-ref p i)
  (vector-ref (unsafe-procedure-env p) i))

(define (unsafe-procedure-set! p i v)
  (vector-set! (unsafe-procedure-env p) i v))
```

Each impersonator property takes a name as a symbol, and returns three values: the property, a predicate identifying the property, and an accessor which retrieves the value of the property from an impersonator to which the property is attached. We also define the interface for updating the procedure's environment. In the compiled output, the environment is part of the procedure's contiguous representation in memory with the label and arity, but in our embedding, we attach it separately as a vector.

Racket already supports introspecting on a procedure's arity using `procedure-arity`. However, in our embedding, the arity from Racket's perspective is that of the underlying implementation (*i.e.* the arity of the label), which differs from the arity of the original procedure. In the intermediate languages, a procedure takes itself as an argument to have access to its environment. We therefore need the arity from the user's perspective, to provide our own implementation of `procedure-arity` that is correct with respect to the user's perspective.

For example, below we define an intermediate language program that corresponds to the compiled output of the procedure `(let ([y 21]) (lambda (x) (+ x y)))`, then call it, and inspect its arity.

```
> (require (only-in racket/base [lambda r:lambda]
                                [procedure-arity r:procedure-arity]))
> (define foo
    (make-procedure (r:lambda (c x) (+ x (unsafe-procedure-ref c 0))) 1 1))
> (unsafe-procedure-set! foo 0 21)
> (call foo foo 21)
42
> (unsafe-procedure-arity foo)
1
> (r:procedure-arity foo)
2                                                    #lang cpsc411/hashlangs/base
```

The procedure is created using `make-procedure`. The embedded representation of a label is a Racket procedure. The original procedure has one parameter, and has one variable in its environment. A low-level call to the procedure explicitly passes the procedure itself as the first argument, and then the arguments corresponding to the parameters. Variables that were free in the body are now explicitly read from the procedure's environment. We can see from `r:procedure-arity` that Racket believes this procedure has arity `2`, but using the accessor defined by our impersonator property, we get the correct (for the source language definition of the procedure) `1`.

Impersonators are crucial to both reusing, but also extending, existing run-time datatypes.

5 INTERFACE DETAILS: INTERPOSITIONS AND EVAL

In most of the intermediate languages, the interpretation of syntax stays the same. However, eventually we add tagged datatypes and must reinterpret earlier primitives. For example, in the low-level languages, `+` is the 64-bit twos-complement addition.

```
> (require cpsc411/machine-ints)
> (+ (max-int 64) #b0)
9223372036854775807
> (+ (max-int 64) #b1)
-9223372036854775808                               #lang cpsc411/hashlangs/base
```

But, after we add tagged data, the surface-language + is defined on 61-bit tagged integers. Intermediate languages further in the pipeline implement this in terms of the 64-bit +.

```
> (require cpsc411/machine-ints)
> (+ (max-int 61) #b0)
1152921504606846975
> (+ (max-int 61) #b1)
-1152921504606846976                                 #lang cpsc411/hashlangs/v7
```

Racket's module system makes importing the existing embedding, redefining +, and exposing only the new implementation easy. In fact, it is no different from extending Racket with an embedding of x64 features; the same features that make embedding x64 into Racket easy make extending our x64 embedding easy.

We essentially define a module like the following.

```
(module tagged-+ racket
  (require (except-in "base.rkt" +)
           cpsc411/machine-ints
           (rename-in racket [define r:define]))
  (provide (all-from-out "base.rkt")
           +)

  (r:define + (curry twos-complement-add 61)))
```

We import everything from the base embedding, except +. Then, we export all the base definitions we imported, and export our new 61-bit two's complement version of +. The new module can then be used as a language. This is the language-oriented approach to building a new interpreter.

While using these embedded languages in the REPL and as a #lang is useful, recall that our original goal was to write *interpreters*. The embedded language provides an interpretation of the programs, but we have not yet defined an interpreter, *i.e.*, a function that consumes the syntax of the program and returns the result.

To do this, we reuse eval, which takes a representation of program syntax (either as a syntax object or as a quoted list) and evaluates it. In Racket, however, eval also takes an optional second argument, specifying the namespace in which to evaluate the program. We can use this to specify evaluating the program in our language, rather than in Racket.

```
> (require cpsc411/machine-ints)
> (define (base-eval x)
    (local-require cpsc411/hashlangs/base)
    (eval x (module->namespace 'cpsc411/hashlangs/base)))
> (define (v7-eval x)
    (local-require cpsc411/hashlangs/v7)
    (eval x (module->namespace 'cpsc411/hashlangs/v7)))
> (base-eval `(+ ,(max-int 64) 0))
9223372036854775807
```

```
> (base-eval `(+ ,(max-int 64) 1))
-9223372036854775808
> (v7-eval `(+ ,(max-int 61) 0))
1152921504606846975
> (v7-eval `(+ ,(max-int 61) 1))
-1152921504606846976                                                          #lang racket
```

Now, when we need an interpreter for one of the intermediate languages, we wrap `eval` with the namespace derived from the module that embeds the right set of features. To expose all the right identifiers, we still did have to do that about 126 times, though.

This method of implementing our interpreters has some limitations. Since almost all the features of all the languages are implemented in a single module, invalid programs have an interpretation. For example, we managed to evaluate the expression `` `(+ ,(max-int 64) 0) ``, although in none of the grammars above is that considered a program.

We separate the implementation of the interpreters from validating the input programs. We use a DSL to generate PLT Redex (Felleisen et al. 2009; Klein et al. 2012) grammar definitions from our typeset language grammars. We then use Redex's `redex-match` to generate predicates for checking that terms are syntactically well formed for each language; the p non-terminal defines a valid program in the language.

As an example, consider the languages involved in adding tagged data. Similar to the previous grammar, in Section 3, the surface language requires programs to be a module with optional top-level procedure definitions and a top-level expression. Since expressions are over tagged data, integer literals are restricted to 61 bits.

```
> (require cpsc411/langs/v7 cpsc411/machine-ints)
> (exprs-lang-v7? `(+ ,(max-int 64) 0))
#f
> (interp-exprs-lang-v7 `(+ ,(max-int 64) 0))
-1
> (exprs-lang-v7? `(module (call + ,(max-int 61) 0)))
#t
> (interp-exprs-lang-v7 `(module (call + ,(max-int 61) 0)))
1152921504606846975                                                           #lang racket
```

The expression `` `(+ ,(max-int 64) 0) `` is invalid, since this language requires an explicit `call` form to call a procedure, and because the language only supports 61-bit integer literals. But the interpreter happily does something anyway. It is useful to expose this unsafe interface to discuss things such as undefined behaviour, but we may want to expose another interface to warn the user that a test failed because the program was syntactically invalid, and not because the behaviour changed.

We can combine the interpreter and these predicates at our desired interface to ensure that our interpreters only receive well-formed terms, using contracts (Findler and Felleisen 2002). Below, we attach the predicate as a contract to the interpreter as it is exported from the module `checked-interp-v7`. The previous, uncontracted implementation is still available from the original module. The contract requires the input to the interpreter to satsify the predicate `exprs-lang-v7?`, and output from the interpreter to be a 61-bit integer.

```
> (module checked-interp-v7 racket
    (require cpsc411/langs/v7 cpsc411/compiler-lib)
```

```
     (provide (contract-out
               [interp-exprs-lang-v7 (-> exprs-lang-v7? int61?)]))
> (require 'checked-interp-v7)
> (interp-exprs-lang-v7 `(+ ,(max-int 64) 0))
interp-exprs-lang-v7: contract violation
  expected: exprs-lang-v7?
  given: '(+ 9223372036854775807 0)
  in: the 1st argument of
      (-> exprs-lang-v7? int61?)
  contract from: checked-interp-v7
  blaming: program
   (assuming the contract is correct)
  at: eval:6:0
> (interp-exprs-lang-v7 `(module (call + ,(max-int 64) 0)))
interp-exprs-lang-v7: contract violation
  expected: exprs-lang-v7?
  given: '(module (call + 9223372036854775807 0))
  in: the 1st argument of
      (-> exprs-lang-v7? int61?)
  contract from: checked-interp-v7
  blaming: program
   (assuming the contract is correct)
  at: eval:6:0
> (interp-exprs-lang-v7 `(module (call + ,(max-int 61) 0)))
1152921504606846975                                              #lang racket
```

The error message isn't precise since the predicate we use in the contract is naïvely generated, but it does tell a user whose test fails that the problem is an invalid source program provided to the interpreter, rather than a compiler pass producing a valid a program that runs to an unexpected result.

6 DISCUSSION

We presented an extended case study of language-oriented design to macro-embed a family of compiler intermediate languages into Racket. The same abstractions used to implement new general-purpose languages and various high-level DSLs prove equally useful for low-level languages and exposing a family of languages quickly.

The full implementation differs slightly from what we described, although the typeset interactive examples all run as above. The main difference a reader might notice is that global state, such as registers and memory, are implicitly reset at module boundaries, except during REPL interactions. Resetting global state makes interoperability with Racket more difficult, but simplifies our testing framework.

Aside from the engineering benefits of the embedding, one persistant thought during this experience is that the implementation by local, compositional macro-embedding essentially forces us to construct a compositional *model* of the language, in a way that writing a recursive interpreter does not. This view of constructing-a-model-as-implementation has been beneficial in helping us understand the semantics of the languages and identifying weaknesses. Invariably, when we cannot find an easy local embedding, one of two things is going on: either the language is poorly designed, or we don't understand the language as well as we thought we did. One instance that stood out and is discussed earlier is our implementation of abstract locations, which requires inverting expansion order and performing an analysis. This analysis felt unnecessary, and upon reflection, we realized the compiler already had the binding information we were trying to reconstruct, but threw it away. If we redesigned the intermediate languages to declare the scope of abstract locations, this would improve our embedding, but also have pedagogical benefits for the course—students are often confused about the scope of abstract locations, and of course they are when no form explicitly declares their scope. A similar thought occurs about frame variables; instead of doing either an analysis or binding an arbitrary number of them *a priori*, wouldn't it be better if we could embed a new variable-like form? If the syntax were `(fv 1)` instead of `fv1`, and variable-like forms could be compound rather than just identifiers, then embedding frame variables would be as easy as registers.

During this project, we struggled in a couple of places that suggest the need for new abstractions.

The primary one discussed in this article is the use of a more general definition of variable by `set!`, so that addresses like `(rbp - 0)` can be interpreted as assignable. We seem to need a generalization of `make-variable-like-transformer` that supports a variable-like *expression*, that can appear either as an identifier or as an operator. Such "variables" could be referenced as expressions, or appear as the left-hand side of a `set!` to indicate assignment. Then addresses could be macro-embedded modularly, like registers, although because registers in operator position would be part of an address, such an implementation would interact with the embedding of registers. The abstraction should be a simple extension of `set!`. Although this feature is desirable in our implementation, it is unclear whether it would have more general use. As is, `set!` implements multiple cross-cutting functionality, and is the least modular aspect of our implementation.

One headache not discussed in this article is the transition from a module language to a `#lang`. Any module can easily be interpreted as an s-expression-based language, with its exported identifiers forming the language. For example, the module defining our macro embedding is `cpsc411/langs/base`; to use it as a language we need only start a language line with `#lang s-exp cpsc411/langs/base`. However, excluding the `s-exp` meta-language to get a language that has the same status as `#lang racket` takes work. We must create a particular file and module structure for each language. As a result, we only create two honest-to-goodness `#lang`s, and normally use `#lang s-exp cpsc411/langs/base` instead. This kind of boilerplate outside the linguistic abstractions is in opposition to The Racket Manifesto (Felleisen et al. 2015), although we don't know how to improve it.

7 ACKNOWLEDGEMENTS

For their time and thoughtful feedback on this paper, I gratefully acknowledge the anonymous reviewers, Jonathan Chan, Yanze Li, Nico Ritschel, Eric Conlon, and Ivan Beschastnikh. I owe many thanks to Andy Keep, Roshan James, and R. Kent Dybvig for teaching me compiler design and implementation, and providing the course notes on which the design of these intermediate languages are based. Thanks also to Ron Garcia, Lily Bryant, Justin Hu, Yuchong Pan, Benjamin Lerner, and David Ewert for providing support and feedback on this project. Finally, I owe a big thank you to all the students of UBC's CPSC 411 for testing and reporting bugs in this software.

BIBLIOGRAPHY

Ryan Culpepper. Fortifying macros. *Journal of Functional Programming (JFP)* 22(4-5), pp. 439–476, 2012.

Ryan Culpepper and Matthias Felleisen. Fortifying macros. In *Proc. International Conference on Functional Programming (ICFP)*, 2010.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. 2009. https://mitpress.mit.edu/9780262062756/semantics-engineering-with-plt-redex/

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay Mccarthy, and Sam Tobin-Hochstadt. A Programmable Programing Language. *Communications of the ACM* 61(3), pp. 62–71, 2018. https://cacm.acm.org/magazines/2018/3/225475-a-programmable-programming-language/fulltext

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In *Proc. SNAPL*, 2015.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proc. International Conference on Functional Programming (ICFP)*, 2002.

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *Proc. International Conference on Functional Programming (ICFP)*, 2009.

Abdulaziz Ghuloum. An Incremental Approach to Compiler Construction. In *Proc. Scheme Workshop*, 2006. http://scheme2006.cs.uchicago.edu/11-ghuloum.pdf

Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In *Proc. International Conference on Functional Programming (ICFP)*, 2013.

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proc. Symposium on Principles of Programming Languages (POPL)*, 2012.

Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. FunTAL: Reasonably Mixing a Functional Language with Assembly. In *Proc. International Conference on Programming Language Design and Implementation (PLDI)*, 2017. http://www.ccs.neu.edu/home/amal/papers/funtal.pdf

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012.

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Proc. Symposium on Principles of Programming Languages (POPL)*, 2008.

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme: From Scripts to Programs. *CoRR* abs/1106.2575, 2011. http://arxiv.org/abs/1106.2575

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages As Libraries. In *Proc. International Conference on Programming Language Design and Implementation (PLDI)*, 2011.