Higher-Order Code Splicing

Peter Thiemann*
thiemann@informatik.uni-freiburg.de

Institut für Informatik
Universität Freiburg, Germany

Abstract. Run-time code generation (RTCG) and just-in-time compilation (JIT) are features of modern programming systems to strike the balance between generality and efficiency. Since RTCG and JIT techniques are not portable and notoriously hard to implement, we propose code splicing as an alternative for dynamically-typed higher-order programming languages. Code splicing combines precompiled pieces of code using higher-order functions. While this approach cannot achieve the performance of compiled code, it can support some intriguing features:

- very fast "compilation" times;
- satisfactory run times, compared with interpretation;
- simple interfacing with compiled code;
- portability.

Starting from implementation models for functional languages we develop and evaluate several approaches to code splicing. This leads to some new insights into compilation techniques for functional programming languages, among them a compositional compilation schema to SKI-combinators. The progression of different techniques sheds some light on their relationship, specifically between combinator-based implementations and closure-based implementations.
All techniques have been implemented and evaluated in Scheme.

1 Introduction

Run-time code generation and just-in-time compilation generate code at run time and execute it subsequently. To amortize code generation time, RTCG and JIT only perform simple optimizations. They can still generate competitive code by exploiting invariants that are only available at run time. On the flip-side these techniques are inherently non-portable and hard to implement because many technical details and pitfalls must be catered for (flushing instruction caches, memory management, access permissions, and so on).

However, there is some indication that RTCG is not always worth the effort. Indeed, Lee [18] stated that restructuring the code to observe a staging discipline already achieves significant speedups. So why go into the complications of RTCG or JIT if some of the speedup is already available just by staging?

* This work has been done while at the School of Computer Science and Information Technology, University of Nottingham, UK. The author acknowledges support by EPSRC grant GR/M22840 "Semantics of Specialization".

This is exactly the motivation for code splicing. Code splicing is a range of techniques for writing staged interpreters. Such an interpreter keeps compile-time computations separate from run-time computations. Applying it to a source program ideally performs all compile-time computations and returns the compiled (or code-spliced) program. It is not necessary to actually build a compiler. This style of "compilation" is attractive for a range of applications:

- The execution of applets and other mobile code: receive high-level code from a network to perform integrity checks, but execute it efficiently.
  Many applets are throw-away code which is only executed a very limited number of times. Even JIT compilation may be too expensive [2].
- The implementation of flexible domain-specific languages.
  The emphasis is on quickly designing, implementing, and (possibly) modifying the language. Writing a full-blown compiler would be too expensive because the user community for such a language is often small and efficiency is not of tantamount importance.
- Efficient metaprogramming.
  Metaprogramming systems [22] allow the generation of high-level program code and its subsequent execution in the same running program. Ideally, there should be no penalty for using generated code, but full compilation would be too slow. The approaches to combining partial evaluation and compilation also fall into this category [5, 21].
- Overcoming restrictions of compilers.
  Some compilers have arbitrary restrictions on the size of code, the number of variables, and so on. Code splicing provides a way to compile and execute arbitrary programs, overcoming implementation restrictions.

All these applications share the following requirements:

1. *Instantaneous compilation.* The time for code splicing should be comparable to the time necessary to construct the corresponding source text.
2. *Satisfactory speed.* Code-spliced programs should be significantly faster than interpreted ones.
3. *Easy interfacing.* It should be possible to freely mix code-spliced and ordinarily compiled code.
4. *Portability.*

The present paper investigates several code splicing techniques through implementations in the higher-order functional programming language Scheme [16]. All techniques are portable and most of them meet all of the requirements. They exploit many features of Scheme, e.g., dynamic typing, side effects, and the `eval` function.

The techniques are inspired by implementation techniques for functional programming languages (SKI-combinators, director strings, categorical combinators, closure conversion, and shallow binding). Most of them can be made to exhibit good staging properties. However, there are some surprises. For example, it turns out that the standard compilation method to SKI-combinators [24]

`ASyntax`	type of generated code (active syntax)
`make-var`	: `Var` $\to$ `ASyntax`
`make-lam`	: `Var` $\times$ `ASyntax` $\to$ `ASyntax`
`make-app`	: `ASyntax` $\times$ `ASyntax` $\to$ `ASyntax`
`compile-term`	: `ASyntax` $\to$ `Val`

Fig. 1. Signature of code generation

is unsuitable for our purposes because it is not compositional. Therefore, we devise a compositional compilation scheme to SKI-combinators and prove some correctness properties about it. Deforestation [25] makes it reasonably efficient.

Outline: Section 2 introduces our experimental setup. Section 3 explains the different approaches to compilation that we consider. Section 4 presents comparative run times. Section 5 discusses related work and Section 6 concludes.

Throughout the paper, we assume knowledge of the Scheme language [16].

2 The experimental setup

Type-directed partial evaluation (TDPE) [8] is our dynamic supply for program text. When applied to a value of type τ and a representation of the type τ, it constructs a pure lambda term of that type. To "compile" the constructed terms using code splicing, it is sufficient to provide staged interpreters for the lambda calculus. We distribute the implementation of such an interpreter over the syntax constructors for the lambda calculus to avoid the cost of the syntax dispatch. That is, the `make-var` function interprets a variable expression, the `make-lam` function interprets a lambda abstraction, and `make-app` implements function application, taking the interpreted subexpressions as parameters. Fig. 1 defines the types of these functions. Each interpreter provides its own definition of the type `ASyntax` (for active syntax) and the functions listed, the constructors for this type. The implementation includes multi-argument versions `make-lam*` and `make-app*` of lambda abstraction and application, as well as `make-let`.

3 Approaches to code splicing

The following subsections introduce different ways to achieve code splicing. Each choice distributes compile-time work between the active syntax constructors and `compile-term` in a different way. In one extreme (see Sec. 3.1), the active syntax constructors construct Scheme expressions and `compile-term` performs full compilation. In the other extreme [21], the active syntax constructors emit and combine byte code and `compile-term` is the identity function.

```
(define (make-var x)      x)
(define (make-lam x e)    `(LAMBDA (,x) ,e))
(define (make-app f a)    `(,f ,a))
(define (compile-term e) (eval e (interaction-environment)))
```

Fig. 2. Active syntax constructors for Scheme

3.1 Using `eval`

The Scheme standard [16] includes a function `eval` that maps a Scheme expression to its value. Hence, the syntax constructors can construct Scheme expressions (using `quasiquote` "`" and `unquote` ",") and `compile-term` can be `eval` (see Fig. 2).

Unfortunately, this implementation does not fulfill all of our requirements: compilation speed is heavily implementation and system dependent. Furthermore, the compiled code is likely to uncover implementation restrictions of the underlying language implementation.

3.2 SKI combinators

This approach takes advantage of a precompiled library of implementations of the combinators S, K, and I. After compiling the source program to a combinator term, an interpreter just sticks the precompiled combinators together as prescribed by this term.

Unfortunately, the naive compilation generates abysmal code and requires multiple passes. An optimized compilation [24] generates much better and smaller code, but it still requires multiple passes. However, we can do better.

Compositional compilation to combinators Some research has gone into optimized combinator systems that keep the resulting terms small [24]. If we adopt the additional combinators B and S' defined by

B = `(lambda (x) (lambda (y) (lambda (z) (x (y z)))))`
S' = `(lambda (k) (lambda (x) (lambda (y) (lambda (z) ((k (x z)) (y z))))))`

then there is a compositional specification of compilation. Let us consider the three constructs in turn for `ASyntax` = `CEnv` $\rightarrow$ `SKI` where the compile-time environment `CEnv` = `Var`* is the list of the bound variables in the reverse order in which they were bound and `SKI` is the set of combinator terms.

`make-var`. An access to the ith variable in an environment of size n compiles into a projection function that returns the ith argument out of n (where $0 \leq i < n$ and $n > 0$): $\lambda x_0 \ldots \lambda x_{n-1}.x_i$. This function can be expressed as $K^{(i)}((BK)^{(n-i-1)}(I))$ where $X^{(0)} = I$ and $X^{(i+1)} = BX^{(i)}X$ is the iterated composition of X (proof by case analysis and induction). Using the equivalent definition $X^{(0)}Y = Y$ and $X^{(i+1)}Y = X^{(i)}(XY)$, we get for $i = 1$ and $n = 3$ the combinator term $K(BKI)$.

`make-lam`. Compilation just needs to update the compile-time environment with the new variable. The translation of the body provides the additional abstraction.

`make-app`. If n is the size of the environment, the combinator

$$S_n f a = \lambda x_0 \ldots \lambda x_{n-1}.(f x_0 \ldots x_{n-1})(a x_0 \ldots x_{n-1})$$

distributes n values to the compiled subterms f and a of the application. Rewriting this combinator using only S, K, and I leads to an explosion of the size of the term. However, the S' combinator was conceived to solve this problem. It is easy to show that, for all $n \geq 0$, $S_n = ({S'}^{(n)}(I))$ (proof by induction).

Assessment The resulting combinator terms can be quadratic in the size of the source term (because the size n of the environment is only bounded by the size of the source term), hence the compile time is also at least quadratic.

In addition, the granularity of the compiled code is too fine, leading to disappointing performance.

Finally, dealing with multi-argument functions is awkward. The standard solution [11, sec. 12.2.3] is to introduce an untupling combinator

$U =$ `(lambda (f) (lambda (x . xs) (apply (f x) xs)))`

and implement the translation of multi-argument abstractions accordingly.

Director strings The key idea of this approach is the following invariant:

pass only the values of the free variables to the compiled expression.

For this optimization, we need further combinators B', C', and C'' defined by

B' = `(lambda (k) (lambda (x) (lambda (y) (lambda (z) ((k x) (y z))))))`
C' = `(lambda (k) (lambda (x) (lambda (y) (lambda (z) ((k (x z)) y)))))`
C'' = `(lambda (k) (lambda (x) (lambda (y) (lambda (z) ((k (x y)) z)))))`

During compilation, the compiler trims the environment according to the free variables while keeping their order. This idea corresponds exactly to director strings [17], where S', B', and C' correspond to Λ, \, and /. The `make-`*xxx* functions compute the free variables on the fly at compile time.

`(make-var x)`. By the invariant, only the value of `x` is passed to this compiled expressions. Hence the translation is I.

`e` $\equiv$ `(make-lam x b)`. If `x` occurs free in `b` then the free variables of `b` are exactly `x` and the free variables of `e`: nothing needs to be done.

If `x` does not occur free in `b` then the free variables of `b` are identical to the free variables of `e`. In this case an additional abstraction must be conjured up. If n is the number of free variables and A is the compiled body then ${C''}^{(n)} K A$ does the job, since, for all $n \geq 0$,

$$\lambda x_0 \ldots \lambda x_n . A x_0 \ldots x_{n-1} = {C''}^{(n)} K A.$$

```
(define (make-var x)
  (lambda (env)
    (let* ((n (length env))
           (i (index x env)))
      (projection i n))))

(define (make-lam x e)
  (lambda (env)
    (e (cons x env))))
```

```
(define (make-app e1 e2)
  (lambda (env)
    (let* ((c1 (e1 env))
           (c2 (e2 env))
           (n (length env)))
      ((application n) c1 c2))))

(define (compile-term e1)
  (e1 '()))
```

Fig. 3. Active syntax constructors: customized combinators

`e` $\equiv$ `(make-app f a)`. This case involves bookkeeping of the free variables of `f`, `a`, and `e`. Let $x_1, \ldots, x_n$ be the free variables of `e` in the order of their bindings and compile to $X_1(X_2 \ldots (X_n I))FA$, where F and A are the compiled terms for f and a and, for $1 \leq i \leq n$,

- $X_i \equiv S'$ if $x_i \in FV(\texttt{f}) \cap FV(\texttt{a})$;
- $X_i \equiv C'$ if $x_i \in FV(\texttt{f}) \setminus FV(\texttt{a})$;
- $X_i \equiv B'$ if $x_i \in FV(\texttt{a}) \setminus FV(\texttt{f})$.

Deforestation Our implementation uses deforestation twice to eliminate intermediate results. First, the active syntax construction is re-interpreted as compilation to a combinator expression. Second, the syntax constructors of the resulting combinator expression (combinator constants and application) are re-interpreted as the compiled values of the combinators and as function application.

3.3 Customized combinators

A possible improvement to using SKI combinators lies in providing precompiled projection functions `(projection i n)` (with `n` $>$ 0 and 0 $\leq$ `i` $<$ `n`) and a generalized S combinator `(application n)` that distributes `n` ≥ 0 arguments.

```
(define (projection i n)
  (lambda (x0) ... (lambda (xn-1) xi)))
(define (application n)
  (lambda (f) (lambda (a)
    (lambda (x0) ... (lambda (xn-1)
      (((f x0) ... xn-1)
       ((a x0) ... xn-1)))))))
```

With this approach, the active syntax constructors perform the entire compilation, as shown in Fig. 3. The type of the active syntax is still `CEnv` $\rightarrow$ `Val` where `CEnv` is as before. The translation of a variable selects a projection using the size of `CEnv` and the index of the variable in it. The translation of a lambda abstraction only pushes the new variable onto the environment, it does not generate code. `Make-app` compiles its subexpressions first. Then it splices the two pieces of

```
(define (make-var x)                    (define (make-app e1 e2)
  (lambda (env)                           (lambda (env)
    (let ((j (index x env)))                (let* ((c1 (e1 env))
      (lambda vs                                   (c2 (e2 env)))
        (list-ref vs j)))))                   (lambda xs
                                                ((apply c1 xs)
(define (make-lam x e)                           (apply c2 xs))))))
  (lambda (env)
    (let ((c (e (cons x env))))         (define (compile-term e)
      (lambda vs                          ((e '())))
        (lambda (y)
          (apply c (cons y vs)))))))
```

Fig. 4. Active syntax constructors: multi-argument combinators

code together to an application using `application`. The function `compile-term` initiates compilation by applying the value to the empty environment `'()`.

This compilation scheme exhibits linear time behavior in practice. Asymptotically it is quadratic due to the linear scans through the environment, whose length is also bounded by the size of the expression.

There are at least three ways to implement this approach in Scheme.

1. Generate the text of the `projection` and `application` functions as required, compile them using `eval`, and cache the results.
2. Use generic versions of `projection` and `application` in Scheme.
3. A mixed approach provides the precompiled versions up to some fixed n_0, falling back to a generic implementation for $n > n_0$.

A drawback is the complicated treatment of multi-argument functions which must be resolved in the same way as explained in Sec. 3.2.

3.4 Multi-argument combinators

The combinator approaches described in the preceding sections represent the run-time environment implicitly using abstractions "built into" the combinators S, K, and I, and generalizations thereof. This representation corresponds roughly to a list of the values of the variables, i.e., a nested representation of closures. It is inefficient because the free variables are passed one by one at each application and variable access.

The next logical step consists of flattening these structures and passing them around using multi-argument functions. With this approach, one application passes the entire run-time environment. Consequently, the type of the active syntax is $\prod \mathtt{env} : \mathtt{Var}^*.\mathtt{Val}^{|\mathtt{env}|} \to \mathtt{Val}$ where $|\mathtt{env}|$ is the size of the environment. The staging is important: compilation executes the $\prod \mathtt{env} : \mathtt{Var}^* \ldots$ part, whereas the $\mathtt{Val}^{|\mathtt{env}|} \to \mathtt{Val}$ part is left till run-time.

Figure 4 shows the interpretation of the constructors. The variable case computes the index `j` of the variable in the run-time environment and returns the generic projection function `(lambda vs (list-ref vs j))` which maps a tuple of size at least `j+1` to the value of its `j`th component. `Make-lam` transfers the argument value `y` into the run-time environment. Effectively, it curries the compiled body `c` of the abstraction. `Make-app` is a generalized S combinator.

It is intriguing to see that we can directly map to compiled code so that not much work is left for `compile-term`: it applies active syntax to the empty environment and runs the resulting thunk (a parameterless function).

The code uses generic implementations of the `projection` and `application` functions (which is viable due to the use of the flat representation of the run-time environment). It is again possible to generate customized versions for each particular arity and cache their compiled code as explained above. In this case, we also need currying functions for different sizes of the run-time environment. To our surprise, we found that the cost of caching was higher than the cost of using the generic implementation. We suspect that checking the number of arguments for functions of fixed arity is the culprit: the generic implementation uses variadic functions that do not check the number of their arguments.

3.5 Explicit linked-list run-time environment

Another variation is inspired by the implementation technique of the categorical abstract machine [7]. Standard expository texts [12] employ the same technique.

The type of the active syntax is `CEnv` $\rightarrow$ `REnv` $\rightarrow$ `Val`. The representation of the run-time environment `REnv` is a linked list of vectors. Each vector holds the values of the variables abstracted by one surrounding lambda. The compile-time environment maps a variable to a depth and an offset, the depth determines the index in the outer linked list and the offset determines the position of the variable's value in the vector. Hence, `make-var` relies on cached projection functions, one for each pair of depth and offset.

`Make-lam` is implemented in terms of its multi-argument cousin `make-lam*`:

```
(define (make-lam* xs e)
  (lambda (env)
    (let ((comp (e (cons xs env))))
      (lambda (rt-env)
        (lambda ys
          (comp (cons (list->vector ys) rt-env)))))))
```

`Make-lam*` is staged: It performs all computations that only depend on the compile-time environment `env` before it abstracts the run-time environment `rt-env`. The run-time part of `make-app` is the S combinator and `compile-term` just supplies the initial compile-time environment.

3.6 Pass free variables

A refinement of the multi-argument approach of Sec. 3.4 passes only the values of the free variables at run time. There is no need to maintain an environment at compile time, instead compilation generates the list of free variables along with

```
(define (make-var x)
  (list (list x)
        (lambda (x) x)))

(define (make-lam x e)
  (let* ((fv-e (freevars e))
         (comp-e (compiled e))
         (fv (set-subtract fv-e x))
         (comp ((get-lambda* (length fv)
                             1
                             (names->index (append fv (list x)) fv-e))
                comp-e)))
    (list fv comp)))
```

Fig. 5. Combinators to pass free variables

the code and the code expects the values of the variables to be passed to it in the sequence specified by the list. Hence, the active syntax type is a dependent sum $\sum \texttt{env} : \texttt{Var}^*.(\texttt{Val}^{|\texttt{env}|} \to \texttt{Val})$, i.e., a pair containing the list of the free variables and the corresponding function. The functions `freevars` and `compiled` are the projections on the first and second component.

Due to the invariant that exactly the values of the free variables are passed to each expression the compilation of variables is straightforward (see Fig. 5).

Compilation of lambda abstraction is more involved, due to the fact that the abstracted variable may not occur free in the body. It relies on a cached function `get-lambda*` that takes the number of variables to be expected from the context (`(length fv)`), the number of variables abstracted (`1`), and a list of numbers that select those variables that are passed on to the body (`(names->index (append fv (list x)) fv-e)`).

For applications, there is similar cached function `get-application*` that selects and distributes the values of the free variables.

3.7 Shallow binding

Shallow binding [4] inspires an approach which represents variables by reference cells. It is safe to bind variables to fixed mutable cells *at compile time*, provided that each cell contains a stack of values with the current value on top.

The type of the active syntax is $\texttt{CEnv} \to \texttt{Unit} \to \texttt{Val}$. There is no explicit run-time environment. All name resolution is performed at compile time.

To conserve space, we only consider the compilation of a lambda abstraction in Fig. 6. The compile-time environment `env` is a list of pairs of variable names and cells. Compilation creates a new cell `c0` corresponding to the variable `v0`. Next, it collects the cells that make up the current environment in `cells`. Finally, it compiles the body of the lambda while binding `v0` to `c0`.

The resulting thunk which is executed at run time forms a closure consisting of the current values of the cells in the environment and returns the real lambda

```
(define (make-lam v0 e)
  (lambda (env)
    (let* ((c0 (make-cell '()))
           (cells (map cdr env))
           (body (e (cons (cons v0 c0) env))))
      (lambda ()
        (let ((freevals (map (lambda (c) (car (cell-ref c))) cells)))
          (lambda (x0)
            (cell-set! c0 (cons x0 (cell-ref c0)))
            (for-each (lambda (c x) (cell-set! c (cons x (cell-ref c))))
                      cells freevals)
            (let ((result (body)))
              (cell-set! c0 (cdr (cell-ref c0)))
              (for-each (lambda (c) (cell-set! c (cdr (cell-ref c))))
                        cells)
              result)))))))
```

Fig. 6. Compilation of lambda abstraction using references

`(lambda (x0) ...)`. Applying the lambda establishes the new binding of the variable `v0` and installs the values of the free variables from the closure. After running the body, it restores all bindings to their previous state before returning the result.

The compilation of a variable obtains its associated cell and returns a thunk that returns the top of the stack stored in that cell.

The compilation of an application wraps the application in a thunk that runs the thunks of the subexpressions as appropriate.

On entry to the body of a lambda only the cells corresponding to free variables of the body need updating. This optimization is straightforward.

4 Results

This section reports practical experiments with implementations of most of the approaches described. We have run three sets of benchmark programs on two different interpreted Scheme implementations, Scheme48 version 0.51 and Gambit 3.0. Scheme48 compiles to byte code for subsequent interpretation whereas Gambit is a quasi source-level interpreter. All measurements were performed on a 233MHz Pentium II machine with 256MB of memory, running FreeBSD 2.2.5. We report the average time of ten runs of the same computations, using the respective `time` commands of the Scheme systems.

We have measured the following variations of code splicing:

scheme section 3.1, generate Scheme source and compile using `eval`;
ski-opt section 3.2 but using an intermediate combinator expression;
ski-comp is the fully deforested version;
flat section 3.4, pass the run-time environment as a flat tuple;

linked section 3.5, implement the environment by a linked list;
free section 3.6, flat environment restricted to the free variables;
ref section 3.7, naive shallow binding; **ref-free** only updates free variables.

We have measured times for

constr construction: generate an object of type `ASyntax`;
comp compile with `compile-term`;
run run the resulting procedure.

We have used TDPE to generate lambda expressions as follows:

church/n constructs the text of the Church numeral for n (see Table 1);
tiny specialize the Tiny (a little imperative language [19]) interpreter wrt. a factorial program (see Table 2);
mixwell specialize the Mixwell (a first-order functional language [15]) interpreter wrt. a program with about 300 functions (see Table 3).

In almost all cases, the construction time of the active syntax is equal to the construction time of the corresponding source. The exception is **free** which performs a free variable analysis at construction time. This amounts to a slowdown by a factor of 10 for **church** and 4 for **tiny** and **mixwell**.

With Scheme48, the time taken for construction and splicing is linear in the size of the source term. In Gambit, it seems to be quadratic. Compilation using `eval` seems to take quadratic time for both systems. In the **church** benchmark, Scheme48 ran out of memory for **scheme/10000** given the maximum possible heap size `-h 33539072`.

The holes in the **mixwell** table come from heap overflow for **linked** and an implementation restriction for **scheme**: the distributed version limits the nesting depth of bindings to 256 while the **mixwell** program has a nesting depth of well over 300. The table demonstrates that code splicing overcomes such restrictions.[1]

Overall, the **free** approach comes out fastest for all benchmarks. However, its compilation time is much slower than all others. The second choice is between **linked** and **ref-free** which combine extremely fast compilation time with fairly good execution times. The **church** benchmark seems to give a fairly distorted picture because of its uncharacteristic behavior for **ref-free**. The **tiny** and **mixwell** programs appear to yield more realistic results.

In terms of compilation time of **ski-comp** vs. **ski-opt**, deforestation pays off: **ski-comp** is two times faster. In terms of run time, **ski-comp** is slightly slower because **ski-opt** avoids some uses of the I combinator by inspecting the text of the combinator expression; this is impossible for **ski-comp** which does not generate an intermediate result.

For the **tiny** and **mixwell** benchmarks, it is interesting to compare with the run time of the original interpreter prior to specialization with TDPE. For **tiny**, its run time is 0.47 seconds for Scheme48 and 0.165 seconds for the Gambit interpreter. For **mixwell**, it is 0.712 seconds (Scheme48) and 0.64 seconds

[1] This restriction has been removed in the mean time.

technique	Scheme48 constr	Scheme48 comp	Scheme48 run	Gambit Interpreter constr	Gambit Interpreter comp	Gambit Interpreter run
scheme/100	0.00	0.28	0.00	0.005	0.003	0.001
scheme/1000	0.03	23.08	0.00	0.085	0.069	0.005
scheme/10000	0.37	—	—	2.640	3.510	2.776
scheme/100000	3.75	—	—	206.993	352.020	91.521
ski-opt/1000	0.03	0.22	0.01	0.073	0.477	0.703
ski-opt/10000	0.37	2.26	0.17	2.649	9.535	17.414
ski-opt/100000	3.72	23.65	1.71	191.800	547.981	740.677
ski-comp/1000	0.03	0.10	0.01	0.087	0.370	0.684
ski-comp/10000	0.37	1.14	0.19	2.676	6.174	21.829
ski-comp/100000	3.74	11.66	1.95	204.936	276.527	868.653
flat/1000	0.03	0.03	0.01	0.087	0.066	0.352
flat/10000	0.37	0.32	0.17	3.649	3.020	8.827
flat/100000	3.71	3.28	1.76	206.176	280.349	450.799
linked/1000	0.04	0.13	0.02	0.090	0.126	0.888
linked/10000	0.40	1.33	0.25	3.099	4.161	13.969
linked/100000	4.08	13.47	2.43	201.652	354.373	445.056
free/1000	0.32	0.00	0.00	0.394	0.000	0.349
free/10000	3.24	0.00	0.08	7.775	0.000	6.677
free/100000	32.55	0.00	0.80	240.686	0.000	125.246
ref/1000	0.03	0.03	0.00	0.086	0.061	0.344
ref/10000	0.36	0.39	0.07	3.909	2.841	9.201
ref/100000	3.68	3.96	0.79	214.278	275.319	385.260
ref-free/1000	0.03	0.07	0.00	0.161	0.117	0.355
ref-free/10000	0.37	0.73	0.08	4.695	3.886	10.433
ref-free/100000	3.74	7.50	0.87	214.103	322.110	604.603

Table 1. Run times for Church numerals (in sec)

technique	Scheme48 constr	Scheme48 comp	Scheme48 run	Scheme48 ratio	Gambit Interpreter constr	Gambit Interpreter comp	Gambit Interpreter run	Gambit Interpreter ratio
linked	0.01	0.01	0.52	1.13	0.011	0.008	0.275	1.666
flat	0.01	0.00	0.72	1.55	0.011	0.001	0.523	3.169
free	0.04	0.00	0.43	0.94	0.030	0.000	0.182	1.103
ref-free	0.01	0.00	0.72	1.53	0.010	0.009	0.534	3.236
ref	0.01	0.00	1.03	2.26	0.012	0.002	0.875	5.303
scheme	0.01	0.04	0.40	0.85	0.013	0.003	0.039	0.236

Table 2. Timings for the Tiny interpreter (sec)

	Scheme48				Gambit Interpreter			
technique	constr	comp	run	ratio	constr	comp	run	ratio
linked	3.45	—	—	—	3.870	32.329	1.158	1.89
flat	3.38	4.36	2.06	2.90	3.841	0.852	2.407	3.70
free	13.44	0.00	0.14	0.20	7.826	0.000	0.093	0.15
ref	3.33	6.42	3.80	5.34	3.743	1.582	3.966	6.26
ref-free	3.35	6.42	0.32	0.44	3.738	2.858	0.843	1.37
scheme	3.42	11.88	—	—	3.723	2.966	0.036	0.06

Table 3. Timings for the Mixwell interpreter (sec)

(Gambit interpreter). The *ratio* column contains the run time of the compiled version divided by the run time of the fully interpreted version.

Both experiments require multi-argument lambda abstraction and application. The corresponding active syntax constructors are not implemented for **ski-opt** and **ski-comp**, hence there are no results for these techniques.

The very first runs of **linked** and **free** are about an order of magnitude slower than the rest because they fill the caches.

Assessment Obviously, our mileage varies depending on the system that we use. For the Scheme48 system, which compiles to byte code, the code splicing approach seems to be viable and the **free** and **ref-free** implementations give encouraging results.

For the Gambit interpreter, the implementation of `eval` is blindingly fast and gives very good results, so this is the method of choice for the Gambit system. Why? Gambit itself uses a code splicing strategy to implement `eval` [10], but since the Gambit interpreter itself is compiled `eval` splices fully compiled code. In contrast, our experiments were conducted with the interpreter, only. Therefore, we expect to obtain better results when we use Gambit's compiler because in that case our combinators are fully compiled, too.

5 Related work

Writing staged interpreters has become popular since Feeley's thesis [9, 10]. It is now a standard technique that is taught in introductory textbooks [1]. The partial evaluation community also exploits this style and a set of transformations to improve the staging properties to specialize programs and generate compilers efficiently [14]. Holst and Gomard [13] show that the same style and very similar transformations enable a lazy functional programming language to achieve similar specialization effects as partial evaluators. In contrast, we are considering staging for strict functional programs.

One of our sources of inspiration is Augustsson's ingenious implementation of lmli, the interactive part of the lazy ML compiler [3]. Lmli compiles an interactive definition to an (SKI) combinator expression and maps it into executable code by folding the expression with respect to the compiled definitions for the combinators (with type checking turned off). This results in respectable speed for code typed in from the terminal and at the same time trivial interfacing with compiled code from other modules and with the run-time system, exactly the goals of our setting. Our design space is somewhat less constrained than Augustsson's: Since we are using an untyped, strict, and impure language, there are interesting options to consider besides SKI combinators.

Two works deal with compiling specialized source to byte code on-the-fly. Sperber and Thiemann [21] implement a back end for a traditional partial evaluator. As in the present work, they reinterpret the active syntax, but their implementation generates byte code on-the-fly. Balat and Danvy [5] construct an internal representation of the source term which they submit in toto to the byte code compiler. They do not deforest the intermediate result. Like the present work they implement a back end for TDPE, thus side-stepping problems with top-level definitions and primitive operations that contributed to the complexity of the other work [21].

6 Conclusion

We have investigated part of the design space for compilation by code splicing. We avoid an expensive compilation step by using higher-order functions to splice together precompiled pieces of code at run time. Each of the techniques that we propose has counterparts in the implementation of functional programming languages. Similar to the tradeoffs between the different implementation techniques, the splicing techniques have tradeoffs that depend on the particular source programs and on the underlying implementation.

Some of the methods have not been considered before in this context, most notably the compositional compilation to SKI combinators, director strings, and the strategy that employs shallow binding.

References

1. Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., second edition, 1996.
2. Ali-Reza Adl-Tabatabei, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In Cooper [6], pages 280–290.
3. Lennart Augustsson. The interactive lazy ML system. *Journal of Functional Programming*, 3(1):77–92, January 1993.
4. Henry G. Baker, Jr. Shallow binding in Lisp 1.5. *Communications of the ACM*, 21(7):565–569, July 1978.
5. Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In TIC '98 [23].
6. Keith D. Cooper, editor. *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 33 of *SIGPLAN Notices*, Montreal, Canada, June 1998. ACM.

7. Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. In *Proc. Functional Programming Languages and Computer Architecture 1985*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64. Springer-Verlag, 1985.
8. Olivier Danvy. Type-directed partial evaluation. In POPL1996 [20], pages 242–257.
9. Marc Feeley. Deux approches à l'implantation du language Scheme. Master's thesis, Université de Montréal, 1986.
10. Marc Feeley and Guy Lapalme. Using closures for code generation. *Computer Languages*, 12(1):47–66, 1987.
11. Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, 1988.
12. Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* MIT Press and McGraw-Hill, 1992.
13. Carsten Kehler Holst and Carsten Krough Gomard. Partial evaluation is fuller laziness. In Paul Hudak and Neil D. Jones, editors, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '91*, pages 223–233, New Haven, CT, June 1991. ACM. SIGPLAN Notices 26(9).
14. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
15. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 124–140, Dijon, France, 1985. Springer-Verlag. LNCS 202.
16. Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76, September 1998.
17. J. R. Kennaway and M. Ronan Sleep. Director strings as combinators. *ACM Transactions on Programming Languages and Systems*, 10(4):602–626, 1988.
18. Peter Lee. What is run-time code generation good for? Panel contribution at APPSEM meeting, September 1998. Pisa, Italy.
19. Larry C. Paulson. Compiler generation from denotational semantics. In Bernhard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 219–250. Cambridge University Press, 1984.
20. *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg, Fla., January 1996. ACM Press.
21. Michael Sperber and Peter Thiemann. Two for the price of one: Composing partial evaluation and compilation. In *Proc. of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 215–225, Las Vegas, NV, USA, June 1997. ACM Press.
22. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In Charles Consel, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '97*, pages 203–217, Amsterdam, The Netherlands, June 1997. ACM Press.
23. *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation (TIC'98)*, volume 1473 of *Lecture Notes in Computer Science*, Kyoto, Japan, March 1998.
24. David A. Turner. *Aspects of the Implementation of Programming Languages.* PhD thesis, University of Oxford, 1981.
25. Philip L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.