Shifting the Stage

Staging with Delimited Control

Yukiyoshi Kameyama
University of Tsukuba
kameyama@acm.org

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Rutgers University
ccshan@cs.rutgers.edu

Abstract

It is often hard to write programs that are efficient yet reusable. For example, an efficient implementation of Gaussian elimination should be specialized to the structure and known static properties of the input matrix. The most profitable optimizations, such as choosing the best pivoting or memoization, cannot be expected of even an advanced compiler because they are specific to the domain, but expressing these optimizations directly makes for ungainly source code. Instead, a promising and popular way to reconcile efficiency with reusability is for a domain expert to write code generators.

Two pillars of this approach are types and effects. Typed multilevel languages such as MetaOCaml ensure *safety*: a well-typed code generator neither goes wrong nor generates code that goes wrong. Side effects such as state and control ease *correctness*: an effectful generator can resemble the textbook presentation of an algorithm, as is familiar to domain experts, yet insert *let* for memoization and *if* for bounds-checking, as is necessary for efficiency. However, adding effects blindly renders multilevel types unsound.

We introduce the first two-level calculus with control effects and a sound type system. We give small-step operational semantics as well as a continuation-passing style (CPS) translation. For soundness, our calculus restricts the code generator's effects to the scope of generated binders. Even with this restriction, we can finally write efficient code generators for dynamic programming and numerical methods in direct style, like in algorithm textbooks, rather than in CPS or monadic style.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Design, Languages

Keywords Staged programming, multilevel languages, code generation, mutable state, delimited control, side effects, continuations

1. Introduction

High-performance computing and high-assurance embedded computing often call for programs that are specialized for particular inputs, usages, or processors. Writing such programs by hand is labor-intensive and error-prone. In contrast, code generation offers a promising approach that reconciles modularity and abstraction with efficiency and safety: the programmer can write a generic program that generates specialized code and assures it safe (Hammond and Michaelson 2003; Püschel et al. 2005).

It is attractive to implement such a code generator in a multilevel language (Gomard and Jones 1991; Nielson and Nielson 1988, 1992) such as MetaOCaml (2006; Lengauer and Taha 2006), because a multilevel language offers a principled interface to a code generator for building and composing code fragments and manipulating their binding structure. The interface can guarantee that the generated code is syntactically well-formed, even well-typed and well-scoped. Multilevel languages are thus popular in applications of code generation such as partial evaluation (Gomard and Jones 1991), continuation-passing style (CPS) translation (Danvy and Filinski 1992), embedding domain-specific languages (DSLs) (Czarnecki et al. 2004; Pašalić et al. 2002), and controlling special processors (Elliott 2004; Taha 2005).

In domains whose experts are not well-versed in programming-language research, code generation can only deliver its promise if a code generator is almost as easy for a domain expert to implement as an unspecialized algorithm. Multilevel languages have gone a long way towards this goal, but not enough yet. For example, using a multilevel language to generate specialized code improves the performance and assurance of dynamic programming (Swadi et al. 2006) as well as Gaussian elimination (Carette and Kiselyov 2008), but current multilevel languages leave us with an agonizing tradeoff: our generators must either be written in CPS (Bondorf 1992; Danvy and Filinski 1990, 1992) or monadic style (Swadi et al. 2006), or use delimited control operators (Danvy and Filinski 1990, 1992; Lawall and Danvy 1994) or mutable state (Sumii and Kobayashi 2001). The first choice renders the code generators inscrutable to the typical programmer who learned unspecialized algorithms from a numerical-methods textbook, whereas the second choice voids the multilevel language's guarantee that the generated code is well-formed. In short, we cannot achieve clarity, safety, and efficiency at the same time.

Contributions Motivated by this agony, this paper introduces $\lambda_1^{\oslash}$, the first two-level language with delimited control operators that assures in its type system that all generated code is well-typed and well-scoped. The main innovation of the language is to maintain type soundness by restricting side effects incurred during code generation to the scope of generated binders. We have embedded the language in MetaOCaml, where the restriction has to be checked manually, and implemented it fully in Twelf. The language is a small, simplified variant of λ_{1v}^{α} (Kameyama et al. 2008), and the restriction on effects is also simple. Nevertheless, we can express code generators that perform *let*- and *if*-insertion—including Gaussian elimination and dynamic programming—in direct style rather

PEPM'09, January 19–20, 2009, Savannah, Georgia, USA.

than resorting to CPS or monadic style. We can thus use our new language to create frameworks and embedded DSLs for program generation that application programmers and domain experts can use. Our implementations of $\lambda_1^{\bigcirc}$ and of direct-style code generators, including the examples of the dynamic programming specialization benchmark (Swadi et al. 2005), are all available online at `http://okmij.org/ftp/Computation/staging/README.dr`.

Organization Section 2 illustrates the challenges of code generation using the Gibonacci function in MetaOCaml. Section 3 introduces our new language, shows how its combination of delimited control and staging meets the challenges, and explains its type system. Section 4 proves that the type system is sound and delivers principal types. Section 5 presents a CPS translation for the language. Section 6 discusses related work, and Section 7 concludes.

2. Running example

Our running example is the Gibonacci function in the dynamic programming specialization benchmark (Swadi et al. 2005). It generalizes the Fibonacci function and can be written in OCaml as follows.

```
let rec gib x y n =
  if n = 0 then x else
  if n = 1 then y else
  gib x y (n-1) + gib x y (n-2)
```

There are better ways of computing Gibonacci—after all, there exists a closed formula. The code above, however, is only slightly simpler than the serious examples of dynamic programming found in the benchmark, such as longest common subsequence, binary knapsack, and optimal matrix multiplication ordering. Optimal staging of the longest common subsequence is described in Appendix A; the accompanying code deals with other examples.

To generate specialized versions of `gib` when the argument `n` is statically known, we can write the following MetaOCaml code.

```
(* val gibgen: int code -> int code -> int -> int code *)
let rec gibgen x y n =
  if n = 0 then x else
  if n = 1 then y else
  .<.~(gibgen x y (n-1)) + .~(gibgen x y (n-2))>.
let test_gibgen n =
  .<fun x y -> .~(gibgen .<x>. .<y>. n)>.
```

A pair of *brackets* `.<`*e*`>.` encloses a *future-stage* expression *e*, which is a fragment of generated code. Whereas `1 + 2` is a *present-stage* expression of type `int`, `.<1 + 2>.` is a present-stage *value* of type `int code`, containing the code to add two integers.[1] To combine code values, we use *escapes* `.~`*e* within brackets. The escaped expression *e* is evaluated at the present stage; its result, which must be a code value, is spliced into the enclosing bracket. The inferred type of `gibgen` above describes it as a code generator that takes two code values as arguments (even open code values such as `.<x>.` and `.<y>.`). Brackets and escapes in MetaOCaml are thus equivalent to `next` and `prev` in $\lambda^{\bigcirc}$ (Davies 1996). They are similar to `quasiquote` and `unquote` in Lisp, except a future-stage binder such as `fun x` above generates a new name and binds it in a single operation, so no generator (even if ill-typed) ever produces ill-scoped code. For example, to specialize `gib` to the case of `n` being `5`, we evaluate `test_gibgen 5` to yield the value

```
.<fun x_1 -> fun y_2 ->
   ((((y_2 + x_1) + y_2) + (y_2 + x_1)) +
    ((y_2 + x_1) + y_2))>.
```

[1] This expression actually has the type `('a,int) code` in MetaOCaml, where the type variable `'a` is an *environment classifier* (Taha and Nielsen 2003). Classifiers are not needed in this paper (see §3.1), so we elide them.

in which MetaOCaml generates the names `x_1` and `y_2` fresh. This code value has the type `(int -> int -> int) code`. Besides printing it, MetaOCaml can compile it into independently usable C or Fortran code (Eckhardt et al. 2005) or run it.

2.1 Memoization

The naively specialized `gib` code is patently inefficient like `gib` itself: the computation `y_2 + x_1` is repeated thrice. Gibonacci, as with dynamic programming algorithms, can be greatly sped up by *memoization* (Michie 1968), a form of information propagation (Sørensen et al. 1994).

The most appealing memoization method requires minimal changes in the code. The programmer only needs to rewrite the code to 'open up the recursion':

```
let gib x y self n =
  if n = 0 then x else
  if n = 1 then y else
  self (n-1) + self (n-2)
```

The function `gib` is no longer recursive. It receives an extra argument `self` for the recursive instance of itself. We 'tie the knot' with the explicit fixpoint combinator `y_simple`:

```
let rec y_simple f n = f (y_simple f) n
```

Evaluating `y_simple (gib 1 1) 5` yields `8` in the same inefficient way as before. To add memoization, we switch to a different fixpoint combinator `y_memo_m`, but keep the *same* `gib` code (McAdam 2001).

```
let y_memo_m f n =
  let table = ref (empty ()) in
  let rec memo n =
    match (lookup n !table) with
    | None -> let v = f memo n in
              (table := ext !table n v; v)
    | Some v -> v
  in f memo n
```

Just as the definition of `gib` closely follows how a textbook might describe the Gibonacci function, the definition of `y_memo_m` closely follows how a textbook might describe memoization. We assume a finite-map data-type with the operations `empty ()` to create the empty map, `lookup n table` to locate a value associated with the integer key `n`, and `ext table n v` to return a new map extending `table` by associating the key `n` to the value `v`. Now we can evaluate `y_memo_m (gib 1 1) 30`, which finishes much faster than `y_simple (gib 1 1) 30`.

This memoization method is appealing because it relegates memoization to a library of fixpoint combinators and does not distort the code of the algorithm (`gib` in our case). In a support library for dynamic programming (which was the goal of Swadi et al. (2006)), this method allows application programmers to write natural and modular code, implementing memoization strategies separately from functions to memoize.

However, this simple method does not work when specializing `gib`, for two reasons. First, the memoizing combinator `y_memo_m` must use mutation so that the two sibling calls to `self` in `gib`, with no explicit data flow between them, could reuse each other's computation by sharing the same memoization table. When specializing memoized `gib`, the table stores code values. Alas, blindly combining mutation and staging leads to *scope extrusion*, a form of type unsoundness. For example, evaluating the expression

```
let r = ref .<1>. in
.<fun y -> .~(r := .<y>.; .<()>.)>.;  !r
```

in MetaOCaml yields `.<y_1>.`, a code fragment that contains an unbound variable and is thus ill-formed. Mutation and other ef-

fects such as exceptions and control defeat MetaOCaml's guarantee that the generated code is well-formed and well-typed. Therefore, MetaOCaml does not assure that `y_memo_m` is safe to use, even though in this case it is.

The most profitable optimizations in each domain often involve a different set of combinators—for memoizing results, pivoting matrices, simplifying arithmetic, and so on (Cohen et al. 2006). Therefore, a language for code generation should empower not just a programming-language researcher but also an application programmer to create combinator libraries, including those using mutation. For such wide use of side effects, the language should assure type soundness, especially the absence of scope extrusion.

2.2 Let-insertion

Besides the risk of scope extrusion, there is a second, deeper problem: code duplication. Suppose we stage `gib` with open recursion:

```
let sgib x y self n =
  if n = 0 then x else
  if n = 1 then y else
  .<.~(self (n-1)) + .~(self (n-2))>.
```

Now `.<fun x y -> .~(y_memo_m (sgib .<x>. .<y>.) 5)>.` produces the same inefficient specialized `gib` as before, with the computation `y_2 + x_1` repeated thrice. Whereas code generation is memoized, the generated code does not memoize (Bondorf and Danvy 1991). For example, we want `y_memo_m (sgib .<x>. .<y>.) 4` to return `.<let t = y + x in let u = t + y in u + t>.`, where no computation is duplicated. In this desired output, `self 2` should contribute the binding and use of `u`, and `self 3` those of `t`, but these contributions are not code fragments—subexpressions—that can be spliced in by escapes.

One way to insert `let` as desired is to write the code generator in CPS or monadic style (Bondorf 1992; Danvy and Filinski 1990, 1992; Swadi et al. 2006). The memoized calls to the code generator can then share the memoization table and insert `let`-bindings as necessary, without risking scope extrusion. In monadic style, the function `gib` takes the following form (Swadi et al. 2005).

```
let sgib_c x y self n =
  if n = 0 then ret x else
  if n = 1 then ret y else
  bind (self (n-2)) (fun r1 ->
  bind (self (n-1)) (fun r2 ->
  ret .<.~r2 + .~r1>.))
```

We omit the definitions of the monad operations `ret` and `bind` and of the memoizing combinator that applies to `sgib_c`. All this code no longer resembles textbook algorithms, so it has lost its appeal of simplicity. Syntactic sugar for monadic code (Carette and Kiselyov 2008; Peyton Jones 2003; Wadler 1992) reduces the clutter but not the need to name intermediate results such as `r1` and `r2` above. In practice (for example, to generate Gaussian-elimination code), monadic style imposes a severe notational overhead (Carette and Kiselyov 2008) that alienates application programmers and obstructs our quest to help end users specialize their code.

2.3 If-insertion

We have seen that let-insertion is necessary to avoid code duplication in practical code generators and requires the unappealing use of CPS or monadic style. A similar pattern is *if-insertion* (or *assertion insertion*), illustrated below. The code generator `gen` invokes an auxiliary generator `retrieve` to extract the result of a complex computation on a working array.

```
let gen retrieve =
  .<fun array n -> (complex computation on array);
                   .~(retrieve .<array>. .<n>.)>.
```

The auxiliary generator `retrieve` receives two code values from `gen`, which represent an array and an index into it. The code generated by `retrieve` could just read the `n`-th element of `array`.

```
let retrieve array n = .< (.~array) . (.~n) >.
```

We would like, however, to check that `n` is in the bounds of `array`. We could insert the bounds check right before the array access:

```
let retrieve array n =
  .<assert (.~n >= 0 && .~n < Array.length .~array);
    (.~array) . (.~n)>.
```

Such a check is too late: we want the check right after the array and the index are determined, before any complex computations commence. We wish the generator `gen` to yield

```
.<fun array_1 -> fun n_2 ->
    assert (n_2 >= 0 && n_2 < Array.length array_1);
    (complex computation on array_1);
    array_1.(n_2)>.
```

Again it seems impossible for `retrieve` to splice in the `assert` far from the escape in `gen`. Again this difficulty can be overcome by writing generators in CPS or monadic style, which looks foreign to the application programmer.

2.4 Delimited control and its risk of scope extrusion

Lawall and Danvy (1994) show how to use Danvy and Filinski's *delimited control operators* `shift` and `reset` (1989, 1990, 1992) to perform let- and if-insertion in the familiar direct style, by effectively hiding trivial uses of continuations as in `sgib_c` above. Since delimited control operators are available in MetaOCaml (Kiselyov 2006), we can build a memoizing staged fixpoint combinator `y_ms` with this technique, so that evaluating

```
.<fun x y -> .~(top_fn (fun _ ->
                    y_ms (sgib .<x>. .<y>.) 5))>.
```

—using the *same* direct-style `sgib` in §2.2—gives the ideal code

```
.<fun x_1 -> fun y_2 ->
    let z_3 = y_2 in
    let z_4 = x_1 in
    let z_5 = (z_3 + z_4) in
    let z_6 = (z_5 + z_3) in
    let z_7 = (z_6 + z_5) in (z_7 + z_6)>.
```

without duplicating computations. (We describe `y_ms` and `top_fn` in §3.4.) The same delimited control operators let us accomplish if-insertion using the intuitive version of `gen` in §2.3.

Delimited control, however, is a side effect whose unrestricted use poses the risk of scope extrusion. For example, the expression

```
top_fn (fun _ -> .<fun x y -> .~(
                    y_ms (sgib .<x>. .<y>.) 5)>.)
```

is well-typed in MetaOCaml with `shift` and `reset` added, but it evaluates to the following code value, which disturbingly uses the variables `y_2` and `x_1` unbound.

```
.<let z_3 = y_2 in
  let z_4 = x_1 in
  let z_5 = (z_3 + z_4) in
  let z_6 = (z_5 + z_3) in
  let z_7 = (z_6 + z_5) in
  fun x_1 -> fun y_2 -> (z_7 + z_6)>.
```

3. Combining staging and control safely

To eliminate the risk of scope extrusion just demonstrated, we propose a simple restriction: informally, we place an implicit present-stage `reset` under each future-stage binder. Any escape under a

$$
\begin{array}{lll}
\text{Variables} & & x, y, z, f, k, n \\
\text{Expressions} & e ::= & i \mid e + e \mid \lambda x.e \mid \mathsf{fix} \mid ee \mid (e,e) \mid \mathsf{fst} \mid \mathsf{snd} \\
& & \mid \mathsf{ifz}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \text{出} \mid \{e\} \mid \langle e \rangle \mid {\sim}e \mid x
\end{array}
$$

Figure 1. Syntax of $\lambda_1^{\oslash}$

$$
\begin{array}{lll}
\text{Values} & v^0 ::= & i \mid \lambda x.e \mid \mathsf{fix} \mid (v^0, v^0) \mid \mathsf{fst} \mid \mathsf{snd} \mid \text{出} \mid \langle v^1 \rangle \mid x \\
& v^1 ::= & i \mid v^1 + v^1 \mid \lambda x.v^1 \mid \mathsf{fix} \mid v^1 v^1 \mid (v^1, v^1) \\
& & \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{ifz}\ v^1\ \mathsf{then}\ v^1\ \mathsf{else}\ v^1 \mid \text{出} \mid \{v^1\} \mid x \\
\text{Frames} & F^0 ::= & \Box + e \mid v^0 + \Box \mid \Box e \mid v^0 \Box \mid (\Box, e) \mid (v^0, \Box) \\
& & \mid \mathsf{ifz}\ \Box\ \mathsf{then}\ e\ \mathsf{else}\ e \\
& F^1 ::= & \Box + e \mid v^1 + \Box \mid \Box e \mid v^1 \Box \mid (\Box, e) \mid (v^1, \Box) \\
& & \mid \mathsf{ifz}\ \Box\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{ifz}\ v^1\ \mathsf{then}\ \Box\ \mathsf{else}\ e \\
& & \mid \mathsf{ifz}\ v^1\ \mathsf{then}\ v^1\ \mathsf{else}\ \Box \mid \{\Box\}
\end{array}
$$

Delimited contexts

$$
\begin{array}{lll}
D^{00} & ::= & \Box \mid D^{00}[F^0] \mid D^{01}[{\sim}\Box] \\
D^{10} & ::= & D^{10}[F^0] \mid D^{11}[{\sim}\Box] \\
D^{01} & ::= & D^{01}[F^1] \mid D^{00}[\langle\Box\rangle] \\
D^{11} & ::= & \Box \mid D^{11}[F^1] \mid D^{10}[\langle\Box\rangle]
\end{array}
$$

$$
\begin{array}{lll}
\text{Contexts} & C^0 ::= & D^{00} \mid C^0[\{D^{00}\}] \mid C^1[\lambda x.D^{10}] \\
& C^1 ::= & D^{01} \mid C^0[\{D^{01}\}] \mid C^1[\lambda x.D^{11}]
\end{array}
$$

Figure 2. Values and contexts

future-stage binder thus incurs no effect *observable* outside the binder's scope. This restriction turns out not to preclude memoization, let-insertion, and if-insertion—which application programmers can now implement safely (without scope extrusion) and naturally (in direct style).

In this section, we detail our proposal by introducing a language with staging and control effects that builds in this restriction and, as we prove, prevents scope extrusion. Our language $\lambda_1^{\oslash}$ models a subset of MetaOCaml extended with delimited control operators. Figure 1 shows the syntax: it features integer literals i and their arithmetic $+$, λ-abstractions and their applications, pairs (e_1, e_2) and their projections fst and snd. We write let $x = e_1$ in e_2 as shorthand for $(\lambda x.e_2)e_1$. The conditional ifz e then e_1 else e_2 reduces to e_1 if e is zero, and to e_2 otherwise. The constant fix is the applicative fixpoint combinator. As usual, we identify α-equivalent terms and assume Barendregt's variable convention. The operational semantics of these constructs is standard and call-by-value, as defined in Figures 2 and 3 in terms of small steps $\rightsquigarrow$ and evaluation contexts C^0. In the subsections below, we explain the staging forms $\langle e \rangle$ and ${\sim}e$, the level superscripts 0 and 1, the delimited control forms 出 and $\{e\}$, and their interaction and typing.

We have implemented the language in Twelf, where the efficient Gibonacci generator can run. Unlike our Twelf implementation, MetaOCaml does not currently build in our restriction, so we must manually examine each escape under a future-stage binder and check that it has no observable control effect. It is possible to automate this check, either by extending MetaOCaml's type checker or by building a separate tool like Leroy and Pessaux's exception checker (2000).

3.1 Staging

As described in §2, our staging facility is comprised of brackets $\langle e \rangle$ and escapes ${\sim}e$. The staging level of an expression affects whether it is a value and how a non-value is decomposed into a

$$
\begin{array}{rcll}
C^0\left[i_1 + i_2\right] & \rightsquigarrow & C^0\left[i_1 + i_2\right] & (+) \\
C^0\left[(\lambda x.e)\, v^0\right] & \rightsquigarrow & C^0\left[e[x := v^0]\right] & (\beta_v) \\
C^0\left[\mathsf{fix}\ v^0\right] & \rightsquigarrow & C^0\left[\lambda x.v^0(\mathsf{fix}\ v^0)x\right] & \\
C^0\left[\mathsf{fst}\ (v_1^0, v_2^0)\right] & \rightsquigarrow & C^0\left[v_1^0\right] & \\
C^0\left[\mathsf{snd}\ (v_1^0, v_2^0)\right] & \rightsquigarrow & C^0\left[v_2^0\right] & \\
C^0\left[\mathsf{ifz}\ 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\right] & \rightsquigarrow & C^0\left[e_1\right] & \\
C^0\left[\mathsf{ifz}\ i\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\right] & \rightsquigarrow & C^0\left[e_2\right] \qquad \text{if } i \neq 0 & \\
C^0\left[\{v^0\}\right] & \rightsquigarrow & C^0\left[v^0\right] & (\{\}) \\
C^1\left[{\sim}\langle v^1 \rangle\right] & \rightsquigarrow & C^1\left[v^1\right] & ({\sim}) \\
C^0\left[\{D^{00}[\text{出}v^0]\}\right] & \rightsquigarrow & C^0\left[\{v^0(\lambda x.\{D^{00}[x]\})\}\right] & (\text{出}^0) \\
C^1\left[\lambda x.D^{10}[\text{出}v^0]\right] & \rightsquigarrow & C^1\left[\lambda x.{\sim}\{\langle D^{10}[\text{出}v^0]\rangle\}\right] & (\text{出}^1)
\end{array}
$$

Figure 3. Operational semantics: small-step reduction $e \rightsquigarrow e'$

context and a redex (Taha 2000). Our calculus has only two levels, present-stage and future-stage.[2] They correspond to two evaluation 'modes', reduction and code-building. To notate these levels, we put the superscripts 0 and 1 on metavariables, such as values and contexts in Figure 2.[3] Brackets enclose a present-stage expression to form a future-stage expression, whereas escapes do the opposite. In particular, present-stage values v^0 include code fragments $\langle v^1 \rangle$, which are bracketed expressions containing no escapes.

A present-stage context C^0 can be *plugged* (that is, have its *hole* $\Box$ replaced) with a present-stage expression e to form a complete program $C^0[e]$, whereas a future-stage context C^1 can be plugged with a future-stage expression. As is usual in a multilevel language, these contexts may contain future-stage bindings introduced by λ, so present-stage evaluation can occur in the body of a future-stage abstraction. Contexts C^i are defined by composing *delimited contexts* D^{ij}, which can be plugged with a level-j expression to form a level-i expression. Delimited contexts are in turn defined by composing *frames* F^i, which can be plugged with a level-i expression to form a slightly larger level-i expression. In a degenerate language with neither staging nor delimited control, the superscripts would all be 0, and a context C^0 and a delimited context D^{00} would both be just a sequence of frames F^0.

If for a moment we disregard delimited control operators (to be explained in §3.2), then the language $\lambda_1^{\oslash}$ is almost the same as our earlier two-level staged calculus λ_{1v}^{α} (Kameyama et al. 2008), but without cross-stage persistence (CSP) and without the operation `run` to execute generated code. It can thus be regarded as Davies's $\lambda^{\bigcirc}$ (1996) restricted to two levels. It is also similar to Nielson and Nielson's (1988, 1992) and Gomard and Jones's (1991) two-level λ-calculi (though the latter does not type-check generated code).

Excluding `run` from our language makes it simpler to implement (because the run-time system need not include a compiler and dynamic linker) and to prove sound (because the type system need not include environment classifiers (Taha and Nielsen 2003) to prevent attempts to `run` open code). The inability to `run` generated code in the language may appear severe, but it is no different from the inability of the typical compiler (especially cross-compiler) to load and run any generated code in the compiler process itself. A code generator written in $\lambda_1^{\oslash}$ cannot run any generated code on the fly to test it, but the generated code is guaranteed to be well-typed and can be saved to a source file to be compiled and run in a separate process. That is enough for the intended use for $\lambda_1^{\oslash}$, namely

[2] More levels would clutter the notation but are probably not hard to add.

[3] Our Twelf formalization marks each expression as well with its level, but we suppress those superscripts in this paper.

implementing DSL 'compilers' that generate families of optimized library routines—such as Gaussian elimination (Carette and Kiselyov 2008), Fast Fourier Transform (Frigo and Johnson 2005; Kiselyov and Taha 2005), linear signal processing (Püschel et al. 2005), and embedded code (Hammond and Michaelson 2003)—to be used in applications other than the generator itself.

The lack of CSP in $\lambda_1^{\oslash}$ means that there is no 'lift' operation to *uniformly* convert a present-stage value of any type to some future-stage code that evaluates to that value. However, $\lambda_1^{\oslash}$ can express lifting at specific data types—integers, pairs of integers, and such. Whereas CSP is important when using `run` (Taha and Nielsen 2003), it is unnecessary for mere code generation. It can even be harmful if unrestricted, because a generated library routine ought to be usable without the generator present.

3.2 Delimited control

Delimited control is realized by the *control delimiter* $\{\ \}$ (pronounced 'reset') and the constant $\mathrm{\Psi}$ (pronounced 'shift').

When $\mathrm{\Psi}$ is not used, the expression $\{e\}$ (pronounced 'reset e') is evaluated like e, as if $\{e\}$ were just shorthand for $(\lambda x.x)e$. We specify this behavior by allowing contexts C^0 to include resets $\{\ \}$.

The constant $\mathrm{\Psi}$ is supposed to be applied to a function value, say v^0. When $\mathrm{\Psi} v^0$ is evaluated, it captures the part of the current evaluation context C^0 up to the nearest dynamically enclosing delimiter. We call this part a *delimited context* D^{00}; unlike C^0, it does not include reset. As the $\mathrm{\Psi}^0$ rule in Figure 3 shows, the subexpression $D^{00}[\mathrm{\Psi} v^0]$ reduces to the application $v^0(\lambda x.\{D^{00}[x]\})$, reifying the captured delimited context D^{00} as the abstraction $\lambda x.\{D^{00}[x]\}$.

We illustrate delimited control by using it to simulate mutable state (Filinski 1994; Kiselyov et al. 2006). We define the terms

$$\mathsf{const} = \lambda y.\lambda z.y,\ \ \mathsf{get} = \mathrm{\Psi}(\lambda k.\lambda z.kzz),\ \ \mathsf{put} = \lambda z'.\mathrm{\Psi}(\lambda k.\lambda z.kz'z').$$

The reduction sequence below illustrates how const and get work.

$$\begin{aligned}
\{\mathsf{const}\ (\mathsf{get}+40)\}\ 2 &\rightsquigarrow_{\mathrm{\Psi}^0} \{(\lambda k.\lambda z.kzz)(\lambda x.\{\mathsf{const}(x+40)\})\}\ 2\\
&\rightsquigarrow_{\beta_v} \{\lambda z.(\lambda x.\{\mathsf{const}(x+40)\})zz\}\ 2\\
&\rightsquigarrow_{\{\}} (\lambda z.(\lambda x.\{\mathsf{const}(x+40)\})zz)\ 2\\
&\rightsquigarrow_{\beta_v} (\lambda x.\{\mathsf{const}(x+40)\})\ 2\ 2\\
&\rightsquigarrow_{\beta_v} \{\mathsf{const}(2+40)\}\ 2
\end{aligned}$$

The first step replaces get and its delimited context $\mathsf{const}(\square+40)$ by an application of $\lambda k.\lambda z.kzz$ to the function $\lambda x.\{\mathsf{const}(x+40)\}$. The latter function is precisely the captured delimited context, enclosed in reset and reified as a function. This single step can be decomposed into a sequence of finer-grain reductions in which the $\mathrm{\Psi}$-application *bubbles up* and builds up the delimited context by local rewriting (Felleisen and Friedman 1987; Parigot 1992).

Comparing the initial and final programs in this reduction sequence shows that its net result is to replace the expression get with 2. It is as if the number 2 were stored in a cell and accessed by get in the program $\mathsf{get}+40$. The reductions continue to a value.

$$\{\mathsf{const}\ 42\}\ 2 \rightsquigarrow_{\beta_v} \{\lambda z.42\}\ 2 \rightsquigarrow_{\{\}} (\lambda z.42)\ 2 \rightsquigarrow_{\beta_v} 42$$

We can mutate the state: the term $\mathsf{put}(\mathsf{get}+1)$ increments the number in the cell and returns the new number.

$$\begin{aligned}
\{\mathsf{const}\ (\mathsf{put}(\mathsf{get}+1)+\mathsf{get})\}\ 2 &\rightsquigarrow^+ \{\mathsf{const}\ (\mathsf{put}(2+1)+\mathsf{get})\}\ 2\\
&\rightsquigarrow_+ \{\mathsf{const}\ (\mathsf{put}\ 3+\mathsf{get})\}\ 2\\
&\rightsquigarrow^+ (\lambda x.\{\mathsf{const}\ (x+\mathsf{get})\})\ 3\ 3\\
&\rightsquigarrow_{\beta_v} \{\mathsf{const}\ (3+\mathsf{get})\}\ 3
\end{aligned}$$

This sequence of reductions replaces the term $\mathsf{put}(\mathsf{get}+1)$ with 3 and at the same time puts the new value 3 outside the reset. The result reduces to $\{\mathsf{const}(3+3)\}\ 3$ and eventually 6. In general, the term $\{\mathsf{const}\ e\}v^0$ behaves as if the expression e were executed in the 'context' of a mutable cell initialized to v^0. Inside e, occurrences of get retrieve the current value of the cell, and a subterm of the form $\mathrm{\Psi}(\lambda k.\lambda z.kz'z')$ assigns z' to the cell.

Although our language has no mutable state, we have just emulated it using delimited control. We can therefore treat a memoization table as a piece of mutable state, express the memoizing fixpoint combinator `y_memo_m` (see the accompanying code in `circle-shift.elf`), and use it to transparently memoize `gib` or another dynamic-programming algorithm in $\lambda_1^{\oslash}$.

For the purpose of code generation, emulating mutable state by delimited control brings two benefits. First, our core calculus is smaller and its soundness is simpler to prove. Second, the delimited nature of our control operations lets us limit the lifetime (or *dynamic extent* (Moreau 1998)) of the mutable state. In other words, we can make sure that a mutable cell is only accessed or updated during the evaluation of a particular subexpression. To prevent scope extrusion, it is crucial that our language provide this assurance both in the operational semantics (described in §3.3 below) and in the type system (described in §3.5 below). Although optimizing compilers of imperative languages can determine the extent of mutation by control-flow analyses, the results of the analyses are not expressed in the language or exposed to the programmer.

3.3 Staging and delimited control, without scope extrusion

At first glance, it appears straightforward to combine staging and delimited control. For example, the emulation of mutable state by delimited control appears to work as explained in §3.2 even if we store code values rather than integers in the mutable state and access them within escapes. For example, the following example reuses a code value using const, get, and put.

$$\begin{aligned}
&\{\mathsf{const}\ \langle{\sim}(\mathsf{put}\langle 8+5\rangle)+{\sim}\mathsf{get}\rangle\}\ \langle 0\rangle\\
&\quad\rightsquigarrow_{\beta_v} \{\mathsf{const}\ \langle{\sim}(\mathrm{\Psi}(\lambda k.\lambda z.k\langle 8+5\rangle\langle 8+5\rangle))+{\sim}\mathsf{get}\rangle\}\ \langle 0\rangle\\
&\quad\rightsquigarrow_{\mathrm{\Psi}^0} \{(\lambda k.\lambda z.k\langle 8+5\rangle\langle 8+5\rangle)(\lambda x.\{\mathsf{const}\langle{\sim}x+{\sim}\mathsf{get}\rangle\})\}\ \langle 0\rangle\\
&\quad\rightsquigarrow^+ \{\mathsf{const}\ \langle{\sim}\langle 8+5\rangle+{\sim}\mathsf{get}\rangle\}\ \langle 8+5\rangle\\
&\quad\rightsquigarrow_{\sim} \{\mathsf{const}\ \langle(8+5)+{\sim}\mathsf{get}\rangle\}\ \langle 8+5\rangle \rightsquigarrow^+ \langle(8+5)+(8+5)\rangle
\end{aligned}$$

Like in §3.2, $\mathsf{put}\langle 8+5\rangle$ assigns $\langle 8+5\rangle$ to the mutable cell, so get later is replaced by $\langle 8+5\rangle$. The final result is a piece of generated code that, when evaluated in the future stage, will add 5 to 8 twice. The only apparent difference between this emulation of mutable state and the examples in §3.2 is that captured delimited contexts, such as $\lambda x.\{\mathsf{const}\langle{\sim}x+{\sim}\mathsf{get}\rangle\}$ in the second reduction above, may span across brackets and escapes.

We now confront the two problems described in §2 that arise when memoizing code generators. The first problem is the risk of scope extrusion, which can happen when we store a code value that uses a bound variable then splice the code value outside the scope of the variable. Let us try to trigger scope extrusion in $\lambda_1^{\oslash}$:

$$\{\mathsf{const}\ (\mathsf{let}\ x = \langle\lambda y.{\sim}(\mathsf{put}\langle y\rangle)\rangle\ \mathsf{in}\ \mathsf{get})\}\ \langle 0\rangle$$

If $\mathsf{put}\langle y\rangle$ above were to assign the code value $\langle y\rangle$ to the mutable cell and get were to retrieve that code value, then this program would generate the ill-scoped code $\langle y\rangle$. Fortunately, put cannot reach the mutable cell because the future-stage binder λy stands in the way. In Figure 2, this restriction is built into the definition of delimited contexts, which excludes not only present-stage resets but also future-stage binders. Our attempt at scope extrusion thus fails; in fact, the type system in §3.5 below rejects it statically. We prove that scope extrusion is impossible in §4.

The $\mathrm{\Psi}^1$ rule in Figure 3 shows that a future-stage binder acts as a control delimiter just as a present-stage reset does: a future-stage abstraction $\lambda x.e$ implicitly expands to $\lambda x.{\sim}\{\langle e\rangle\}$. In this regard, staging and delimited control are not orthogonal: whenever staging brings evaluation under λ, any side effect (in particular the

lifetime of mutable state) must also stay under λ. Our language prevents different future-stage scopes from sharing a memoization table, because doing so risks scope extrusion.

The second problem with memoizing code, described in §2.2, is that the generated code duplicates computations such as $8+5$ above. Armed with delimited control operators, we can now solve this problem by inserting let in the generated code without writing our code generator in CPS or monadic style. To illustrate this key idea (due to Lawall and Danvy (1994) in an untyped setting), we define the following alternative to put.

$$\mathsf{put}' = \lambda z'.\, \text{出}(\lambda k.\, \lambda z.\, \langle \mathsf{let}\ x = {\sim}z'\ \mathsf{in}\ {\sim}(k\langle x\rangle\langle x\rangle)\rangle)$$

Using this put' instead of put, it is easy to insert let in the generated code to avoid duplicating computations.

$$\begin{aligned}
&\{\mathsf{const}\ \langle {\sim}(\mathsf{put}'\langle 8+5\rangle) + {\sim}\mathsf{get}\rangle\}\ \langle 0\rangle \\
&\quad \rightsquigarrow_{\beta_v}\ \{\mathsf{const}\ \langle {\sim}(\text{出}(\lambda k.\, \lambda z.\, \langle \mathsf{let}\ x = {\sim}\langle 8+5\rangle\ \mathsf{in}\ {\sim}(k\langle x\rangle\langle x\rangle)\rangle)) \\
&\qquad\qquad + {\sim}\mathsf{get}\rangle\}\ \langle 0\rangle \\
&\quad \rightsquigarrow_{\text{出}^0}\ \{(\lambda k.\, \lambda z.\, \langle \mathsf{let}\ x = {\sim}\langle 8+5\rangle\ \mathsf{in}\ {\sim}(k\langle x\rangle\langle x\rangle)\rangle) \\
&\qquad (\lambda x.\, \{\mathsf{const}\langle {\sim}x + {\sim}\mathsf{get}\rangle\})\}\ \langle 0\rangle \\
&\quad \rightsquigarrow^+\ \langle \mathsf{let}\ x = 8+5\ \mathsf{in}\ {\sim}(\{\mathsf{const}\ \langle x + {\sim}\mathsf{get}\rangle\}\ \langle x\rangle)\rangle \\
&\quad \rightsquigarrow^+\ \langle \mathsf{let}\ x = 8+5\ \mathsf{in}\ x + x\rangle
\end{aligned}$$

Instead of storing any code for reuse that may contain a complex computation, put' inserts a let to bind the result of the computation to a new variable ($x = 8+5$ above) that takes scope over the entire generated expression, then stores just the variable. The generated code performs the computation only once and can reuse the result.

3.4 Payoff: safe and efficient code generation in direct style

We have shown how to simulate mutable state and perform let-insertion using delimited control. Using these techniques, we have built the desired memoizing staged fixpoint combinator y_ms (see `circle-shift.elf` for the complete code and tests). Roughly,[4] y_ms has the type $((\mathsf{int} \to \langle \mathsf{int}\rangle) \to \mathsf{int} \to \langle \mathsf{int}\rangle) \to \mathsf{int} \to \langle \mathsf{int}\rangle$. As this type suggests, this combinator should be applied to a code generator with open recursion, whose first argument is the recursive instance of itself, and whose second argument is a present-stage integer on which to specialize and recur. For example, recall the `sgib` function of §2.2:

$$\begin{aligned}
\mathsf{sgib} = \lambda x.\, \lambda y.\, \lambda \mathit{self}.\, \lambda n.\, &\mathsf{ifz}\ n\ \mathsf{then}\ x\ \mathsf{else} \\
&\mathsf{ifz}\ n-1\ \mathsf{then}\ y\ \mathsf{else} \\
&\langle {\sim}(\mathit{self}\,(n-1)) + {\sim}(\mathit{self}\,(n-2))\rangle
\end{aligned}$$

To specialize the Gibonacci function to $n = 5$, we evaluate[5]

$$\langle \lambda x.\, \lambda y.\, {\sim}(\{\mathsf{const}\ (\mathsf{y\_ms}\,(\mathsf{sgib}\,\langle x\rangle\langle y\rangle)\,5)\}\ \mathsf{empty})\rangle$$

(where empty is the empty memoization table) to obtain

$$\begin{aligned}
\langle \lambda x.\, \lambda y.\, &\mathsf{let}\ z_3 = y\ \mathsf{in}\ \mathsf{let}\ z_4 = x\ \mathsf{in}\ \mathsf{let}\ z_5 = z_3 + z_4\ \mathsf{in} \\
&\mathsf{let}\ z_6 = z_5 + z_3\ \mathsf{in}\ \mathsf{let}\ z_7 = z_6 + z_5\ \mathsf{in}\ z_7 + z_6\rangle.
\end{aligned}$$

This result is the ideal promised in §2.4—a linear sequence of operations without any code duplication.

Figure 4 shows our definition of y_ms. This fixpoint combinator simulates mutable state to maintain a memoization table that maps integer keys to previously generated code values. Therefore, it uses the table operations empty, lookup, and ext specified in §2.1, which are trivial to implement. Whereas `lookup` in §2.1 returns a value of type int option, our language $\lambda_1^{\bigcirc}$ does not include option, so we emulate the sum type τ option by a product type

$$\begin{aligned}
\mathsf{y_ms} = \lambda f.\, f(\mathsf{fix}\ (\lambda \mathit{self}.\, \lambda n.\, &\\
&\mathsf{let}\ x = \text{出}(\lambda k.\, \lambda \mathit{table}.\, k\ (\mathsf{lookup}\ n\ \mathit{table})\ \mathit{table})\ \mathsf{in} \\
&\mathsf{ifz}\ \mathsf{fst}\ x \\
&\mathsf{then}\ \mathsf{let}\ y = f\ \mathit{self}\ n\ \mathsf{in} \\
&\quad \text{出}(\lambda k.\, \lambda \mathit{table}.\, \langle \mathsf{let}\ z = {\sim}y\ \mathsf{in}\ {\sim}(k\langle z\rangle(\mathsf{ext}\ \mathit{table}\ n\ \langle z\rangle))\rangle) \\
&\mathsf{else}\ \{\mathsf{snd}\ x\ 0\}))
\end{aligned}$$

Figure 4. The memoizing staged fixpoint combinator y_ms

$(\mathsf{int}, \mathsf{int} \to \tau)$: the variant `None` is represented as $(0, \mathsf{fix}\ \lambda f.\, f)$ and the variant `Some` x as $(1, \lambda z.\, x)$.

When y_ms is applied to a user function f and that function invokes *self* on an integer argument n, our combinator retrieves the current state of the memoization table to check if code has been generated for n already. The lookup result (a pair) is bound to the variable x in Figure 4. If $\mathsf{fst}\ x$ is zero, meaning n is new, then the combinator invokes f to generate an expression y for n, binds y to a new future-stage variable z, and updates the memoization table to map n to $\langle z\rangle$. If the lookup succeeds (the last line of the code), the combinator returns the found value without invoking f.

In this way, we have successfully specialized the Gibonacci function in direct style as well as Gaussian elimination and Swadi et al.'s (2005) other examples. These successes show that our language is expressive enough for practical applications, despite not allowing delimited control to reach beyond any binder.

If-insertion is also within reach. To use a simpler example than in §2.3, suppose that gen is a code generator in $\lambda_1^{\bigcirc}$ of the form $\lambda f.\, \langle \lambda n.\, e + {\sim}(f\langle n\rangle)\rangle$, where e is some complex computation. The argument f is an auxiliary generator, a function from code to code. Suppose that the code produced by f only makes sense if the future-stage argument n is nonzero—perhaps $f\langle n\rangle$ computes the inverse of n. We would like the generated code to check if n is nonzero before evaluating e. To achieve this goal, we can define f to be

$$\lambda n.\, \text{出}(\lambda k.\, \langle \mathsf{ifz}\ {\sim}n\ \mathsf{then}\ \mathit{fail}\ \mathsf{else}\ {\sim}(k\langle \mathit{inverse}\ {\sim}n\rangle)\rangle).$$

Passing this auxiliary generator to gen produces the desired code

$$\langle \lambda n.\, \mathsf{ifz}\ n\ \mathsf{then}\ \mathit{fail}\ \mathsf{else}\ e + \mathit{inverse}\ n\rangle.$$

The complex expression e will not be evaluated if n turns out zero.

3.5 Type system

Figure 5 displays the type system of our language $\lambda_1^{\bigcirc}$. It combines a simplification of Danvy and Filinski's type system for delimited control (1989) and a simplification of Davies's type system for staging (1996) in a sound but not orthogonal way.

The types τ of $\lambda_1^{\bigcirc}$ are the base type int, arrow types $\tau \to \tau'/\tau_0$, product types (τ, τ'), and code types $\langle \upsilon/\upsilon_0\rangle$. The type system is monomorphic like the simply-typed λ-calculus; we include type variables α only to state that our language has principle typings. Without impredicative effect polymorphism (Asai and Kameyama 2007), in order to write the desired code as in §3.4, we are forced to build product types into the language rather than Church-encode them. (This task is not difficult; sum types would have sufficed too.)

As a two-level language, $\lambda_1^{\bigcirc}$ operates on code values in the present stage only. Hence, we require the types of future-stage expressions to be *flat*, that is, to contain no code types. Types with nested brackets such as $\langle\langle \mathsf{int}/\mathsf{int}\rangle/\mathsf{int}\rangle$ are thus excluded along with terms such as $\langle\langle 42\rangle\rangle$. A type environment Γ is a set of associations $x : \tau$ of present-stage variables x with general types τ and associations $\langle x : \upsilon\rangle$ of future-stage variables x with flat types υ.

There are two judgment forms, one that assigns general types to present-stage expressions and one that assigns flat types to future-stage expressions. Both forms include *answer types* to track the control effects that may occur: in a present judgment $\Gamma \vdash e : \tau\ ;\ \tau_0$,

[4] We suppress effect annotations in types until §3.5, where we introduce the type system formally.

[5] This example reveals that `top_fn` in §2.4 is $\lambda z.\, \{\mathsf{const}\,(z\,0)\}\,\mathsf{empty}$.

Type variables α Types $\tau ::= \mathsf{int} \mid \tau \to \tau'/\tau_0 \mid \langle \upsilon/\upsilon_0 \rangle \mid (\tau, \tau') \mid \alpha$ Present judgments $\Gamma \vdash e : \tau \,;\, \tau_0$

Flat type variables β Flat types $\upsilon ::= \mathsf{int} \mid \upsilon \to \upsilon'/\upsilon_0 \mid (\upsilon, \upsilon') \mid \beta$ Future judgments $\Gamma \vdash e : \upsilon \,;\, \tau_0 \,;\, \upsilon_0$

Environments $\Gamma ::= [] \mid \Gamma, x:\tau \mid \Gamma, \langle x:\upsilon \rangle$

$$\frac{}{\Gamma \vdash i : \mathsf{int} \,;\, \tau_0 \;[;\, \upsilon_0]} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \,;\, \tau_0 \;[;\, \upsilon_0] \quad \Gamma \vdash e_2 : \mathsf{int} \,;\, \tau_0 \;[;\, \upsilon_0]}{\Gamma \vdash e_1 + e_2 : \mathsf{int} \,;\, \tau_0 \;[;\, \upsilon_0]} \qquad \frac{\Gamma, x:\tau \vdash e : \tau' \,;\, \tau_1}{\Gamma \vdash (\lambda x.e) : \tau \to \tau'/\tau_1 \,;\, \tau_0} \qquad \frac{\Gamma, \langle x:\upsilon \rangle \vdash e : \upsilon' \,;\, \langle \upsilon'/\upsilon_1 \rangle \,;\, \upsilon_1}{\Gamma \vdash (\lambda x.e) : \upsilon \to \upsilon'/\upsilon_1 \,;\, \tau_0 \,;\, \upsilon_0}$$

$$\frac{T = (\tau \to \tau'/\tau_2)}{\Gamma \vdash \mathsf{fix} : (T \to T/\tau_2) \to T/\tau_1 \,;\, \tau_0 \;[;\, \upsilon_0]} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau'/\tau_0 \,;\, \tau_0 \quad \Gamma \vdash e_2 : \tau \,;\, \tau_0}{\Gamma \vdash e_1 e_2 : \tau' \,;\, \tau_0} \qquad \frac{\Gamma \vdash e_1 : \upsilon \to \upsilon'/\upsilon_0 \,;\, \tau_0 \,;\, \upsilon_0 \quad \Gamma \vdash e_2 : \upsilon \,;\, \tau_0 \,;\, \upsilon_0}{\Gamma \vdash e_1 e_2 : \upsilon' \,;\, \tau_0 \,;\, \upsilon_0}$$

$$\frac{\Gamma \vdash e : \tau \,;\, \tau_0 \;[;\, \upsilon_0] \quad \Gamma \vdash e' : \tau' \,;\, \tau_0 \;[;\, \upsilon_0]}{\Gamma \vdash (e, e') : (\tau, \tau') \,;\, \tau_0 \;[;\, \upsilon_0]} \qquad \frac{}{\Gamma \vdash \mathsf{fst} : (\tau, \tau') \to \tau/\tau_1 \,;\, \tau_0 \;[;\, \upsilon_0]} \qquad \frac{}{\Gamma \vdash \mathsf{snd} : (\tau, \tau') \to \tau'/\tau_1 \,;\, \tau_0 \;[;\, \upsilon_0]}$$

$$\frac{\Gamma \vdash e : \mathsf{int} \,;\, \tau_0 \;[;\, \upsilon_0] \quad \Gamma \vdash e_1 : \tau \,;\, \tau_0 \;[;\, \upsilon_0] \quad \Gamma \vdash e_2 : \tau \,;\, \tau_0 \;[;\, \upsilon_0]}{\Gamma \vdash \mathsf{ifz}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau \,;\, \tau_0 \;[;\, \upsilon_0]} \qquad \frac{}{\Gamma \vdash \text{⊞}_1 : ((\tau \to \tau'/\tau_1) \to \tau'/\tau') \to \tau/\tau' \,;\, \tau_0 \;[;\, \upsilon_0]}$$

$$\frac{\Gamma \vdash e : \tau \,;\, \tau}{\Gamma \vdash \{e\} : \tau \,;\, \tau_0} \qquad \frac{\Gamma \vdash e : \upsilon \,;\, \tau_0 \,;\, \upsilon}{\Gamma \vdash \{e\} : \upsilon \,;\, \tau_0 \,;\, \upsilon_0} \qquad \frac{\Gamma \vdash e : \upsilon \,;\, \tau_0 \,;\, \upsilon_0}{\Gamma \vdash \langle e \rangle : \langle \upsilon/\upsilon_0 \rangle \,;\, \tau_0} \qquad \frac{\Gamma \vdash e : \langle \upsilon/\upsilon_0 \rangle \,;\, \tau_0}{\Gamma \vdash {\sim}e : \upsilon \,;\, \tau_0 \,;\, \upsilon_0} \qquad \frac{(x:\tau) \in \Gamma}{\Gamma \vdash x : \tau \,;\, \tau_0} \qquad \frac{(\langle x:\upsilon \rangle) \in \Gamma}{\Gamma \vdash x : \upsilon \,;\, \tau_0 \,;\, \upsilon_0}$$

Figure 5. The type system of $\lambda_1^{\oslash}$

the answer type is τ_0 at the present stage; in a future judgment $\Gamma \vdash e : \upsilon \,;\, \tau_0 \,;\, \upsilon_0$, the answer types are τ_0 at the present stage and υ_0 at the future stage.[6] The future answer type υ_0 is needed to ensure that the generated code, which may incur control effects in the future stage, never goes wrong. Those type metavariables in Figure 5 with numeric subscripts (such as τ_0) are answer types that can be ignored on the first reading. Because constructs such as addition that have nothing to do with staging or delimited control are type-checked in the same way at both stages, we write $\Gamma \vdash e : \tau \,;\, \tau_0 \;[;\, \upsilon_0]$ to mean either a present judgment (without '; υ_0') or a future judgment (with '; υ_0' and requiring τ to be flat).

An answer type is the type of the result of plugging an expression into a delimited context. In other words, an answer type is the type of an expression surrounded by a control delimiter. To take an example from §3.2, in the program $\{\mathsf{const}\ (\mathsf{get} + 40)\}\ 2$, the answer type of the expression get plugged into the delimited context $\mathsf{const}(\square + 40)$ is the type of a function from int to int, even though the whole program has the type int instead. In terms of CPS, an answer type is just the codomain type of a continuation or computation. In fact, our type system is just a 'pullback' of the staged type system of our CPS target language in §5.

Because answer types are effect annotations, they appear not just in judgments but also in function types and code types ('$/\tau_0$' and '$/\upsilon_0$'), where effects are delayed. The typing rules for $\lambda x.e$ show that the effect of e (represented by the answer types τ_1 and υ_1) is incurred only when the function is invoked. The typing rules for $\langle e \rangle$ and ${\sim}e$ show that the future effect of a code value (represented by the answer type υ_0) will be incurred only where the code value is spliced in (and never in the present stage).

An expression is *pure* if it incurs no observable control effect; a pure expression is polymorphic in its answer type(s) (Thielecke 2003). For example, it is easy to derive the judgment $\Gamma \vdash (2+40) : \mathsf{int} \,;\, \tau_0$ for an arbitrary answer type τ_0. In words, it is safe to plug the expression $2+40$ into any delimited context that expects an int, no matter what type τ_0 results from the plugging. In contrast, the expression $\mathsf{get} + 40$ incurs a control effect, as observed in §3.2. Our type system detects this effect: it derives the judgment $\Gamma \vdash (\mathsf{get} + 40) : \mathsf{int} \,;\, \tau_0$ if and only if the answer type τ_0 has the form $\mathsf{int} \to \tau'/\tau_1$. In words, it is safe to plug the expression $\mathsf{get} + 40$ into a delimited context that expects an int if and only if a function from int results from the plugging.

[6] For simplicity, we equate the two answer types distinguished by Danvy and Filinski (1989). The distinction is not hard to add, but not needed here.

All values are pure. Control delimiters also make an expression pure, by masking its effect: in the typing rules for present-stage $\{e\}$ and future-stage $\lambda x.e$, the answer type τ_0 is arbitrary. The latter rule crucially forces future-stage binders to delimit present-stage control: the present-stage answer type is the code type $\langle \upsilon'/\upsilon_1 \rangle$ in the premise but arbitrary in the conclusion.

As an example, the following derivation shows that the program $\{\mathsf{const}\ (\mathsf{get} + 40)\}\ 2$ in §3.2 is well-typed. (Let $T = \mathsf{int} \to \mathsf{int}/\tau_0$ and $S = ((\mathsf{int} \to T/\tau_0) \to T/T) \to \mathsf{int}/T$.)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[] \vdash \text{⊞}_1 : S \,;\, T} \quad \dfrac{\vdots}{k : (\mathsf{int} \to T/\tau_0) \vdash \lambda z.kzz : T \,;\, T}}{[] \vdash \lambda k.\lambda z.kzz : (\mathsf{int} \to T/\tau_0) \to T/T \,;\, T}}{[] \vdash \mathsf{get} : \mathsf{int} \,;\, T}}{[] \vdash \mathsf{get} + 40 : \mathsf{int} \,;\, T}}{[] \vdash \mathsf{const}\ (\mathsf{get} + 40) : T \,;\, T}}{[] \vdash \{\mathsf{const}\ (\mathsf{get} + 40)\} : T \,;\, \tau_0}}{[] \vdash \{\mathsf{const}\ (\mathsf{get} + 40)\}\ 2 : \mathsf{int} \,;\, \tau_0}$$

The accompanying file `circle-shift.elf` type-checks many tests in Twelf. For example, the fixpoint combinator y_ms in §3.3 has the type $((\mathsf{int} \to \langle \upsilon/\upsilon_1 \rangle) \to \mathsf{int} \to \langle \upsilon/\upsilon_0 \rangle) \to \mathsf{int} \to \langle \upsilon/\upsilon_0 \rangle$, in which the present-stage answer types are all $(\mathsf{int} \to T/T) \to \langle \upsilon'/\upsilon_0 \rangle/\langle \upsilon'/\upsilon_0 \rangle$, in which $\mathsf{int} \to T/T$ is the type of the memoization table, and the type $T = (\mathsf{int}, \mathsf{int} \to \langle \upsilon/\upsilon_0 \rangle/\langle \upsilon/\upsilon_0 \rangle)$ encodes the lookup result type $\langle \upsilon/\upsilon_0 \rangle\,\mathsf{option}$ as explained in §3.4.

4. Formal properties

Proposition 4.1 (principal typing) If e is a term such that some judgment $\Gamma \vdash e : \tau \,;\, \tau_0 \;[;\, \upsilon_0]$ is derivable, then some judgment $\Gamma \vdash e : \tau \,;\, \tau_0 \;[;\, \upsilon_0]$ is derivable such that, for any derivable judgment $\Gamma' \vdash e : \tau' \,;\, \tau_0' \;[;\, \upsilon_0']$, there exists a substitution θ for type variables and flat type variables so that $\Gamma' \supseteq \Gamma\theta$, $\tau' = \tau\theta$, and $\tau_0' = \tau_0\theta$ [and $\upsilon_0' = \upsilon_0$].

Proof Our syntax-directed type system constitutes an algorithm to infer Γ, τ, and τ_0 [and υ_0] from e using unification. □

Proposition 4.2 (subject reduction) If $\Gamma \vdash e : \tau \,;\, \tau_0$ is derivable and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : \tau \,;\, \tau_0$ is derivable.

Proof We first note that, if $\Gamma \vdash v^0 : \tau \,;\, \tau_0$ is derivable, then $\Gamma \vdash v^0 : \tau \,;\, \tau_1$ is derivable for any type τ_1. We need a substitution

lemma: if $\Gamma, x:\tau \vdash e:\tau' \,;\, \tau_0$ [; υ_0] and $\Gamma \vdash v^0 : \tau \,;\, \tau_1$ [; υ_1], then $\Gamma \vdash e[x := v^0] : \tau' \,;\, \tau_0$ [; υ_0]. We also need a weakening lemma: if $\Gamma \vdash e : \tau \,;\, \tau_0$ [; υ_0], then $\Gamma, \Delta \vdash e : \tau \,;\, \tau_0$ [; υ_0]. The proofs are routine.

Because our typing rules are all compositional, we can assume without loss of generality that C^0 and C^1 in Figure 3 are just $\Box$. We prove the proposition by case analysis on the reduction. We show a few interesting cases.

Case β_v: Suppose $\Gamma \vdash (\lambda x.e)v^0 : \tau' \,;\, \tau_0$. By inversion, we have $\Gamma, x:\tau \vdash e:\tau' \,;\, \tau_0$ and $\Gamma, v^0 \vdash \tau \,;\, \tau_0$ for some τ. Then the substitution lemma gives $\Gamma \vdash e[x := v^0] : \tau' \,;\, \tau_0$.

Case $出^0$: We have the derivation

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash 出 : T \to \tau/\tau' \,;\, \tau'} \quad \Gamma \vdash v^0 : T \,;\, \tau'}{\Gamma \vdash 出 v^0 : \tau \,;\, \tau'} \\ \vdots \\ \dfrac{\Gamma \vdash D^{00}[出 v^0] : \tau' \,;\, \tau'}{\Gamma \vdash \{D^{00}[出 v^0]\} : \tau' \,;\, \tau_0}}$$

where $T = (\tau \to \tau'/\tau_1) \to \tau'/\tau'$ and the derivation from $出 v^0$ to $D^{00}[出 v^0]$ does not change the answer type τ'. Then we can use the weakening lemma to derive:

$$\dfrac{\dfrac{}{\Gamma \vdash v^0 : T \,;\, \tau'} \quad \dfrac{\dfrac{\dfrac{}{\Gamma, x:\tau \vdash x:\tau \,;\, \tau'} \\ \vdots \\ \dfrac{\Gamma, x:\tau \vdash D^{00}[x] : \tau' \,;\, \tau'}{\Gamma, x:\tau \vdash \{D^{00}[x]\} : \tau' \,;\, \tau_1}}{\Gamma \vdash \lambda x.\{D^{00}[x]\} : \tau \to \tau'/\tau_1 \,;\, \tau'}}{\Gamma \vdash v^0(\lambda x.\{D^{00}[x]\}) : \tau' \,;\, \tau'} \\ \dfrac{}{\Gamma \vdash \{v^0(\lambda x.\{D^{00}[x]\})\} : \tau' \,;\, \tau_0}$$

Case $出^1$: Suppose $\Gamma \vdash \lambda x.D^{10}[出 v^0] : \upsilon \to \upsilon'/\upsilon_1 \,;\, \tau_0 \,;\, \upsilon_0$. By inversion, $\Gamma, \langle x:\upsilon\rangle \vdash D^{10}[出 v^0] : \upsilon' \,;\, \langle \upsilon'/\upsilon_1\rangle \,;\, \upsilon_1$. Then it is easy to derive $\Gamma \vdash \lambda x. {\sim}\{\langle D^{10}[出 v^0]\rangle\} : \upsilon \to \upsilon'/\upsilon_1 \,;\, \tau_0 \,;\, \upsilon_0$. $\Box$

Corollary 4.3 (absence of scope extrusion) If $[] \vdash e : \tau \,;\, \tau_0$ is derivable and $e \rightsquigarrow^* e'$, then e' does not contain free variables.

Proof By induction on the number of steps from e to e'. $\Box$

To state the next two propositions, we define the type environment $\langle\Gamma\rangle = \langle x_1 : \upsilon_1\rangle, \ldots, \langle x_n : \upsilon_n\rangle$ whenever $\Gamma = x_1 : \upsilon_1, \ldots, x_n : \upsilon_n$.

Proposition 4.4 (progress) If $\langle\Gamma\rangle \vdash \{e\} : \tau \,;\, \tau_0$ is derivable then there exists a term e' such that $\{e\} \rightsquigarrow e'$.

Proof By induction on the derivation of $\langle\Gamma\rangle \vdash \{e\} : \tau \,;\, \tau_0$. $\Box$

The statement of this proposition differs from the ordinary form in two respects. First, we consider terms in the form $\{e\}$ only, since a term like $出 v^0$ by itself, without an enclosing control delimiter, cannot be reduced. Second, we allow a potentially non-empty environment $\langle\Gamma\rangle$, in order for the inductive proof to go through.

Propositions 4.2 and 4.4 together show that well-typed programs never go wrong in $\lambda_1^{\oslash}$. In particular, well-typed code generators never go wrong. Moreover, any code they generate never goes wrong either, by the following argument. If $[] \vdash \{e\} : \langle \upsilon/\upsilon_0\rangle \,;\, \tau_0$ is derivable, then the program $\{e\}$ either fails to terminate or evaluates to a code value $\langle v^1\rangle$ such that $[] \vdash v^1 : \upsilon \,;\, \tau_0 \,;\, \upsilon_0$. The next proposition then assures us that v^1 is well-typed at the *present* stage.

Proposition 4.5 If $\langle\Gamma\rangle \vdash v^1 : \upsilon \,;\, \tau_0 \,;\, \upsilon_0$, then $\Gamma \vdash v^1 : \upsilon \,;\, \upsilon_0$.

Proof By induction on the derivation of $\langle\Gamma\rangle \vdash v^1 : \upsilon \,;\, \tau_0 \,;\, \upsilon_0$. $\Box$

5. CPS translation

We define a CPS translation for $\lambda_1^{\oslash}$. Its target is $\lambda_1^{\oslash}$ without present-stage control effects. In other words, the terms of the target are as in $\lambda_1^{\oslash}$ but without the control operators $出$ and $\{\ \}$ at the present stage. The type system of the target is as in $\lambda_1^{\oslash}$ but with all present-stage answer types removed from types and judgments. We define term equality in the target language as the least congruence containing all $\lambda_1^{\oslash}$ reductions (except $\{\}$, $出^0$, $出^1$) and the call-by-value η-reduction $\lambda x.ex = e$, even under present-stage λ.

First, we CPS-translate present-stage types τ to τ^*.

$$\mathsf{int}^* = \mathsf{int} \quad \langle \upsilon_1/\upsilon_2\rangle^* = \langle \upsilon_1/\upsilon_2\rangle \quad (\tau_1, \tau_2)^* = (\tau_1^*, \tau_2^*)$$
$$(\tau_1 \to \tau_2/\tau_3)^* = \tau_1^* \to (\tau_2^* \to \tau_3^*) \to \tau_3^* \qquad \alpha^* = \alpha$$

(Future-stage types do not change.) We also translate environments.

$$[]^* = [] \qquad (\Gamma, x:\tau)^* = \Gamma^*, x:\tau^* \qquad (\Gamma, \langle x:\upsilon\rangle)^* = \Gamma^*, \langle x:\upsilon\rangle$$

The translation of terms depends on the level: a level-i term e translates to $[\![e]\!]_i$ for $i = 0, 1$; a level-0 value v^0 translates to $(\!|v^0|\!)$.

$$\begin{aligned}
[\![v^0]\!]_0 &= \lambda k.k(\!|v^0|\!) \\
[\![e_1 + e_2]\!]_0 &= \lambda k.[\![e_1]\!]_0(\lambda x.[\![e_2]\!]_0(\lambda y.k(x+y))) \\
[\![e_1 e_2]\!]_0 &= \lambda k.[\![e_1]\!]_0(\lambda f.[\![e_2]\!]_0(\lambda x.fxk)) \\
[\![(e_1, e_2)]\!]_0 &= \lambda k.[\![e_1]\!]_0(\lambda x.[\![e_2]\!]_0(\lambda y.k(x,y))) \\
[\![\mathsf{ifz}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]_0 &= \lambda k.[\![e]\!]_0(\lambda x. \\
&\qquad \mathsf{ifz}\ x\ \mathsf{then}\ [\![e_1]\!]_0 k\ \mathsf{else}\ [\![e_2]\!]_0 k) \\
[\![\{e\}]\!]_0 &= \lambda k.k([\![e]\!]_0(\lambda z.z)) \\
[\![\langle e\rangle]\!]_0 &= [\![e]\!]_1 \\
(\!|\lambda x.e|\!) &= \lambda x.[\![e]\!]_0 \\
(\!|\mathsf{fix}|\!) &= \lambda y.\lambda k.k(\mathsf{fix}\,(\lambda f.\lambda x.\lambda k'.yf(\lambda z.zxk'))) \\
(\!|(v_1^0, v_2^0)|\!) &= ((\!|v_1^0|\!), (\!|v_2^0|\!)) \\
(\!|p|\!) &= \lambda x.\lambda k.k(px) \text{ where } p = \mathsf{fst}, \mathsf{snd} \\
(\!|出|\!) &= \lambda f.\lambda k.f(\lambda x.\lambda k'.k'(kx))(\lambda z.z) \\
(\!|v^0|\!) &= v^0 \text{ for all other values } v^0 \\
[\![v^1]\!]_1 &= \lambda k.k\langle v^1\rangle \\
[\![e \oplus e']\!]_1 &= \lambda k.[\![e]\!]_1(\lambda x.[\![e']\!]_1(\lambda y.k\langle {\sim}x \oplus {\sim}y\rangle)) \\
&\qquad \text{where } e \oplus e' \text{ is } e+e',\ ee',\ \text{or } (e,e') \\
[\![\lambda x.e]\!]_1 &= \lambda k.k\langle \lambda x.{\sim}([\![e]\!]_1(\lambda z.z))\rangle \\
[\![\mathsf{ifz}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]_1 &= \lambda k.[\![e]\!]_1(\lambda x.[\![e_1]\!]_1(\lambda y.[\![e_2]\!]_1(\lambda z. \\
&\qquad k\langle \mathsf{ifz}\ {\sim}x\ \mathsf{then}\ {\sim}y\ \mathsf{else}\ {\sim}z\rangle))) \\
[\![\{e\}]\!]_1 &= \lambda k.[\![e]\!]_1(\lambda x.k\langle\{{\sim}x\}\rangle) \\
[\![{\sim}e]\!]_1 &= [\![e]\!]_0
\end{aligned}$$

The output of the translation is in tail form except in $[\![\{e\}]\!]_0$, $(\!|出|\!)$, and $[\![\lambda x.e]\!]_1$. If we define $[\![\lambda x.e]\!]_1 = \lambda k.[\![e]\!]_1(\lambda z.k\langle\lambda x.{\sim}z\rangle)$ naively in tail form, then occurrences of x in e would translate to unbound future-stage variables. The CPS translation thus relies on our treating future-stage binders as present-stage control delimiters.

Proposition 5.1 (type preservation) If the present judgment $\Gamma \vdash e : \tau \,;\, \tau_0$ is derivable in $\lambda_1^{\oslash}$, then the present judgment $\Gamma^* \vdash [\![e]\!]_0 : (\tau^* \to \tau_0^*) \to \tau_0^*$ is derivable. If the future judgment $\Gamma \vdash e : \upsilon \,;\, \tau_0 \,;\, \upsilon_0$ is derivable in $\lambda_1^{\oslash}$, then the *present* judgment $\Gamma^* \vdash [\![e]\!]_1 : (\langle \upsilon/\upsilon_0\rangle \to \tau_0^*) \to \tau_0^*$ is derivable.

Proof By mutual induction on typing derivations.

For the level-0 translation, the only interesting case is when $e = \langle e'\rangle$. Suppose $\Gamma \vdash e' : \upsilon \,;\, \tau_0 \,;\, \upsilon_0$. By the induction hypothesis, we have $\Gamma^* \vdash [\![\langle e'\rangle]\!]_0 = [\![e']\!]_1 : (\langle \upsilon/\upsilon_0\rangle \to \tau_0^*) \to \tau_0^*$.

For the level-1 translation, the interesting cases are as follows.

(Case $e = x$) Suppose $\Gamma \vdash x : \upsilon \; ; \; \tau_0 \; ; \; \upsilon_0$. Then $(\langle x : \upsilon \rangle) \in \Gamma$, so $(\langle x : \upsilon \rangle) \in \Gamma^*$. Thus, $\Gamma^* \vdash [\![x]\!]_1 = \lambda k.k\langle x \rangle : (\langle \upsilon / \upsilon_0 \rangle \to \tau_0^*) \to \tau_0^*$.

(Case $e = \lambda x.e'$) Suppose $\Gamma, \langle x : \upsilon'' \rangle \vdash e' : \upsilon' \; ; \; \langle \upsilon' / \upsilon_1 \rangle \; ; \; \upsilon_1$. The induction hypothesis gives $\Gamma^*, \langle x : \upsilon'' \rangle \vdash [\![e']\!]_1 : (\langle \upsilon' / \upsilon_1 \rangle \to \langle \upsilon' / \upsilon_1 \rangle) \to \langle \upsilon' / \upsilon_1 \rangle$. Then $\Gamma \vdash [\![e]\!]_1 = \lambda k.k\langle \lambda x. \sim([\![e']\!]_1(\lambda z.z)) \rangle : (\langle (\upsilon'' \to \upsilon' / \upsilon_1) / \upsilon_0 \rangle \to \tau_0^*) \to \tau_0^*$.

(Case $e = \{e'\}$) Suppose $\Gamma \vdash e' : \upsilon \; ; \; \tau_0 \; ; \; \upsilon$. By the induction hypothesis, we have $\Gamma^* \vdash [\![e']\!]_1 : (\langle \upsilon / \upsilon \rangle \to \tau_0^*) \to \tau_0^*$. Moreover, we have $\Gamma^*, x : \langle \upsilon / \upsilon \rangle \vdash \langle \{\sim x\} \rangle : \langle \upsilon / \upsilon_0 \rangle$. Hence, we can derive $\Gamma^* \vdash [\![\{e'\}]\!]_1 = \lambda k.[\![e']\!]_1(\lambda x.k\langle \{\sim x\} \rangle) : (\langle \upsilon / \upsilon_0 \rangle \to \tau_0^*) \to \tau_0^*$. □

Proposition 5.2 (equality preservation) If $\Gamma \vdash e : \tau \; ; \; \tau_0$ is derivable and $e \rightsquigarrow e'$, then $[\![e]\!]_0 = [\![e']\!]_0$.

The proof of this proposition can be found in Appendix B.

We leave to future work to prove *simulation*: if $\Gamma \vdash e : \tau \; ; \; \tau$ is derivable and $\tau^* = \tau$, then the $\lambda_1^{\oslash}$ programs $\{e\}$ and $[\![e]\!]_0(\lambda z.z)$ either both evaluate to the same value or both fail to terminate.

6. Related work

Our work draws from two strands of research on partial evaluation and code generation, namely side effects and custom generators.

There is a long tradition of using CPS to write program generators such as pattern-match compilers. Danvy and Filinski (1990, 1992) first applied delimited control to program generation: they showed how to fuse a CPS translation and administrative reductions into one pass by writing the translation either in CPS or using `shift` and `reset`. Similarly in partial-evaluation research, Bondorf (1992) showed how to improve binding times by writing the specializer rather than source programs in CPS. This move helps because the specializer is a fixed program that a programming-language expert can write and prove correct once and for all, whereas many source programs are written and fed to the specializer over time, by domain experts who may be unfamiliar with CPS.

Danvy and Filinski's CPS translations and Bondorf's specializer are sound, in the sense that their continuations are *well-behaved* and do not lead to scope extrusion. Given that these code generators were fixed, it was sensible for their authors to prove their soundness as part of specific proofs of their correctness, rather than as a corollary of some type system that assures that every well-typed generator is sound. Lawall and Danvy (1994) did not rely on such a type system either when they used `shift` and `reset` to reduce Bondorf's specializer to direct style. Our type system accepts these generators not as is but reformulated as combinators (Thiemann 1999). It thus assures them sound; in particular, it is the first to accept Danvy and Filinski's and Lawall and Danvy's uses of delimited control. In contrast, we are not aware of any sound type system for code generators that accepts Sumii and Kobayashi's specializer (2001), which performs let-insertion using mutable state rather than delimited control to speed up specialization.

Programs in particular domains often need to be optimized or specialized using specific techniques that experts of the domains can implement more readily than compiler or specializer writers. Examples include memoization for Gibonacci and dynamic programming (Swadi et al. 2006), pivoting for Gaussian elimination (Carette and Kiselyov 2008), and simplifying complex arithmetic for Fast Fourier Transform (Frigo and Johnson 2005; Kiselyov and Taha 2005). To better support these domain-specific optimizations, two approaches have been developed in the literature.

The first approach is to specialize a source language that has features (such as state) that help express custom optimizations. Along this line, Thiemann and Dussart (1999) built an offline specializer for a higher-order language with mutable references. The example source programs in their paper show how application programmers can persuade their specializer to produce efficient code: by expressing unspecialized optimization techniques (such as memoization) and by improving binding times manually (for instance, writing a recursive coercion to transform static data into dynamic data).

As usual, Thiemann and Dussart's specializer uses continuations to perform let-insertion. What is less usual is that it is written in a store-passing style, so as to manage mutable references at specialization time. These static references are organized by a binding-time analysis (BTA) into *regions* (Talpin and Jouvelot 1992), which limit the references' lifetimes much as is rudimentarily achieved by the simulation of state by delimited control in §3. To prevent scope extrusion, the BTA ensures that a static reference is used in the scope of a dynamic binder only if the reference's lifetime is local to the binder. This constraint is analogous to our restriction of static effects (not just mutation) to the scope of dynamic binders. Whereas Thiemann and Dussart's specializer infers binding-time annotations and performs let-insertion automatically and safely, our type system (akin to their constraints on annotations) ensures the safety of code generators written by application programmers.

The last difference brings us to the second way to support domain-specific optimizations: letting domain experts write code generators. This approach has the advantage that the behavior of a code generator on a static input tends to be more predictable than the behavior of a specializer (especially its BTA) on a source program. Swadi et al.'s (2006) and Carette and Kiselyov's (2008) uses of CPS and monadic style in domain-specific code generators raise the need for a multilevel language to provide the convenience of effects without the risk of scope extrusion. Such a language is needed to ease the development and assure the safety of a variety of domain-specific code generators, not just a fixed specializer.

This paper addresses this need, following two previous papers. To prevent scope extrusion in a multilevel language with references, Calcagno et al. (2000) proposed to store only values of *closed types* in mutable cells. This (unimplemented) proposal is too restrictive for our purposes, because we want to store future-stage variables in memoization tables (as in our Gibonacci example).

We previously (Kameyama et al. 2008) introduced a typed two-level language λ_{1v}^{α} and translated it to System F. That translation fails in the presence of effects (due to scope extrusion, manifest as a lack of type coercions), yet it is more complex than our CPS translation in §5 here. So far, then, it seems simpler to combine staging and effects by translating effects rather than staging away.

7. Conclusions

We have presented the first language for writing code generators that provides delimited control operators while assuring statically that all generated code is well-formed. This language thus strikes a balance between clarity and safety that helps application programmers implement domain-specific optimizations in practical, reusable generators of specialized programs. The key idea that enables this balance is to restrict control effects to the scope of generated binders, that is, to treat generated binders as control delimiters.

As the examples illustrate, our language is expressive enough in many practical settings we have encountered. Nevertheless, it would be useful to find a sound way to relax our restriction on control effects, so as to perform let- and if-insertion outside the closest generated binder. We could then express loop-invariant code motion and generate assertions to be checked as early as possible. It might also help us simultaneously access multiple pieces of state at different generated scopes, not just one piece as in §3.2.

Another good way to enrich our language is to add delimited control to a richer language (like Taha and Nielsen's (2003)) with more than two levels (probably not hard), `run`, and CSP. Finally, our language can be made much more comfortable to use by adding polymorphism over effect annotations (Asai and Kameyama 2007).

Acknowledgments

We thank Kenichi Asai, Olivier Danvy, and Atsushi Igarashi for helpful discussions, and the reviewers for their comments.

References

Asai, Kenichi, and Yukiyoshi Kameyama. 2007. Polymorphic delimited continuations. In *APLAS*, 239–254. LNCS 4807.

Bondorf, Anders. 1992. Improving binding times without explicit CPS-conversion. In *Lisp & functional programming*, 1–10.

Bondorf, Anders, and Olivier Danvy. 1991. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming* 16(2):151–195.

Calcagno, Cristiano, Eugenio Moggi, and Walid Taha. 2000. Closed types as a simple approach to safe imperative multi-stage programming. In *ICALP*, 25–36. LNCS 1853.

Carette, Jacques, and Oleg Kiselyov. 2008. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Science of Computer Programming*.

Cohen, Albert, Sébastien Donadio, María Jesús Garzarán, Christoph Armin Herrmann, Oleg Kiselyov, and David A. Padua. 2006. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming* 62(1):25–46.

Czarnecki, Krzysztof, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. 2004. DSL implementation in MetaOCaml, Template Haskell, and C++. In *DSPG 2003*, 51–72. LNCS 3016.

Danvy, Olivier, and Andrzej Filinski. 1989. A functional abstraction of typed contexts. Tech. Rep. 89/12, DIKU, University of Copenhagen, Denmark. `http://www.daimi.au.dk/~danvy/Papers/fatc.ps.gz`.

———. 1990. Abstracting control. In *Lisp & functional programming*, 151–160.

———. 1992. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2(4): 361–391.

Davies, Rowan. 1996. A temporal logic approach to binding-time analysis. In *LICS*, 184–195.

Eckhardt, Jason, Roumen Kaiabachev, Emir Pašalić, Kedar N. Swadi, and Walid Taha. 2005. Implicitly heterogeneous multi-stage programming. In *GPCE*, 275–292. LNCS 3676.

Elliott, Conal. 2004. Programming graphics processors functionally. In *Haskell workshop*, 45–56.

Felleisen, Matthias, and Daniel P. Friedman. 1987. A reduction semantics for imperative higher-order languages. In *PARLE: Parallel architectures and languages Europe. Volume II: Parallel languages*, 206–223. LNCS 259.

Filinski, Andrzej. 1994. Representing monads. In *POPL*, 446–457.

Frigo, Matteo, and Steven G. Johnson. 2005. The design and implementation of FFTW3. *Proceedings of the IEEE* 93(2):216–231.

Gomard, Carsten K., and Neil D. Jones. 1991. A partial evaluator for the untyped lambda calculus. *Journal of Functional Programming* 1(1):21–69.

Hammond, Kevin, and Greg Michaelson. 2003. Hume: A domain-specific language for real-time embedded systems. In *GPCE*, 37–56. LNCS 2830.

Kameyama, Yukiyoshi, Oleg Kiselyov, and Chung-chieh Shan. 2008. Closing the stage: From staged code to typed closures. In *PEPM*, 147–157.

Kiselyov, Oleg. 2006. Native delimited continuations in (byte-code) OCaml. `http://okmij.org/ftp/Computation/Continuations.html#caml-shift`.

Kiselyov, Oleg, Chung-chieh Shan, and Amr Sabry. 2006. Delimited dynamic binding. In *ICFP*, 26–37.

Kiselyov, Oleg, and Walid Taha. 2005. Relating FFTW and split-radix. In *ICESS*, 488–493. LNCS 3605.

Lawall, Julia L., and Olivier Danvy. 1994. Continuation-based partial evaluation. In *Lisp & functional programming*, 227–238.

Lengauer, Christian, and Walid Taha, eds. 2006. *Special issue on the 1st MetaOCaml workshop (2004)*, vol. 62(1) of *Science of Computer Programming*.

Leroy, Xavier, and François Pessaux. 2000. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems* 22(2):340–377.

McAdam, Bruce J. 2001. Y in practical programs. Workshop on fixed points in computer science; `http://www.dsi.uniroma1.it/~labella/absMcAdam.ps`.

MetaOCaml. 2006. MetaOCaml. `http://www.metaocaml.org`.

Michie, Donald. 1968. "Memo" functions and machine learning. *Nature* 218:19–22.

Moreau, Luc. 1998. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation* 11(3):233–279.

Nielson, Flemming, and Hanne Riis Nielson. 1988. Automatic binding time analysis for a typed λ-calculus. In *POPL*, 98–106.

———. 1992. *Two-level functional languages*. Cambridge University Press.

Parigot, Michel. 1992. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR*, 190–201. LNAI 624.

Pašalić, Emir, Walid Taha, and Tim Sheard. 2002. Tagless staged interpreters for typed languages. In *ICFP*, 157–166.

Peyton Jones, Simon L. 2003. The Haskell 98 language and libraries. *Journal of Functional Programming* 13(1):1–255.

Püschel, Markus, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. 2005. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE* 93(2): 232–275.

Sørensen, Morten Heine B., Robert Glück, and Neil D. Jones. 1994. Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In *ESOP*, 485–500. LNCS 788.

Sumii, Eijiro, and Naoki Kobayashi. 2001. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation* 14(2–3):101–142.

Swadi, Kedar, Walid Taha, and Oleg Kiselyov. 2005. Dynamic programming benchmark. `http://www.metaocaml.org/examples/dp/`.

Swadi, Kedar, Walid Taha, Oleg Kiselyov, and Emir Pašalić. 2006. A monadic approach for avoiding code duplication when staging memoized functions. In *PEPM*, 160–169.

Taha, Walid. 2000. A sound reduction semantics for untyped CBN multi-stage computation. In *PEPM*, 34–43.

———. 2005. Resource-aware programming. In *ICESS*, 38–43. LNCS 3605.

Taha, Walid, and Michael Florentin Nielsen. 2003. Environment classifiers. In *POPL*, 26–37.

Talpin, Jean-Pierre, and Pierre Jouvelot. 1992. Polymorphic type, region and effect inference. *Journal of Functional Programming* 2(3):245–271.

Thielecke, Hayo. 2003. From control effects to typed continuation passing. In *POPL*, 139–149.

Thiemann, Peter. 1999. Combinators for program generation. *Journal of Functional Programming* 9(5):483–525.

Thiemann, Peter, and Dirk Dussart. 1999. Partial evaluation for higher-order languages with state. `http://www.informatik.uni-freiburg.de/~thiemann/papers/mlpe.ps.gz`.

Wadler, Philip L. 1992. Comprehending monads. *Mathematical Structures in Computer Science* 2(4):461–493.