JFP **21** (6): 617–662, 2011.
doi:10.1017/S0956796811000256

Shifting the stage

Staging with delimited control

YUKIYOSHI KAMEYAMA
Department of Computer Science, University of Tsukuba
(*e-mail:* kameyama@acm.org)

OLEG KISELYOV
Monterey, CA, USA
(*e-mail:* oleg@okmij.org)

CHUNG-CHIEH SHAN
Cornell University
(*e-mail:* ccshan@post.harvard.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* and early error reporting: a well-typed code generator neither goes wrong nor generates code that goes wrong. Side effects such as state and control ease *correctness* and *expressivity*: 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. Together, types and effects enable structuring code generators as compositions of modules with well-defined interfaces, and hence scaling to large programs. However, blindly adding effects renders multilevel types unsound.

We introduce the first multilevel calculus with control effects and a sound type system. We give small-step operational semantics as well as a one-pass continuation-passing-style 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 continuation-passing or monadic style.

1 Introduction

High-performance computing and high-assurance embedded computing often call for programs that are specialized for particular inputs, usages, or processors. Examples include specializing matrix multiplication to the target computer system (Whaley & Petitet, 2005; Cohen *et al.*, 2006), specializing Fast Fourier Transform to

the length of the transformed sequence (Frigo & Johnson, 2005; Püschel *et al.*, 2005), and specializing signal processing algorithms to the architecture of a particular Field Programmable Gate Array (Püschel *et al.*, 2005). As another example, Carette (2006) found 35 variously specialized implementations of the same Gaussian elimination algorithm in the industrial computer algebra system Maple.

Specializations like these go well beyond inlining and constant propagation, demanding domain-specific knowledge (such as symmetry groups of the roots of unity, in the case of Fast Fourier Transform). A good illustration is the powerful specialization of `0 * x` as `0`, which predicts the result of multiplication even for unknown `x` and leads to further simplifications. For floating-point numbers, this specialization is invalid – unless one knows that in no program run can `x` be `NaN` or `Infinity`. General-purpose compilers cannot be counted on for optimizations so complex and particular; in fact, optimizing compilers are lagging further behind experts in producing the highest performance code (Cohen *et al.*, 2006). That is why writing specialized programs by hand – however labor-intensive and error-prone – is commonplace: All the versions of Gaussian eliminations in Maple were written by hand.

The leading approach to automating these specializations is code generation, also called generative programming (Cohen *et al.*, 2006). In this approach, domain experts versed in the desired optimizations express their knowledge by building a custom code generator or specializer. The generated or specialized code is then compiled by a general-purpose compiler. This division of labor avoids the dilemma that either a compiler writer has to acquire application-specific knowledge or a domain expert has to learn the intimate internals of a compiler (such as closure conversion, frame structure, instruction scheduling, and register allocation). The division of labor may go further: Ideally, one expert on computational geometry, another on linear algebra, and a third on loop transformations can work on separate modules of the same custom code generator, pooling their expertise to produce high-performance programs with a variety of optimizations.

Custom code generators today are built on top of a variety of substrates, including general-purpose programming languages (Kamin, 1996) as well as preprocessors (such as m4) and macro processors (Lisp macros, camlp4), template systems (C++ templates), partial evaluators and supercompilers, and multilevel languages. To support the division of labor just described and enable modular reasoning about the specialization process, a language for custom code generators needs to be high-level: It ought to let domain experts express algorithms clearly, specify abstract interfaces between modules, and establish basic prerequisites to correctness, such as type safety. Most languages fall short in this regard. For example, the popular strategy of generating code using `printf` is described by one practitioner Whaley (Whaley & Petitet, 2005) as "spending time in hell."[1] After all, domain experts who use the language are not professional metaprogrammers, so it is paramount to detect and report errors early. At the very least, we demand that if the source of a code generator is well-formed and well-typed, so must be its result. On the other hand, although many partial evaluators have been proven correct (e.g., Bondorf,

[1] `http://math-atlas.sourceforge.net/devel/atlas_contrib/`

1992; Dussart & Thiemann, 1996), they make it difficult to express domain-specific optimizations.

This paper presents a language for code generation that is high level in the sense that it features an unprecedented combination of expressivity, modularity, and type safety. Our starting point is multilevel languages such as MetaOCaml (Nielson and Nielson, 1988; Gomard and Jones, 1991; Calcagno *et al*., 2004; Lengauer and Taha, 2006; MetaOCaml, 2006), which extend a high-level language with code-generation constructs. These languages already offer attractive support for modular, type-safe programming. To start with, they treat generated code fragments – and functions that transform them – as first-class values that can be built and composed. Moreover, they guarantee that the generated code is syntactically well-formed, even well-typed and well-scoped, so that it shall always compile. Multilevel languages are thus popular in applications of code generation such as partial evaluation (Gomard & Jones, 1991), continuation-passing style (CPS) translation (Danvy & Filinski, 1992), embedding domain-specific languages (Pašalić *et al*., 2002; Czarnecki *et al*., 2004), and controlling special processors (Elliott, 2004; Taha, 2005).

Unfortunately, the type safety that current multilevel languages offer comes at a stiff price of inexpressivity: many optimizations cannot be implemented in a module that encapsulates domain-specific knowledge reusably, or even states them clearly. For example, many textbook numerical algorithms are typically expressed as computations over mutable arrays. When specializing such algorithms, we may be able to perform some of the computations on arrays at specialization time and so produce faster code (see Section 5.3 for an example; Sumii & Kobayashi (2001) extensively discuss mutation in specialization). Alas, current multilevel languages either prohibit mutations when it comes to code fragments or rescind their guarantees of always generating well-formed code (see Section 2.1). Another example is a common optimization called *let-insertion* or *scalar promotion*, which means to bind the result of a complex expression such as array lookup to a temporary variable so as to avoid recomputing it. Expressing this optimization in a clear and modular way requires control effects (Bondorf, 1992; Lawall & Danvy, 1994; Carette & Kiselyov, 2011); again, current multilevel languages either prohibit control effects when it comes to code fragments or rescind their safety guarantees (see Section 2.4).

This tension between expressivity and safety is the main challenge that we address in this paper. We draw our notion of expressivity from Felleisen (1991): Informally, it is the ability to define an operation without requiring a global, unmodular transformation of the code that uses the operation. Just as it is possible to implement mutable state in a call-by-value lambda-calculus by a state-passing transformation, it is possible to implement both mutable array references and let-insertion in current multilevel languages without voiding their safety guarantees. However, these features are not *expressible* in current multilevel languages: We would have to program in CPS (Danvy & Filinski, 1990; Bondorf, 1992; Danvy & Filinski, 1992) or, equivalently, monadic style (Swadi *et al.* 2006; Carette and Kiselyov, 2011). Hence, although let-insertion is implementable, it is not implementable as a *library function.* An expert on let-insertion cannot offer it as a function that other experts can use whenever they need it. To use let-insertion, the experts must write all of their code in CPS or monadic style. The pervasive use of CPS or monadic style is prohibitively

unpalatable, especially by domain experts who are not programming-language researchers and especially in languages where first-class functions are awkward to express. CPS and monadic style also distort the expression of the algorithm beyond its textbook-familiar style, as evident from the (moderately large) body of code developed by Carette & Kiselyov (2011). In theoretical terms, the inexpressivity of let-insertion in existing type-safe multilevel languages is troubling.

Contributions. This paper[2] introduces $\lambda^{\oslash}$, the first multilevel language that allows expressing a form of let-insertion – in general, a form of effectful operations on potentially open code values – while ensuring in its type system that all generated code is well-typed and well-scoped. The language $\lambda^{\oslash}$ is the first type-sound calculus that combines code generation and delimited control. (Delimited control in turn expresses any representable computational effect (Filinski, 1994).) 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. On one hand, the language $\lambda^{\oslash}$ extends the multilevel calculus $\lambda^{\bigcirc}$ (Davies, 1996) with delimited control operators. On the other hand, the language simplifies a variant of λ^{α}_{1v} (Kameyama *et al.*, 2008), and the restriction on effects is also simple.

We have embedded the language in MetaOCaml, where the restriction has to be checked manually, and implemented it fully in Twelf. The latter implementation also mechanizes the proofs of type soundness. Another way to implement our language is to translate it into any existing multilevel language using the one-pass CPS translation we present in Section 6.

Our restriction means that let-bindings and if- and other statements are inserted under the closest generated binder. Likewise, mutable cells and arrays created under a generated binder are not accessible beyond the scope of the binder or within nested generated binders. In other words, no control, mutation, or other effect can cross any generated binder. *Exactly* the same restriction holds if we implement let-insertion or mutable cells in existing multilevel languages by means of CPS or monadic style. Thus, any code that cannot be written in our $\lambda^{\oslash}$ (such as inserting let-bindings across future-stage binders) cannot be written *at all* in any existing type-safe multilevel language, irrespective of the style.

We demonstrate that our restriction is not severe: It does permit writing code generators and specialization libraries that encapsulate useful high-level abstractions such as dynamic programming. We are also able to write a framework for specialized Gaussian elimination, this time preserving the textbook-familiar style of the algorithm. We can thus use our new language to create frameworks and embedded domain-specific languages for program generation that application programmers and domain experts can use.

Our implementations of $\lambda^{\oslash}$ and direct-style code generators, including the examples of the dynamic-programming specialization benchmark (Swadi *et al.*, 2005), are

[2] The present paper is the expanded version of our PEPM 2009 paper (Kameyama *et al.*, 2009). The major changes are as follows: We expanded Section 5 to discuss three larger examples of our system in use. We revised Section 6 to present a one-pass CPS translation and prove that it preserves reductions. We added Section 7 to extend our language to an arbitrary number of levels.

all available as supplementary material online at http://dx.doi.org/10.1017/S0956796811000256 and also at http://okmij.org/ftp/Computation/staging/README.dr

Organization. Section 2 illustrates the challenges of specialization on a simple example of staging the memoizing fixpoint combinator in MetaOCaml. For clarity, the bulk of the paper deals with a two-level version $\lambda_1^{\oslash}$ of our language. Section 3 introduces our new language and shows how its combination of delimited control and staging meets the challenges. Section 4 explains the type system of our language and proves that it is sound and delivers principal types. Section 5 gives larger programming examples, found in the MetaOCaml literature. Section 6 presents a CPS translation for the language. Section 7 generalizes $\lambda_1^{\oslash}$ to the full language $\lambda^{\oslash}$ with an arbitrary number of levels. Section 8 discusses related work, and Section 9 concludes.

2 Running example

Our running example is to generate specialized code using a memoizing fixpoint combinator. The combinator underpins a simple library for dynamic programming that lets domain experts program their tasks without worrying about avoiding repeated computations. Our memoizing fixpoint combinator for code generation lets these domain experts generate specialized versions of their programs. The running example illustrates both how to use the combinator to generate specialized code and how to build the combinator to provide a useful abstraction.

We borrow the running example, quirks and all, from the dynamic-programming specialization benchmark (Swadi *et al.*, 2005). The benchmark had to use monadic style to implement specialization. We use the language enriched with delimited continuations to demonstrate the benefits of expressivity, the difference between just implementing let-insertion and expressing it. The example also illustrates the dangers of the unrestricted use of control effects, motivating our restriction. We focus on *adjusting* an existing library for dynamic programming and its open-recursion coding style to permit generation of specialized programs. Whether open recursion coupled with the memoizing fixpoint combinator is the best way to express dynamic-programming algorithms is beyond the scope of this paper, and so is a general discussion of dynamic programming, its domain-specific languages, and implementation strategies.

As a toy dynamic-programming problem we take computing the Gibonacci function, which 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 (Swadi *et al.*, 2005), such as longest common subsequence, binary knapsack, and optimal matrix-multiplication ordering. We discuss such larger examples in Section 5.

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.[3] 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. 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))>.
```

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 (and the one used by Swadi *et al.* (2006), whom we follow) is to use the abstraction that has been developed for that purpose.

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

The 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 tableref = ref (empty ()) in
  let rec memo n =
    match (lookup n !tableref) with
    | None -> let v = f memo n in (tableref := ext !tableref 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 `table`, `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. We refer the reader to Swadi *et al.* (2006) for further justification and discussion of this memoization method. Once again, our goal is to add staging to an already developed framework rather than to introduce our own.

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. (MetaOCaml implicitly α-converts the names of all future-stage bound-variables into fresh names like `y_1`, `y_2` etc.) This example illustrates that mutation and other effects, 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.

In different domains, the most profitable optimizations 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. Thus, whereas code generation is memoized, the generated code does not memoize (Bondorf & 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 (Danvy & Filinski, 1990; Bondorf, 1992; Danvy & Filinski, 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 (Wadler, 1992; Peyton Jones, 2003; Minsky, 2008; Carette & Kiselyov, 2011) 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 & Kiselyov, 2011) that alienates application programmers and obstructs our quest to help end users specialize their code. Theoretically, the problem is of expressivity (Felleisen, 1991): The direct-style specialized `sgib` differs from the unspecialized code `gib` only in staging annotations for parts of the code; erasing annotations from `sgib` recovers the original `sgib`. The relation of `sgib_c` to `sgib` is quite more involved, including not only the placement of local staging annotations but also the global rewriting of the whole code to the monadic style. Therefore, in order to use the staging memoizing fixpoint combinator of (Swadi *et al.*, 2006), the end user has to first rewrite the code in the monadic style – indicating that the memoizing fixpoint combinator is inexpressive in pure MetaOCaml.

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. Again, the rewriting of the code in CPS or monadic style indicates that if-insertion is not expressible in pure MetaOCaml (without resorting to effects).

2.4 Delimited control and its risk of scope extrusion

Lawall & Danvy (1994) show how to use Danvy and Filinski's *delimited control operators* `shift` and `reset` (Danvy & Filinski, 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, 2010), 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 Section 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 the memoizing staged fixpoint combinator `y_ms` and the let-insertion locus specifier `top_fn` in Section 3.4.) The same delimited control operators let us accomplish if-insertion using the intuitive way to write `gen` in Section 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)>.
```

Variables x, y, z, f, k

Expressions $e ::= n \mid e + e \mid \lambda x.e \mid \mathsf{fix} \mid ee \mid (e,e) \mid \mathsf{fst} \mid \mathsf{snd}$
$\mid \mathsf{ifz}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathrm{出} \mid \{e\} \mid \langle e \rangle \mid {\sim}e \mid x$

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

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 future-stage binder thus incurs no effect *observable* outside the binder's scope. This restriction turns out to permit many practical forms of memoization, let-insertion, and if-insertion – in particular, all of the cases described by Carette and Kiselyov (2011) and Swadi *et al.* (2005, 2006) – so application programmers can now implement such optimizations safely (without risking scope extrusion) and naturally (in direct style). Theoretically, let- and if-insertion become expressible.

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. For clarity, until Section 7 we restrict our attention to a two-level version of the language, called $\lambda_1^{\oslash}$. The language models a subset of MetaOCaml extended with delimited control operators. (Delimited control operators may appear in both stages. Just as we defer dealing with staging forms in the future-stage code until Section 7, we could have likewise deferred future-stage delimited control. Doing so however makes the type system, Section 4, less uniform and harder to understand.) Figure 1 shows the syntax: It features integer literals n and their arithmetic $+$, λ-abstractions and their applications, and pairs (e_1, e_2) and their projections fst and snd. We write $\mathsf{let}\ x_1 = e_1\ \mathsf{and} \ldots \mathsf{and}\ x_n = e_n\ \mathsf{in}\ e$ as shorthand for $(\lambda x_1 \ldots \lambda x_n.\, e) e_1 \ldots e_n$. The conditional $\mathsf{ifz}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ reduces to e_1 if e is zero, and to e_2 if e is a nonzero integer literal. The constant fix is the applicative fixpoint combinator. In $\lambda_1^{\oslash}$, λ-abstractions are the only binding forms, which simplifies the Twelf implementation and the mechanization of type soundness. (The same motivation explains our implementation choice for pair projections, as higher-order constants rather than syntactic forms.) 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.

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, inserting `reset` otherwise. 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 (Leroy & Pessaux 2000). It may be simpler however to add staging to a

Values
$$v^0 ::= n \mid \lambda x.e \mid \mathsf{fix} \mid (v^0,v^0) \mid \mathsf{fst} \mid \mathsf{snd} \mid \amalg \mid \langle v^1 \rangle \mid x$$
$$v^1 ::= n \mid v^1+v^1 \mid \lambda x.v^1 \mid \mathsf{fix} \mid v^1 v^1 \mid (v^1,v^1) \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{ifz}\, v^1 \,\mathsf{then}\, v^1 \,\mathsf{else}\, v^1 \mid \amalg \mid \{v^1\} \mid x$$

Frames
$$F^0 ::= \Box+e \mid v^0+\Box \mid \Box e \mid v^0\Box \mid (\Box,e) \mid (v^0,\Box) \mid \mathsf{ifz}\,\Box\,\mathsf{then}\,e\,\mathsf{else}\,e$$
$$F^1 ::= \Box+e \mid v^1+\Box \mid \Box e \mid v^1\Box \mid (\Box,e) \mid (v^1,\Box) \mid \mathsf{ifz}\,\Box\,\mathsf{then}\,e\,\mathsf{else}\,e \mid \mathsf{ifz}\,v^1\,\mathsf{then}\,\Box\,\mathsf{else}\,e \mid \mathsf{ifz}\,v^1\,\mathsf{then}\,v^1\,\mathsf{else}\,\Box \mid \{\Box\}$$

Delimited contexts
$$D^{00} ::= \Box \mid D^{00}[F^0] \mid D^{01}[\sim\Box]$$
$$D^{10} ::= D^{10}[F^0] \mid D^{11}[\sim\Box]$$
$$D^{01} ::= D^{01}[F^1] \mid D^{00}[\langle\Box\rangle]$$
$$D^{11} ::= \Box \mid D^{11}[F^1] \mid D^{10}[\langle\Box\rangle]$$

Contexts
$$C^0 ::= D^{00} \mid C^0[\{D^{00}\}] \mid C^1[\lambda x.D^{10}]$$
$$C^1 ::= D^{01} \mid C^0[\{D^{01}\}] \mid C^1[\lambda x.D^{11}]$$

Fig. 2. Values and contexts.

dialect of ML that has delimited control with the (superset of the) type system described in the present paper (Masuko & Asai, 2009).

3.1 Staging

As described in Section 2, our staging facility consists of brackets $\langle e \rangle$ and escapes $\sim e$. These constructs are written `.< >.` and `.~` in actual MetaOCaml code. The staging *level* of an expression affects whether it is a value and how a non-value is decomposed into a context and a redex (Taha, 2000). So far, our calculus has only two levels, present stage and future stage (more levels are introduced in Section 7). They correspond to two evaluation "modes," reduction and code-building (Igarashi & Iwaki, 2007). To notate these levels, we put the superscripts 0 and 1 on metavariables, such as values and contexts in Figure 2.[4] Brackets enclose a future-stage expression to form a present-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 are the smallest, with respect to decomposition, non-empty contexts. A frame F^i can be plugged with a level-i expression to form a slightly larger level-i expression. In a degenerate language

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

$$
\begin{array}{rll}
C^0\,[n_1 + n_2] \rightsquigarrow C^0\,[n] & \text{where } n = n_1 \dot{+} n_2 & (+) \\
C^0\,[(\lambda x.e)\,v^0] \rightsquigarrow C^0\,[e[x := v^0]] & & (\beta_v) \\
C^0\,[\mathsf{fix}\,v^0] \rightsquigarrow C^0\,[\lambda x.v^0(\mathsf{fix}\,v^0)\,x] & \text{where } x \text{ is fresh} & (\mathsf{fix}) \\
C^0\,[\mathsf{fst}\,(v_1^0, v_2^0)] \rightsquigarrow C^0\,[v_1^0] & & \\
C^0\,[\mathsf{snd}\,(v_1^0, v_2^0)] \rightsquigarrow C^0\,[v_2^0] & & \\
C^0\,[\mathsf{ifz}\,0\,\mathsf{then}\,e_1\,\mathsf{else}\,e_2] \rightsquigarrow C^0\,[e_1] & & \\
C^0\,[\mathsf{ifz}\,n\,\mathsf{then}\,e_1\,\mathsf{else}\,e_2] \rightsquigarrow C^0\,[e_2] & \text{if } n \neq 0 & \\
C^0\,[\{v^0\}] \rightsquigarrow C^0\,[v^0] & & (\{\}) \\
C^1\,[{\sim}\langle v^1\rangle] \rightsquigarrow C^1\,[v^1] & & ({\sim}) \\
C^0\,[\{D^{C0}[\text{山}v^0]\}] \rightsquigarrow C^0\,[\{v^0(\lambda y.\{D^{C0}[y]\})\}] & \text{where } y \text{ is fresh} & (\text{山}^0) \\
C^1\,[\lambda x.D^{10}[\text{山}v^0]] \rightsquigarrow C^1\,[\lambda x.{\sim}(v^0(\lambda y.\{\langle D^{10}[y]\rangle\}))] & \text{where } y \text{ is fresh} & (\text{山}^1)
\end{array}
$$

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

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 Section 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 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) and Gomard and Jones's (1991) two-level λ-calculi (though the latter does not type-check the generated code).

Excluding `run` from our language makes it simpler to implement: otherwise, either the run-time system has to include a compiler and a dynamic linker, or our compiler should be capable of producing target code that produces target code at run-time (see Leone & Lee, 1998 for the example of the latter). Excluding `run` makes the language simpler to prove sound, because the type system need not include environment classifiers (Taha & 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 already supports the intended use for $\lambda_1^{\oslash}$, namely to build domain-specific language "compilers" that generate families of optimized library routines – such as Gaussian elimination (Carette & Kiselyov, 2011), Fast Fourier Transform (Frigo & Johnson, 2005; Kiselyov & Taha, 2005), linear signal processing (Püschel *et al.*, 2005), and embedded code (Hammond & Michaelson, 2003) – to be used in applications other than the generator itself.

The lack of cross-stage persistence in $\lambda_1^{\oslash}$ means that there is no "polymorphic 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

$$C^0\,[F^0[出 v^0]] \rightsquigarrow C^0\,[出(\lambda k.\, v^0(\lambda y.\,\{k(F^0[y])\}))] \qquad \text{where } k \text{ and } y \text{ are fresh}$$
$$C^0\,[\{出 v^0\}] \rightsquigarrow C^0\,[\{v^0(\lambda y.\, y)\}] \qquad \text{where } y \text{ is fresh}$$
$$C^1\,[F^1[\sim(出 v^0)]] \rightsquigarrow C^1\,[\sim(出(\lambda k.\, v^0(\lambda y.\,\{k\langle F^1[\sim y]\rangle\})))] \qquad \text{where } k \text{ and } y \text{ are fresh}$$
$$C^0\,[\langle\sim(出 v^0)\rangle] \rightsquigarrow C^0\,[出 v^0]$$
$$C^1\,[\lambda x.\sim(出 v^0)] \rightsquigarrow C^1\,[\lambda x.\sim(v^0(\lambda y.\, y))] \qquad \text{where } y \text{ is fresh}$$

Fig. 4. Bubble-up reductions for delimited control, replacing rules $出^0$ and $出^1$ in Figure 3.

specific data types – integers, pairs of integers, and so on, as discussed by Davies and Pfenning (2001). Whereas cross-stage persistence is important when using `run` (Taha & 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 being present.

3.2 Delimited control

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

When 出 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 出 is supposed to be applied to a function value, say v^0. When $出 v^0$ 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 $出^0$ rule in Figure 3 shows, the subexpression $D^{00}[出 v^0]$ reduces to the application $v^0(\lambda x.\,\{D^{00}[x]\})$, reifying the captured delimited context D^{00} as the abstraction $\lambda x.\,\{D^{00}[x]\}$.

The single step $出^0$ reduction (and its companion $出^1$, discussed in Section 3.3) can be decomposed into a sequence of finer grain reductions in which the 出-application *bubbles up* and builds up the delimited context by local rewriting (Felleisen *et al.*, 1986; Parigot, 1992). More specifically, we can replace the rules $出^0$ and $出^1$ in Figure 3 by the rules in Figure 4. The bubble-up reductions are truly small-step in that they do not require examining an arbitrarily long part of the context looking for the delimiter (such as reset). The bubble-up reductions are therefore insightful, and easier to mechanize, especially in our case where future-stage binders also act as control delimiters (see Kameyama *et al.*, 2010 for more details on mechanization).

The attraction of delimited control is the ability to express any representable computational effect (Filinski, 1994). We illustrate this ability by using delimited control to simulate mutable state (Filinski, 1994; Kiselyov *et al.*, 2006). We define the terms

$$\mathsf{const} = \lambda y.\, \lambda z.\, y, \qquad \mathsf{get} = 出(\lambda k.\, \lambda z.\, kzz), \qquad \mathsf{put} = \lambda z'.\, 出(\lambda k.\, \lambda z.\, kz'z'). \qquad (1)$$

The reduction sequence below illustrates how const and get work:

$$\begin{aligned}
\{\mathsf{const}\ (\mathsf{get}+40)\}\ 2 &\rightsquigarrow_{\text{出}_0} \{(\lambda k.\,\lambda z.\,kzz)(\lambda x.\,\{\mathsf{const}(x+40)\})\}\ 2 \\
&\rightsquigarrow_{\beta_v} \{\lambda z.\,(\lambda x.\,\{\mathsf{const}(x+40)\})zz\}\ 2 \\
&\rightsquigarrow_{\{\}} (\lambda z.\,(\lambda x.\,\{\mathsf{const}(x+40)\})zz)\ 2 \\
&\rightsquigarrow_{\beta_v} (\lambda x.\,\{\mathsf{const}(x+40)\})\ 2\ 2 \\
&\rightsquigarrow_{\beta_v} \{\mathsf{const}(2+40)\}\ 2
\end{aligned} \tag{2}$$

The first step replaces get and its delimited context 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.

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 number 2 were stored in a cell and accessed by get in the program get + 40. The reductions continue to a value:

$$\{\mathsf{const}\ 42\}\ 2 \rightsquigarrow_{\beta_v} \{\lambda z.\,42\}\ 2 \rightsquigarrow_{\{\}} (\lambda z.\,42)\ 2 \rightsquigarrow_{\beta_v} 42 \tag{3}$$

The reader may be reminded of the standard state-passing emulation of mutable state, in which every expression receives the current state as the argument. The just shown reductions have also occurred in the context of the application to the current state, 2. To be precise, the program like get + 40 is evaluated in the context $\{\mathsf{const}\ \square\}\ 2$ that when plugged with a state-invariant expression e will ignore the current state, delivering e's value. (One can easily see that if a program e contains no 出, then $\{e\}$ is observationally equivalent to e.) The expressions like get "reach out and grab" the current state from the context.

We can also mutate the state: the term put(get + 1) increments the number in the cell and returns the new number.

$$\begin{aligned}
\{\mathsf{const}\ (\mathsf{put}(\mathsf{get}+1)+\mathsf{get})\}\ 2 &\rightsquigarrow^{+} \{\mathsf{const}\ (\mathsf{put}(2+1)+\mathsf{get})\}\ 2 \\
&\rightsquigarrow^{+} \{\mathsf{const}\ (\mathsf{put}\ 3+\mathsf{get})\}\ 2 \\
&\rightsquigarrow^{+} (\lambda x.\,\{\mathsf{const}\ (x+\mathsf{get})\})\ 3\ 3 \\
&\rightsquigarrow_{\beta_v} \{\mathsf{const}\ (3+\mathsf{get})\}\ 3
\end{aligned} \tag{4}$$

This sequence of reductions replaces the term put(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 calls to put mutate 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` (for details, see the accompanying code in `circle-shift-1.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 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 provides this assurance both in the operational semantics (described in Section 3.3) and in the type system (described in Section 4). 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 Section 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:

$$\begin{aligned}
&\{\mathsf{const}\ \langle{\sim}(\mathsf{put}\langle 8+5\rangle)+{\sim}\mathsf{get}\rangle\}\ \langle 0\rangle\\
&\quad\rightsquigarrow_{\beta_v}\quad \{\mathsf{const}\ \langle{\sim}(\text{出}(\lambda k.\,\lambda z.\,k\langle 8+5\rangle\langle 8+5\rangle))+{\sim}\mathsf{get}\rangle\}\ \langle 0\rangle\\
&\quad\rightsquigarrow_{\text{出}_0}\quad \{(\lambda k.\,\lambda z.\,k\langle 8+5\rangle\langle 8+5\rangle)(\lambda x.\,\{\mathsf{const}\langle{\sim}x+{\sim}\mathsf{get}\rangle\})\}\ \langle 0\rangle \qquad (5)\\
&\quad\rightsquigarrow^{+}\quad \{\mathsf{const}\ \langle{\sim}\langle 8+5\rangle+{\sim}\mathsf{get}\rangle\}\ \langle 8+5\rangle\\
&\quad\rightsquigarrow_{\sim}\quad \{\mathsf{const}\ \langle(8+5)+{\sim}\mathsf{get}\rangle\}\ \langle 8+5\rangle \rightsquigarrow^{+} \langle(8+5)+(8+5)\rangle
\end{aligned}$$

Like in Section 3.2, $\mathsf{put}\langle 8+5\rangle$ assigns $\langle 8+5\rangle$ to the mutable cell, so get is later 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 Section 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 Section 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 $\lambda_1^{\oslash}$:

$$\{\mathsf{const}\ (\mathsf{let}\ x=\langle\lambda y.\,{\sim}(\mathsf{put}\langle y\rangle)\rangle\ \mathsf{in}\ \mathsf{get})\}\ \langle 0\rangle \qquad (6)$$

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 Section 4 rejects it statically. We prove that scope extrusion is impossible in Section 4.2.

The 出^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 Section 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 & Danvy 1994 in an untyped setting), we define the following alternative to put:

$$\mathsf{put}' = \lambda z'.\, \text{出}(\lambda k.\, \lambda z.\, \langle \mathsf{let}\ x = {\sim}z'\ \mathsf{in}\ {\sim}(k\langle x\rangle\langle x\rangle)\rangle) \tag{7}$$

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

$$\begin{aligned}
&\{\mathsf{const}\ \langle {\sim}(\mathsf{put}'\langle 8+5\rangle) + {\sim}\mathsf{get}\rangle\}\ \langle 0\rangle \\
&\quad \rightsquigarrow_{\beta_v} \quad \{\mathsf{const}\ \langle {\sim}(\text{出}(\lambda k.\, \lambda z.\, \langle \mathsf{let}\ x = {\sim}\langle 8+5\rangle\ \mathsf{in}\ {\sim}(k\langle x\rangle\langle x\rangle)\rangle)) + {\sim}\mathsf{get}\rangle\}\ \langle 0\rangle \\
&\quad \rightsquigarrow_{\text{出}_0} \quad \{(\lambda k.\, \lambda z.\, \langle \mathsf{let}\ x = {\sim}\langle 8+5\rangle\ \mathsf{in}\ {\sim}(k\langle x\rangle\langle x\rangle)\rangle)(\lambda y.\, \{\mathsf{const}\langle {\sim}y + {\sim}\mathsf{get}\rangle\})\}\ \langle 0\rangle \\
&\quad \rightsquigarrow^{+} \quad \langle \mathsf{let}\ x = 8+5\ \mathsf{in}\ {\sim}(\{\mathsf{const}\ \langle x + {\sim}\mathsf{get}\rangle\}\ \langle x\rangle)\rangle \\
&\quad \rightsquigarrow^{+} \quad \langle \mathsf{let}\ x = 8+5\ \mathsf{in}\ x + x\rangle
\end{aligned} \tag{8}$$

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-1.elf` for the complete code and tests). Roughly,[5] 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 second argument is a present-stage integer on which to specialize and recur. For example, recall the `sgib` function in Section 2.2:

$$\begin{aligned}
\mathsf{sgib} = \lambda x.\, \lambda y.\, \lambda \mathit{self}.\, \lambda n.\ &\mathsf{ifz}\ n\ \mathsf{then}\ x\ \mathsf{else} \\
&\mathsf{ifz}\ n-1\ \mathsf{then}\ y\ \mathsf{else} \\
&\langle {\sim}(\mathit{self}\,(n-1)) + {\sim}(\mathit{self}\,(n-2))\rangle
\end{aligned} \tag{9}$$

To specialize the Gibonacci function to $n = 5$, we evaluate[6]

$$\langle \lambda x.\, \lambda y.\, {\sim}(\{\mathsf{const}\ (\mathsf{y\_ms}\,(\mathsf{sgib}\,\langle x\rangle\langle y\rangle)\,5)\}\ \mathsf{empty})\rangle \tag{10}$$

[5] We suppress effect annotations in types until Section 4, where we introduce the type system formally.
[6] This example reveals that `top_fn` in Section 2.4 is $\lambda z.\, \{\mathsf{const}\,(z\,0)\}\,\mathsf{empty}$.

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

Fig. 5. The memoizing staged fixpoint combinator y_ms.

(where empty is the empty memoization table) to obtain

$$\begin{aligned}\langle \lambda x.\, \lambda y.\, &\mathsf{let}\ z_3 = y\ \mathsf{in}\ \mathsf{let}\ z_4 = x\ \mathsf{in}\ \mathsf{let}\ z_5 = z_3 + z_4\ \mathsf{in}\\ &\mathsf{let}\ z_6 = z_5 + z_3\ \mathsf{in}\ \mathsf{let}\ z_7 = z_6 + z_5\ \mathsf{in}\ z_7 + z_6\rangle.\end{aligned} \tag{11}$$

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

Figure 5 shows our definition of y_ms. (The reset on the last line of the code compensates for the lack of impredicative answer-type polymorphism in our system as Section 4.1 explains in detail.) 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 Section 2.1, which are pure functions and trivial to implement. Whereas `lookup` in Section 2.1 returns a value of type int option, our language $\lambda_1^{\oslash}$ does not include option, so we emulate the sum type τ option by a product type $(\mathsf{int}, \mathsf{int} \to \tau)$: 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 already generated for n. The lookup result (a pair) is bound to the variable x in Figure 5. If 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), then 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. We discuss these larger examples in Section 5.

If-insertion is also within reach. To use a simpler example than in Section 2.3, suppose that gen is a code generator in $\lambda_1^{\oslash}$ 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 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

$$\lambda n.\, \text{出}(\lambda k.\, \langle \mathsf{ifz}\ {\sim}n\ \mathsf{then}\ fail\ \mathsf{else}\ {\sim}(k\langle inverse\ {\sim}n\rangle)\rangle). \tag{12}$$

Passing this auxiliary generator to gen produces the desired code

$$\langle \lambda n.\, \mathsf{ifz}\ n\ \mathsf{then}\ \mathit{fail}\ \mathsf{else}\ e + \mathit{inverse}\ n \rangle. \tag{13}$$

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

Our language is too restrictive when a code generator needs to reach beyond the nearest generated binder to insert a let, if, or assert, or access a piece of mutable state. For example, if the code generated by f above takes a second argument m after n, then the test on n would be inserted under λm, even though it may save more computation to insert the test above λm. This situation can arise in Section 2.3 if the order of the arguments `array` and `n` is reversed in `gen`: we want to insert assertions as in

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

but we can only achieve

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

even though the assertion `n_2 >= 0` does not mention `array_1`. Similarly, for let-insertion: while generating the body of a loop that binds an index variable, we cannot insert a let-binding outside the loop even if the right-hand side of the let-binding does not mention the index variable. In other words, we cannot express loop-invariant code motion. Balat *et al.*'s (2004) use of control effects for normalization-by-evaluation of sums also needs to reach beyond generated binders and is unsupported by our language.

4 Type system

We have seen in Section 3.3 that attempting to generate code with scope extrusion in $\lambda_1^{\oslash}$ causes the generator to get stuck. We now describe the type system that statically prevents such a dynamic error. A well-typed generator shall not get stuck; an attempt at scope extrusion will be reported early, when type checking the generator rather than when running it. Figure 6 displays the type system of our language $\lambda_1^{\oslash}$. It combines a simplification of Danvy and Filinski's (1989) type system for delimited control and a simplification of Davies's (1996) type system for staging in a sound but not orthogonal way.

The types τ of $\lambda_1^{\oslash}$ 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 principal typings.

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

$$\frac{}{\Gamma \vdash n : \mathsf{int} \,;\, \tau_0 \,[;\, \upsilon_0]} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \,;\, \tau_0 \,[;\, \upsilon_0] \quad \Gamma \vdash e_2 : \mathsf{int} \,;\, \tau_0 \,[;\, \upsilon_0]}{\Gamma \vdash e_1 + e_2 : \mathsf{int} \,;\, \tau_0 \,[;\, \upsilon_0]}$$

$$\frac{\Gamma, x:\tau \vdash e : \tau' \,;\, \tau_1}{\Gamma \vdash (\lambda x.e) : \tau \to \tau'/\tau_1 \,;\, \tau_0} \qquad \frac{\Gamma, \langle x:\upsilon \rangle \vdash e : \upsilon' \,;\, \langle \upsilon'/\upsilon_1 \rangle \,;\, \upsilon_1}{\Gamma \vdash (\lambda x.e) : \upsilon \to \upsilon'/\upsilon_1 \,;\, \tau_0 \,;\, \upsilon_0}$$

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

$$\frac{\Gamma \vdash e_1 : \tau \to \tau'/\tau_0 \,;\, \tau_0 \quad \Gamma \vdash e_2 : \tau \,;\, \tau_0}{\Gamma \vdash e_1 e_2 : \tau' \,;\, \tau_0} \qquad \frac{\Gamma \vdash e_1 : \upsilon \to \upsilon'/\upsilon_0 \,;\, \tau_0 \,;\, \upsilon_0 \quad \Gamma \vdash e_2 : \upsilon \,;\, \tau_0 \,;\, \upsilon_0}{\Gamma \vdash e_1 e_2 : \upsilon' \,;\, \tau_0 \,;\, \upsilon_0}$$

$$\frac{\Gamma \vdash e : \tau \,;\, \tau_0 \,[;\, \upsilon_0] \quad \Gamma \vdash e' : \tau' \,;\, \tau_0 \,[;\, \upsilon_0]}{\Gamma \vdash (e, e') : (\tau, \tau') \,;\, \tau_0 \,[;\, \upsilon_0]}$$

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

$$\frac{\Gamma \vdash e : \mathsf{int} \,;\, \tau_0 \,[;\, \upsilon_0] \quad \Gamma \vdash e_1 : \tau \,;\, \tau_0 \,[;\, \upsilon_0] \quad \Gamma \vdash e_2 : \tau \,;\, \tau_0 \,[;\, \upsilon_0]}{\Gamma \vdash \mathsf{ifz}\, e \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2 : \tau \,;\, \tau_0 \,[;\, \upsilon_0]}$$

$$\frac{}{\Gamma \vdash \boxplus_1 : ((\tau \to \tau'/\tau_1) \to \tau'/\tau') \to \tau/\tau' \,;\, \tau_0 \,[;\, \upsilon_0]} \qquad \frac{\Gamma \vdash e : \tau \,;\, \tau}{\Gamma \vdash \{e\} : \tau \,;\, \tau_0} \qquad \frac{\Gamma \vdash e : \upsilon \,;\, \tau_0 \,;\, \upsilon}{\Gamma \vdash \{e\} : \upsilon \,;\, \tau_0 \,;\, \upsilon_0}$$

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

Fig. 6. The type system of $\lambda_1^{\oslash}$.

As a two-level language, $\lambda_1^{\oslash}$ operates on code values in the present stage only. Hence, the type of a future-stage expression never contains any code type. Terms such as $\langle\langle 42 \rangle\rangle$ are thus disallowed, and types such as $\langle\langle \mathsf{int} \,/\, \mathsf{int} \rangle / \mathsf{int} \rangle$ are uninhabited by any closed value. We reserve the metavariable υ for a type that contains no code type. A type environment Γ is a set of associations $x : \tau$ of present-stage variables x with types τ and associations $\langle x : \upsilon \rangle$ of future-stage variables x with types υ.

There are two judgment forms, one for present-stage expressions and another for future-stage expressions. Both forms include *answer types* to track the control effects that may occur: in a present judgment $\Gamma \vdash e : \tau \,;\, \tau_0$, 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.[7] 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 6 with

[7] For simplicity, we equate the two answer types distinguished by Danvy and Filinski (1989). It is easy but not necessary to restore the distinction. The distinction is useful – for example, to express *typed printf* (Danvy, 1998) in direct style with staging (Asai, 2009, personal communication).

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").

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 Section 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 Section 6.

Since answer types are effect annotations, they appear not only 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).

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[] \vdash \text{出} : S \ ; T} \quad \dfrac{\dfrac{\vdots}{k : (\mathsf{int} \to T/\tau_0) \vdash \lambda z.\, kzz : T \ ; T}}{[] \vdash \lambda k.\, \lambda z.\, kzz : (\mathsf{int} \to T/\tau_0) \to T/T \ ; T}}{[] \vdash \mathsf{get} : \mathsf{int} \ ; T}}{[] \vdash \mathsf{get} + 40 : \mathsf{int} \ ; T}}{[] \vdash \mathsf{const}\ (\mathsf{get} + 40) : T \ ; T}}{[] \vdash \{\mathsf{const}\ (\mathsf{get} + 40)\} : T \ ; \tau_0}}{[] \vdash \{\mathsf{const}\ (\mathsf{get} + 40)\}\ 2 : \mathsf{int} \ ; \tau_0} \tag{14}$$

The accompanying file `circle-shift-1.elf` type-checks many tests in Twelf. For example, the fixpoint combinator y_ms in Section 3.3 has the type

$$((\mathsf{int} \to \langle \upsilon/\upsilon_1 \rangle) \to \mathsf{int} \to \langle \upsilon/\upsilon_0 \rangle) \to \mathsf{int} \to \langle \upsilon/\upsilon_0 \rangle, \tag{15}$$

in which the present-stage answer types are all

$$(\mathsf{int} \to T/T) \to \langle \upsilon'/\upsilon_0 \rangle / \langle \upsilon'/\upsilon_0 \rangle, \tag{16}$$

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 Section 3.4.

4.1 Purity and answer-type polymorphism

Our type system has no polymorphism and hence is unable to straightforwardly represent answer-type polymorphism that is characteristic of pure expressions. The answer-type polymorphism is required in practice, even for our examples. Fortunately (somewhat inconvenient) roundabout ways of representing answer-type polymorphism are possible, which we discuss in this section. The inconvenience of emulating answer-type polymorphism is the drawback we share with the CPS/monadic style of encoding effectful generators in a system without impredicative polymorphism.

An expression is *pure* if it incurs no observable control effect; a pure expression is polymorphic in the answer type (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, since the expression $2 + 40$ incurs no observable control effect, it is safe to plug it 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 Section 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 $\mathsf{int} \to \tau'/\tau_1$ for some τ' and τ_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.

Our use of answer types to track control effects, like any effect system, exacerbates the need for polymorphism. To start with, all values are pure, and the soundness of our type system relies on their polymorphism in the answer type.

Lemma 1 If e is a value and $\Gamma \vdash e : \tau\ ;\ \tau_1\ [;\ \upsilon_0]$, then $\Gamma \vdash e : \tau\ ;\ \tau_2\ [;\ \upsilon_0]$.

Proof By induction on e and inversion on the derivation of $\Gamma \vdash e : \tau\ ;\ \tau_1\ [;\ \upsilon_0]$. (In our Twelf code `circle-shift-1.elf`, the constructive proof of the lemma is represented by total type families `val-new-at` and `val-new-at+`.) □

For example, each occurrence of a bound variable $x : \tau$ must have the same type τ but may have a different answer type τ_0, so subject reduction for our β_v and $\{\}$ rules relies on Lemma 1.

Values are just one particularly easy-to-identify class of pure expressions. Control delimiters also make an expression pure by masking its effect: In the typing rules for present-stage $\{e\}$ and future-stage $\lambda x.\, e$ in Figure 6, the answer type τ_0 is arbitrary. The latter rule reflects how future-stage binders delimit present-stage control in the operational semantics: The present-stage answer type is the code type $\langle \upsilon'/\upsilon_1 \rangle$ in the premise but arbitrary in the conclusion.

It is also pure to apply a pure function, but our type system does not represent the purity of functions except by hard-coding the answer-type polymorphism of "built-in functions" such as fix, fst, snd, and ifz into their typing rules (witness the type metavariables τ_0 and τ_1 in those rules). For example, each occurrence of $\mathsf{fix}\ v^0$ must have the same function type $\tau \to \tau'/\tau_2$ but may have a different answer type τ_1, so subject reduction for our fix rule relies on the trivial variant of Lemma 1 where e is replaced by $\mathsf{fix}\ v^0$.

Without impredicative answer-type polymorphism (Asai & Kameyama, 2007), our system never infers the purity of a user-defined function. Consequently, in order to

write the desired code as in Section 3.4, we are forced to build product types and int into the language rather than Church-encode them. (This task is not difficult; we could have introduced the desired `option` data type.) Furthermore, we sometimes need to annotate programs with additional resets. For example, the last line in Figure 5 contains a rather mysterious reset, without which the code would not type-check. The reason is that the lookup result x must have a type of the form $(\mathsf{int}, \mathsf{int} \to \langle \upsilon/\upsilon_0 \rangle/\tau_0)$, where τ_0 is *some* answer type. Without the reset, τ_0 would be unified with the answer type of the overall memoizing computation, which contains the type of the memoization table, which in turn contains the type of x, which causes an occurs-check failure. With the reset, the answer type τ_0 is unified with the return type $\langle \upsilon/\upsilon_0 \rangle$, which passes the occurs check. In general, we can always emulate a pure function type $\tau \to \tau'$ by an impure function type $\tau \to \tau'/\tau'$, at the cost of additional resets. (We could take advantage of let-polymorphism, were it present in our calculus, to infer the answer-type polymorphic type for pure functions. Let-polymorphism does not help in the above case since the pure function is stored in a data structure.)

The need to insert resets is a drawback – which is not however specific to our direct-style approach; *exactly the same issue* arises if we program in CPS/monadic style. We observe the problem even for the unstaged state-passing memoizing fixpoint combinator in OCaml, if we implement the memo table, with keys `'a` and values `'b` as we did in our calculus, with the type `('a -> bool * (unit -> 's -> ('s,'b)))`. The function of type `unit -> 's -> ('s,'b)`, essentially from `Some`, is written in state-passing style, for state of type `'s`. The memo table is the state of the overall computation, thus `'s` must be the type of the whole table – hence the occurs-check failure. The solution is to note that the projection function must be pure, that is, does not affect the state; to indicate the purity, we wrap the projection function into "reset," defined in our case as `fun m -> fun s -> (s,snd (m empty))`. The problem does not arise if we avail ourselves to the built-in option type; the projection function from `Some` is manifestly pure. In our extensive practical experience of programming in both CPS/monadic style and direct style, the problem of lack of polymorphism and the need to introduce reset is exceedingly rare.

4.2 Formal properties

We are ready to formalize the safety assurances provided by our type system. The formal properties of $\lambda_1^{\mathcal{O}}$ and their proofs have been mechanized in the accompanying Twelf code; see the file `circle-shift-1.elf`. In the present section, we restate these properties in mathematical notation and draw their corollaries.

Lemma 2 (type substitution) Let θ be a substitution on type variables. If the judgment $\Gamma \vdash e : \tau \; ; \; \tau_0 \; [; \upsilon_0]$ is derivable and $\Gamma' \supseteq \Gamma\theta$, then the judgment $\Gamma' \vdash e : \tau\theta \; ; \; \tau_0\theta \; [; \upsilon_0\theta]$ is derivable.

Proof By induction on the derivation of $\Gamma \vdash e : \tau \; ; \; \tau_0 \; [; \upsilon_0]$. For example, suppose $\Gamma \vdash x : \tau \; ; \; \tau_0$. By inversion, $(x : \tau) \in \Gamma$. Thus, $(x : \tau\theta) \in \Gamma\theta \subseteq \Gamma'$, so $\Gamma' \vdash x : \tau\theta \; ; \; \tau_0\theta$. □

Proposition 3 (principal typing) If some judgment $\Gamma \vdash e : \tau \ ; \tau_0 \ [; \upsilon_0]$ is derivable, then some judgment $\hat{\Gamma} \vdash e : \hat{\tau} \ ; \hat{\tau_0} \ [; \hat{\upsilon_0}]$ is derivable such that, for any derivable judgment $\Gamma' \vdash e : \tau' \ ; \tau'_0 \ [; \upsilon'_0]$, there exists a substitution θ for type variables so that $\Gamma' \supseteq \hat{\Gamma}\theta$, $\tau' = \hat{\tau}\theta$, and $\tau'_0 = \hat{\tau_0}\theta$ [and $\upsilon'_0 = \hat{\upsilon_0}\theta$].

Proof By induction on e, essentially reading our syntax-directed type system as a bottom-up logic program that infers the principal typing using unification. We show two cases.

Case $e = x$ at the present stage: Set $\hat{\tau}$ and $\hat{\tau_0}$ to fresh type variables, and $\hat{\Gamma}$ to $x : \hat{\tau}$.

Case $e = \lambda x.\, e_1$ at the present stage: We want to prove that, given $\Gamma \vdash \lambda x.\, e_1 : (\tau_1 \to \tau_2/\tau_3) \ ; \tau_0$ is derivable, some judgment $\hat{\Gamma} \vdash \lambda x.\, e_1 : (\hat{\tau_1} \to \hat{\tau_2}/\hat{\tau_3}) \ ; \hat{\tau_0}$ is derivable such that, for any derivable $\Gamma' \vdash \lambda x.\, e_1 : (\tau'_1 \to \tau'_2/\tau'_3) \ ; \tau'_0$, there is a substitution θ such that $\Gamma' \supseteq \hat{\Gamma}\theta$ and $\tau'_i = \hat{\tau_i}\theta$ $(i = 0, 1, 2, 3)$. By inversion, we get $\Gamma, x : \tau_1 \vdash e_1 : \tau_2 \ ; \tau_3$ is derivable. By induction hypothesis, we have a principal derivation $\check{\Gamma} \vdash e_1 : \check{\tau_2} \ ; \check{\tau_3}$. We have the following two subcases:

Subcase $(x : \check{\tau_1}) \in \check{\Gamma}$ for some $\check{\tau_1}$: Set $\hat{\Gamma} = \check{\Gamma} - (x : \check{\tau_1})$ and $\hat{\tau_i} = \check{\tau_i}$ $(i = 1, 2, 3)$. Let $\hat{\tau_0}$ be a fresh type variable, then $\hat{\Gamma} \vdash \lambda x.\, e_1 : \hat{\tau_1} \to \hat{\tau_2}/\hat{\tau_3} \ ; \hat{\tau_0}$ is derivable. We also have that for each derivable judgment $\Gamma' \vdash \lambda x.\, e_1 : \tau'_1 \to \tau'_2/\tau'_3 \ ; \tau'_0$ there exists a derivable judgment $\Gamma', x : \tau'_1 \vdash e_1 : \tau'_2 \ ; \tau'_3$. By induction hypothesis, there exists a substitution θ' such that $(\Gamma', x : \tau'_1) \supseteq \check{\Gamma}\theta'$ and $\tau'_i = \check{\tau_i}\theta'$ $(i = 2, 3)$. The former implies $\tau'_1 = \check{\tau_1}\theta'$. By setting $\theta = \theta'[\hat{\tau_0} \mapsto \tau'_0]$, we have $\Gamma' \supseteq \hat{\Gamma}\theta$ and $\tau'_i = \hat{\tau_i}\theta$ $(i = 0, 1, 2, 3)$.

Subcase $(x : \check{\tau_1}) \notin \check{\Gamma}$ for any $\check{\tau_1}$: Set $\hat{\Gamma} = \check{\Gamma}$, $\hat{\tau_i} = \check{\tau_i}$ (for $i = 2, 3$), and let $\hat{\tau_0}$ and $\hat{\tau_1}$ be fresh type variables. Then we can prove this subcase in the same way as the previous one. □

The proof of subject reduction relies on the following three lemmas.

Lemma 4 (value substitution) If $\Gamma, x : \tau \vdash e : \tau' \ ; \tau_0 \ [; \upsilon_0]$ and $\Gamma \vdash v^0 : \tau \ ; \tau_1$, then $\Gamma \vdash e[x := v^0] : \tau' \ ; \tau_0 \ [; \upsilon_0]$.

The proof is routine, relying on Lemma 1.

Lemma 5 (weakening) If $\Gamma \vdash e : \tau \ ; \tau_0 \ [; \upsilon_0]$, then $\Gamma, \Delta \vdash e : \tau \ ; \tau_0 \ [; \upsilon_0]$.

The proof is also routine.

Lemma 6 (abstraction) If $\Gamma \vdash e : \tau_1 \ ; \tau_0$ and $\Gamma \vdash D^{00}[e] : \tau_2 \ ; \tau_0$ are derivable, then so is $\Gamma, x : \tau_1 \vdash D^{00}[x] : \tau_2 \ ; \tau_0$ for a fresh variable x.

Proof A straightforward induction on the context D^{00}. □

Proposition 7 (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 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 couple of 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 $\Gamma \vdash v^0 : \tau \ ; \tau_0$ for some τ. Then the substitution lemma gives $\Gamma \vdash e[x := v^0] : \tau' \ ; \tau_0$.

Case $出^0$: We have the derivation

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash 出 : T \to \tau/\tau' \ ; \tau'} \quad \begin{array}{c}\vdots\\ \Gamma \vdash v^0 : T \ ; \tau'\end{array}}{\Gamma \vdash 出 v^0 : \tau \ ; \tau'}}{\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash D^{00}[出 v^0] : \tau' \ ; \tau'\end{array}}{\Gamma \vdash \{D^{00}[出 v^0]\} : \tau' \ ; \tau_0}} \tag{17}$$

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 and abstraction lemmas to derive

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash v^0 : T \ ; \tau'\end{array} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : \tau \vdash x : \tau \ ; \tau'}}{\begin{array}{c}\vdots\\ \Gamma, x : \tau \vdash D^{00}[x] : \tau' \ ; \tau'\end{array}}}{\Gamma, x : \tau \vdash \{D^{00}[x]\} : \tau' \ ; \tau_1}}{\Gamma \vdash \lambda x.\{D^{00}[x]\} : \tau \to \tau'/\tau_1 \ ; \tau'}}{\dfrac{\Gamma \vdash v^0(\lambda x.\{D^{00}[x]\}) : \tau' \ ; \tau'}{\Gamma \vdash \{v^0(\lambda x.\{D^{00}[x]\})\} : \tau' \ ; \tau_0}} \tag{18}$$

The $出^1$ case is similar. □

Corollary 8 (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'. □

To state the next two propositions, we define the future-stage type environment $\langle \Gamma \rangle = \langle x_1 : v_1 \rangle, \ldots, \langle x_n : v_n \rangle$ whenever $\Gamma = x_1 : v_1, \ldots, x_n : v_n$.

Proposition 9 (progress) If $\langle \Gamma \rangle \vdash e : \tau \ ; \tau$ is derivable, then there exists a term e' such that $\{e\} \rightsquigarrow e'$.

Proof Define a *pre-redex* p^i to be an expression of the following form:

$$p^0 \ ::= \ v^0 + v^0 \ \mid \ v^0 v^0 \ \mid \ \mathsf{ifz}\ v^0\ \mathsf{then}\ e\ \mathsf{else}\ e \ \mid \ \{v^0\}, \qquad p^1 \ ::= \ {\sim} v^0. \tag{19}$$

We slightly generalize the assumption to $\langle \Gamma \rangle \vdash e : \tau \ ; \tau_1$. By induction on its derivation, either e is already a value ($e = v^0$), or e can be decomposed into a context plugged with a pre-redex ($e = C^i[p^i]$).

If e is already a value, then $\{e\} \rightsquigarrow e$. Otherwise, the reduction required follows from case analysis on the typing derivation of the pre-redex. For example, if $e = C^1[{\sim} v^0]$, then v^0 must be of the form $\langle v^1 \rangle$, so $\{e\} \rightsquigarrow \{C^1[v^1]\}$. In the case where $e = C^0[出 v^0]$, we choose between the $出^0$ and $出^1$ reduction rules by case analysis on C^0. □

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 future-stage environment $\langle \Gamma \rangle$, in order for the induction on

the typing derivation of e to go through. This strengthening corresponds to the world declaration `bl-ev` in the accompanying Twelf code.

Propositions 7 and 9 together show that well-typed programs never go wrong in $\lambda_1^{\bigcirc}$. In particular, well-typed code generators never go wrong. Moreover, by the following argument any code they generate never goes wrong either. 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 10 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 $\langle \Gamma \rangle \vdash v^1 : \upsilon \ ; \tau_0 \ ; \upsilon_0$. □

As a corollary of the previous proposition, we obtain the following result, which is similar to the binding-time correctness theorem of Davies (1996).

Corollary 11 If $[] \vdash \{e\} : \langle \upsilon / \upsilon_0 \rangle \ ; \tau_0$ and $\{e\} \rightsquigarrow^* v^0$ for a value v^0, then v^0 has the form $\langle v^1 \rangle$ such that $[] \vdash v^1 : \upsilon \ ; \upsilon_0$.

Proof By subject reduction, $[] \vdash v^0 : \langle \upsilon / \upsilon_0 \rangle \ ; \tau_0$. By inversion, we have $v^0 = \langle v^1 \rangle$ for some v^1 such that $[] \vdash v^1 : \upsilon \ ; \tau_0 \ ; \upsilon_0$. Then by the previous proposition, $[] \vdash v^1 : \upsilon \ ; \upsilon_0$. □

5 Larger examples of program specialization

In this section, we give larger examples to demonstrate the applicability of our combination of side effects and code generation. These examples show that our approach does not require the open-recursion style and its utility goes far beyond dynamic programming and even let-insertion.

5.1 Longest common subsequence

The longest common subsequence is one of the textbook dynamic-programming algorithms in Swadi *et al.*'s (2005) benchmark. The problem is, given two sequences, to find (the length of) the longest subsequence of elements that occur in both sequences in the same order, though not necessarily contiguously. The naive implementation of the algorithm

```
let rec lcs x y (i,j) =
  if i = 0 || j = 0 then 0
  else if x.(i) = y.(j) then 1 + lcs x y (i-1,j-1)
       else max (lcs x y (i,j-1)) (lcs x y (i-1,j))
```

takes two arrays `x` and `y` as input and computes the length of the longest common subsequence of `x[1..i]` and `y[1..j]`. The input arrays can be emulated with functions in $\lambda_1^{\bigcirc}$. This program is very inefficient. To make it efficient, the programmer only needs to rewrite it in the open-recursion style:

```
let lcs x y self (i,j) =
  if i = 0 || j = 0 then 0
  else if x.(i) = y.(j) then 1 + self (i-1,j-1)
       else max (self (i,j-1)) (self (i-1,j))
```

The rewriting keeps the algorithm clear. Applying the memoizing fixpoint combinator `y_memo_m` (described in Sections 2.1 and 3.2) gives the efficient implementation.

To specialize the algorithm when the lengths of the inputs are statically known (although their contents are not), we merely need to add brackets and escapes to this `lcs` code:

```
let slcs x y self (i,j) =
  if i = 0 || j = 0 then .<0>.
  else .<if (.~x).(i) = (.~y).(j) then 1 + .~(self (i-1,j-1))
         else max .~(self (i,j-1)) .~(self (i-1,j))>.
```

Applying the memoizing staged fixpoint combinator `y_ms` described in Sections 2.4 and 3.4

```
.<fun x y -> .~(top_fn (fun _ -> y_ms (slcs .<x>. .<y>.) (3,4)))>.
```

produces the ideal code

```
.<fun x_1 -> fun y_2 ->
  let t_3 = 0 in let t_4 = 0 in let t_5 = 0 in
  let t_6 = 0 in let t_7 = 0 in let t_8 = 0 in
  let t_9  = if x_1.(1) = y_2.(1) then 1+t_8 else max t_7 t_6 in
  let t_10 = if x_1.(1) = y_2.(2) then 1+t_6 else max t_9 t_5 in
  let t_11 = if x_1.(1) = y_2.(3) then 1+t_5 else max t_10 t_4 in
  let t_12 = if x_1.(1) = y_2.(4) then 1+t_4 else max t_11 t_3 in
  ...
  let t_19 = if x_1.(3) = y_2.(1) then 1+t_13 else max t_18 t_14 in
  let t_20 = if x_1.(3) = y_2.(2) then 1+t_14 else max t_19 t_15 in
  let t_21 = if x_1.(3) = y_2.(3) then 1+t_15 else max t_20 t_16 in
  if x_1.(3) = y_2.(4) then 1+t_16 else max t_21 t_17>.
```

The complexity is the optimal $O(n \times m)$, where n and m are the lengths of the inputs.

For comparison, to use Swadi *et al.*'s (2006) dynamic-programming specialization framework, Swadi *et al.* (2005) have to write the staged `lcs` code in a less direct monadic style:

```
let lcs_mks x y self (i,j) =
  if i = 0 || j = 0 then ret .<0>.
  else bind (self (i-1,j-1)) (fun r1 ->
       bind (self (i  ,j-1)) (fun r2 ->
       bind (self (i-1,j  )) (fun r3 ->
       ret .<if (.~x).(i) = (.~y).(j) then 1 + .~r1
             else max .~r2 .~r3>.)))
```

Other dynamic-programming algorithms can be specialized in the same way. For example, to compute the product of a sequence of matrices while minimizing the number of multiplications, we have specialized the minimization when the number of matrices is known (although their dimensions and contents are not).

5.2 Gaussian elimination

Let-insertion is useful not just for dynamic programming. Carette and Kiselyov (2011) implemented Gaussian elimination as a code generator that produces a routine specialized to a particular choice of numeric representation and matrix pivoting. Among the many parameters of this code generator is whether to compute the determinant. If the determinant is desired, the generated code should include determinant accumulation, and, more importantly, let-bindings for mutable cells in which to accumulate the determinant's sign and magnitude. Carette and Kiselyov (2011) achieved this let-insertion by writing the entire generator in monadic style. We briefly describe writing the generator for a family of Gaussian eliminations in our approach, highlighting the modularity and the close similarity of the generator to the textbook, unstaged code (written, for example, by an expert in linear algebra).

We start with the unstaged OCaml code for Gaussian elimination, whose fragment is shown below. (See `ge_unstaged.ml` in the accompanying code for the complete implementation and tests.) The code implements the standard textbook pseudo-code description of the algorithm in the most straightforward way. In the loop, we search for the pivot, exchange the current row and column with the pivot's row and column to bring the pivot to the matrix diagonal, and then perform the row reduction. Swapping rows or columns in a matrix changes the sign of its determinant; we keep track of the sign in the mutable cell `det_sign`.

```
let ge = fun a_orig ->
    let r = ref 0 in                    (* current row index    *)
    let c = ref 0 in                    (* current column index *)
    let a = Array.copy (a_orig.arr) in  (* save input matrix A  *)
    let m = a_orig.m in                 (* the number of cols   *)
    let n = a_orig.n in                 (* the number of rows   *)
    let det_sign = ref 1 in             (* Accumulate sign and  *)
    let det_magn = ref 1.0 in           (*   magnitude of det   *)
    while !c < m && !r < n do
        let pivot = find_pivot_row a (n,m) !r !c in
        let piv_val = (match pivot with
        | Some ((piv_r, piv_c),piv_val) ->
            if piv_c <> !c then begin
                swap_cols a (n,m) !c piv_c;
                det_sign := - !det_sign
            end;
            if piv_r <> !r then ...;
            Some piv_val
        | None -> None) in
        (* do the row-reduction over the (r,c)-(n,m) block *)
        ...
```

The following is our generator for Gaussian elimination, which generates the unstaged code above, among many other variants. (The complete generator code is

in the file `ge_gen.ml`.)

```
let gge outchoice = .<fun a_orig ->
    let r = ref 0 in                    (* current row index    *)
    let c = ref 0 in                    (* current column index *)
    let a = Array.copy (a_orig.arr) in  (* save input matrix A  *)
    let m = a_orig.m in                 (* the number of cols   *)
    let n = a_orig.n in                 (* the number of rows   *)
 .~(let p = new_prompt () in
    push_prompt p (fun () ->
    let env = {env_p = p; env_a = .<a>.; env_n = .<n>.;
               env_m = .<m>.; env_r = .<r>.;
               env_pivot = gfind_pivot_row;
               env_det   = make_det_nodet p} in
    let env = outchoice.oc_init env in
  .<begin
    while !c < m && !r < n do
        let pivot = .~(env.env_pivot env .<!r>. .<!c>.) in
        let piv_val = (match pivot with
        | Some ((piv_r, piv_c),piv_val) ->
            if piv_c <> !c then begin
                 .~(gswap_cols env .<!c>. .<piv_c>.);
                 .~(env.env_det.det_flip_sign)
            end;
            if piv_r <> !r then ...;
            Some piv_val
        | None -> None) in
        (* do the row-reduction over the (r,c)-(n,m) block *)
     ... >.
```

The generator is strikingly similar to the unstaged code; in fact, it was written by wrapping the unstaged code into a pair of brackets and placing a few escapes at places where inlining (e.g., of the row swapping) or variant implementations were desired. The main difference between the generator and the unstaged code is the factoring-out of pivoting and determinant computations. The unstaged code used row pivoting, implemented by a function `find_pivot_row`. The generator inlines the result of the pivoting generator `env.env_pivot`, which defaults to the row-pivoting generator but can be changed to a full-pivoting generator, for example. Textbooks on Gaussian elimination describe several pivoting strategies and conditions for using them. The user specifies the desired variant of Gaussian elimination by passing `gge` a value of the data type

```
type ('c,'result,'out) outchoice =
    {oc_init : ('c,'result) env -> ('c,'result) env;
     oc_fin  : ('c,'result) env -> ('c,'out) code;
   }
```

that sets the generation environment `env` and specifies the generation of the final result. The generator `gge` may produce code that returns the complete LU decomposition, just the U factor, the U factor and the determinant, the U factor and the rank, etc. (The result type of the generated code varies accordingly.) Carette and Kiselyov (2011) thoroughly discuss various aspects and parametrizations of Gaussian elimination.

Of interest to us here is how the determinant is computed. That computation is spread throughout the Gaussian elimination code: defining mutable cells to be used; flipping the sign upon an exchange of rows or columns; and accumulating the magnitude during row reductions. When we converted the unstaged code to the generator, we abstracted determinant computations away, replacing, for example, `det_sign := - !det_sign` in the unstaged code with the invocation of the corresponding generator `env.env_det.det_flip_sign`. The generators for flipping the sign etc. are collected in a data structure `'c det`:

```
type 'c det = {det_computes  : bool;
               det_flip_sign : ('c,unit) code;
               ...}
```

The field `env.env_det` of the generation environment holds an instance of `'c det` to be used when generating a particular version of Gaussian elimination.

There are two instances of the `'c det` data structure. One generates no determinant computation:

```
let make_det_nodet p =
  {det_computes  = false;
   det_flip_sign = .<()>.; ...}
```

The generator `det_flip_sign` generates `()` then, effectively no code. More interesting is the instance that does generate determinant computation:

```
let genlet p e =
  shift p (fun k -> .<let t = .~e in .~(k .<t>.)>.);;
let make_det_compute_det p =
  let det_sign = genlet p .<ref 1>. in
  let det_magn = genlet p .<ref 1.0>. in
  {det_computes  = true;
   det_flip_sign = .<.~det_sign := - !(.~det_sign)>.;
   ...}
```

Here, `det_flip_sign` generates the code that we have abstracted away from the unstaged original. To use that code however, we have to first generate definitions of the variables `det_sign` and `det_magn`. The `genlet` combinator lets us generate let-bindings for these mutable cells at the place indicated at `push_prompt` (or, reset). In our case that place is after all the other let-bindings at the beginning of the Gaussian elimination code.

The determinant aspect is an example of the modularity and expressivity that is required for real-world generators and is afforded in our approach. We abstracted

not only expressions (such as flipping the sign) but also definitions of (private) variables used within the abstracted expressions. The abstractions carry no run-time penalty for the generated code. Our approach, despite its restriction, does allow writing useful modular generators.

5.3 Markov models

Delimited control is not just useful for let-insertion. Taha[8] used MetaOCaml to specialize the execution of, and thus speed up the search for, Markov models of a given form – whose number of states is known and many transition probabilities are known to be zero. The Markov model is expressed using mutable state to initialize and multiply matrices.

Whereas naively adding mutable state to MetaOCaml risks scope extrusion, our system allows this use of mutable state and assures it safe because all of the matrix operations take place in the same generated scope. Our MetaOCaml implementation relying on delimited control operators can be found in the file `band_markov_lc.ml` of the accompanying code.

6 CPS translation

In this section, we define a CPS translation for $\lambda_1^{\oslash}$. The translation maps terms and types from $\lambda_1^{\oslash}$ to a multilevel calculus without control effects. We show that the translation simulates $\lambda_1^{\oslash}$ – in other words, it preserves operational semantics. We can thus understand the reduction semantics in Section 3 and the type system in Section 4 in terms of pure staging: For example, the $出^0$ and $出^1$ reductions in Figure 3 correspond to the translation $(\!|出|\!)$ in Figure 8; and the types of $\lambda_1^{\oslash}$ pull back those of the target language. The translation also lets us implement $\lambda_1^{\oslash}$ by implementing the target language.

The target language of our CPS translation 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, for instance, $\Gamma \vdash e : \upsilon \;;;\; \upsilon_0$ for a future-stage judgment. We equate any level-0 expression e of code type with $\langle\sim e\rangle$, and any level-1 expression e with $\sim\langle e\rangle$.[9] These equations bring our treatment of level switching closer to that in the two-level λ-calculus (Nielson

[8] `http://metaocaml.org/examples/band-markov.ml`

[9] The first equation $e = \langle\sim e\rangle$ is crucial in Equation (43). It forces us to turn the $\sim$ reduction in Figure 3 into the second equation $e = \sim\langle e\rangle$ in order to avoid the infinite loop $\langle\sim\langle 3\rangle\rangle \rightsquigarrow \langle 3\rangle = \langle\sim\langle 3\rangle\rangle$. Besides, to prove Proposition 15, we need to reduce the target term $\sim\langle v^1\rangle$ in zero or more steps to v^1 everywhere – even under a present-stage λ, because $\bar{k}\ @\ \sim\langle v^1\rangle$ in Equation (41) may put $\sim\langle v^1\rangle$ under a present-stage λ. For example, we need to translate the second step in

$$\begin{aligned}\{\langle\sim(\mathsf{let}\ x = 0\ \mathsf{in}\ \langle 3\rangle) + \sim(出\lambda f.\ldots)\rangle\} &\rightsquigarrow_{\beta_v} \{\langle\sim\langle 3\rangle + \sim(出\lambda f.\ldots)\rangle\}\\ &\rightsquigarrow_{\sim} \{\langle 3 + \sim(出\lambda f.\ldots)\rangle\}\\ &\rightsquigarrow_{出^0} \{\mathsf{let}\ f = \lambda y.\{\langle 3 + \sim y\rangle\}\ \mathsf{in}\ \ldots\}\end{aligned}$$

to zero or more steps from $\sim\langle 3\rangle$ to 3 under a present-stage λ (namely λy, essentially).

Take for example the source program

$$\langle(\sim e, \sim e)\rangle,$$

in which

$$e \;=\; 出\lambda f.\langle \mathsf{let}\ x = 1+1\ \mathsf{in}\ \sim(f\langle x\rangle)\rangle$$

for short. We calculate

$$\begin{array}{rcl}[\![e]\!]_0 & = & \bar{\lambda}\bar{k}.\,\mathsf{let}\ \ g = \lambda f.\lambda k.[\![\lambda x.\sim(f\langle x\rangle)]\!]_1 \mathbin{\bar{@}} \bar{\lambda}\bar{x}.k\langle\bar{x}(1+1)\rangle \\ & & \quad\ \mathsf{and}\ k = \lambda x.\bar{k} \mathbin{\bar{@}} x \\ & & \quad\ \ \mathsf{in}\ g(\lambda x.\lambda k'.(\lambda x'.k'x')(kx))(\lambda x.x),\end{array}$$

in which the body $[\![\lambda x.\sim(f\langle x\rangle)]\!]_1 \mathbin{\bar{@}} \bar{\lambda}\bar{x}.k\langle\bar{x}(1+1)\rangle$ is equal to the target term

$$\mathsf{let}\ z = \langle\lambda x.\sim(f\langle x\rangle\lambda z.z)\rangle\ \mathsf{in}\ k\langle\sim z(1+1)\rangle.$$

So, for all $\bar{k}$, the target term $[\![e]\!]_0 \mathbin{\bar{@}} \bar{k}$ reduces by 9 uses of the reduction (20) to

$$\bar{\lambda}\bar{k}.\,\mathsf{let}\ z = \langle\lambda x.\sim((\lambda x'.x')(\bar{k} \mathbin{\bar{@}} \langle x\rangle))\rangle\ \mathsf{in}\ \langle\sim z(1+1)\rangle.$$

Because the reduction (20) applies everywhere, the CPS transform

$$[\![\langle(\sim e, \sim e)\rangle]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{x}.\bar{x}$$

of the source program $\langle(\sim e, \sim e)\rangle$ reduces by 18 uses of the reduction (20) to the term (23), which in turn reduces to the value (22).

Fig. 7. An example of applying the one-pass CPS translation to perform let-insertion.

& Nielson, 1988). We also consider $\sim v^0$ (including $\sim x$) a value. The reductions of the target are

- the reductions of $\lambda_1^{\oslash}$ (Figure 3) restricted to these terms,
- minus the now-superfluous $\sim$ reduction,
- plus the reduction

$$(\lambda x.\,e)\,v^0 \to e[x := v^0] \quad \text{where } x \text{ occurs at most once in } e \tag{20}$$

everywhere – even under a present-stage λ. This additional reduction does not affect the set of terminating terms.

Since the sets of values, frames, delimited contexts, and contexts (Figure 2) are invariant up to the equations $e = \langle\sim e\rangle$ and $e = \sim\langle e\rangle$ just introduced, these reductions are well-defined over equivalence classes of target terms.

Before presenting our CPS translation in detail, we first show an example of it at work. Take the source program

$$\langle\ (\sim(出\lambda f.\langle \mathsf{let}\ x = 1+1\ \mathsf{in}\ \sim(f\langle x\rangle))),\ \ \sim(出\lambda f.\langle \mathsf{let}\ x = 1+1\ \mathsf{in}\ \sim(f\langle x\rangle)))\ \rangle \tag{21}$$

in $\lambda_1^{\oslash}$. This program performs two let-insertions in a row. When placed inside a top-level reset, it evaluates to

$$\langle \mathsf{let}\ x = 1+1\ \mathsf{in}\ \mathsf{let}\ y = 1+1\ \mathsf{in}\ (x, y)\rangle \tag{22}$$

as specified in Figure 3. As detailed in Figure 7, our CPS translation maps the

program (21) to a target term that reduces to

$$\begin{array}{l}\mathsf{let}\ z = \langle \lambda x.\sim((\lambda x'.\, x')(\\ \qquad \mathsf{let}\ z = \langle \lambda y.\sim((\lambda x'.\, x')(\\ \qquad\qquad \langle (x, y)\rangle \\ \qquad))\rangle\ \mathsf{in}\ \langle \sim z(1+1)\rangle \\ ))\rangle\ \mathsf{in}\ \langle \sim z(1+1)\rangle\end{array} \tag{23}$$

using the reduction (20). This term does not use any delimited control operators, but rather uses ordinary abstractions over $\langle x\rangle$ and $\langle y\rangle$ to represent the continuations of the two 出-applications in the source program (21). This term eventually reduces to the value (22) just as the source program does.

Figure 8 shows the formal rules of our CPS translation. It is a one-pass translation (that is, it produces no administrative redexes), which makes it simpler to state and prove that it simulates the source language. Like Danvy and Filinski (1992), we express it in a higher-order metalanguage and regard it as the result of analyzing the binding times in a CPS translation that does produce administrative redexes. We write $\bar{x}$, $\bar{\lambda}$, and $\bar{@}$ to denote variables, abstraction, and application at the level of this metalanguage.

Figure 8 defines the translation $[\![e]\!]_0$ of a present-stage expression e and the translation $[\![e]\!]_1$ of a future-stage expression e. Given a present-stage value v^0, the figure also defines its value translation $(\!|v^0|\!)$, which is itself a value in the target language. For example, $(\!|\lambda x.\, e|\!)$ is a target term of the form $\lambda x.\, \lambda k. \ldots$ where x and k are fresh variable names; in particular, k is a dynamic continuation. In contrast, $[\![e]\!]_0$ and $[\![e]\!]_1$ are meta-level functions that map a term-to-term function $\bar{k}$ (a static continuation) to a term. More precisely,

- the meta-level function $[\![e]\!]_0$ returns a present-stage term when given a meta-level function mapping a present-stage value to a present-stage term;
- the meta-level function $[\![e]\!]_1$ returns a present-stage term when given a meta-level function mapping a future-stage value to a present-stage term.

The definition of $[\![e]\!]_0$ uses the static continuation $\uparrow k$ (pronounced "reflect k") and the dynamic continuation $\downarrow \bar{k}$ (pronounced "reify $\bar{k}$"). If we ignore the difference of object language and metalanguage, then reflection and reification do not change the continuation at all. We make the distinction and use the metalanguage level to avoid producing administrative redexes and simplify the proof of simulation.

The output of the translation is in tail form except in $[\![\{e\}]\!]_0$, $(\!|出|\!)$, and $[\![\lambda x.\, e]\!]_1$. If we naively define $[\![\lambda x.\, e]\!]_1 = \bar{\lambda}\bar{k}.\, [\![e]\!]_1\ \bar{@}\ \bar{\lambda}\bar{z}.\, \bar{k}\ \bar{@}\ \lambda x.\, \bar{z}$ 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.

To make this description precise and to show that the translation preserves types in $\lambda_1^{\bigcirc}$, we define a CPS translation τ^* of present-stage types τ.

$$\begin{array}{rr}\mathsf{int}^* = \mathsf{int} \qquad \langle \upsilon/\upsilon_0\rangle^* = \langle \upsilon/\upsilon_0\rangle & (\tau_1, \tau_2)^* = (\tau_1^*, \tau_2^*) \\ (\tau_1 \to \tau_2/\tau_3)^* = \tau_1^* \to (\tau_2^* \to \tau_3^*) \to \tau_3^* & \alpha^* = \alpha\end{array} \tag{24}$$

Present-stage term translation $[\![e]\!]_0$

$$
\begin{aligned}
[\![v^0]\!]_0 &= \bar{\lambda}\bar{k}.\,\bar{k}\,\bar{@}\,(\!|v^0|\!)\\
[\![e_1+e_2]\!]_0 &= \bar{\lambda}\bar{k}.\,[\![e_1]\!]_0\,\bar{@}\,\bar{\lambda}\bar{x}.\,[\![e_2]\!]_0\,\bar{@}\,\bar{\lambda}\bar{y}.\downarrow\bar{k}(\bar{x}+\bar{y})\\
[\![e_1e_2]\!]_0 &= \bar{\lambda}\bar{k}.\,[\![e_1]\!]_0\,\bar{@}\,\bar{\lambda}\bar{f}.\,[\![e_2]\!]_0\,\bar{@}\,\bar{\lambda}\bar{x}.\,\bar{f}\bar{x}\downarrow\bar{k}\\
[\![(e_1,e_2)]\!]_0 &= \bar{\lambda}\bar{k}.\,[\![e_1]\!]_0\,\bar{@}\,\bar{\lambda}\bar{x}.\,[\![e_2]\!]_0\,\bar{@}\,\bar{\lambda}\bar{y}.\,\bar{k}\,\bar{@}\,(\bar{x},\bar{y})\\
[\![\mathsf{if}\,z\,e\,\mathsf{then}\,e_1\,\mathsf{else}\,e_2]\!]_0 &= \bar{\lambda}\bar{k}.\,[\![e]\!]_0\,\bar{@}\,\bar{\lambda}\bar{x}.\,\mathsf{let}\,k=\downarrow\bar{k}\,\mathsf{in}\,\mathsf{if}\,\bar{x}\,\mathsf{then}\,[\![e_1]\!]_0\,\bar{@}\uparrow k\,\mathsf{else}\,[\![e_2]\!]_0\,\bar{@}\uparrow k\\
[\![\langle e\rangle]\!]_0 &= \bar{\lambda}\bar{k}.\,[\![e]\!]_1\,\bar{@}\,\bar{\lambda}\bar{x}.\,\bar{k}\,\bar{@}\,\langle\bar{x}\rangle\\
[\![\{e\}]\!]_0 &= \bar{\lambda}\bar{k}.\downarrow\bar{k}([\![e]\!]_0\,\bar{@}\,\bar{\lambda}\bar{z}.\,\bar{z})
\end{aligned}
$$

Present-stage value translation $(\!|v^0|\!)$

$$
\begin{aligned}
(\!|\lambda x.e|\!) &= \lambda x.\lambda k.\,[\![e]\!]_0\,\bar{@}\uparrow k\\
(\!|\mathsf{fix}|\!) &= \lambda y.\,G(\mathsf{fix}\,G)y \qquad \text{where } G=\lambda F.\lambda y.\lambda k.k(\lambda x.\lambda k'.Fy\lambda f.yf\lambda z.zxk')\\
(\!|(v_1^0,v_2^0)|\!) &= ((\!|v_1^0|\!),(\!|v_2^0|\!))\\
(\!|p|\!) &= \lambda x.\lambda k.k(px) \qquad \text{where } p=\mathsf{fst},\mathsf{snd}\\
(\!|\mathrm{出}|\!) &= \lambda f.\lambda k.\,f(\lambda x.\lambda k'.(\lambda x'.k'x')(kx))(\lambda x.x)\\
(\!|v^0|\!) &= v^0 \qquad \text{for all other values } v^0
\end{aligned}
$$

Future-stage term translation $[\![e]\!]_1$

$$
\begin{aligned}
[\![v^1]\!]_1 &= \bar{\lambda}\bar{k}.\,\bar{k}\,\bar{@}\,v^1\\
[\![e\oplus e']\!]_1 &= \bar{\lambda}\bar{k}.\,[\![e]\!]_1\,\bar{@}\,\bar{\lambda}\bar{x}.\,[\![e']\!]_1\,\bar{@}\,\bar{\lambda}\bar{y}.\,\bar{k}\,\bar{@}\,(\bar{x}\oplus\bar{y}) \quad \text{where } e\oplus e' \text{ is } e+e',\,ee',\,\text{or } (e,e')\\
[\![\mathsf{if}\,e\,\mathsf{then}\,e_1\,\mathsf{else}\,e_2]\!]_1 &= \bar{\lambda}\bar{k}.\,[\![e]\!]_1\,\bar{@}\,\bar{\lambda}\bar{x}.\,[\![e_1]\!]_1\,\bar{@}\,\bar{\lambda}\bar{y}.\,[\![e_2]\!]_1\,\bar{@}\,\bar{\lambda}\bar{z}.\,\bar{k}\,\bar{@}\,\mathsf{if}\,\bar{x}\,\mathsf{then}\,\bar{y}\,\mathsf{else}\,\bar{z}\\
[\![\sim e]\!]_1 &= \bar{\lambda}\bar{k}.\,[\![e]\!]_0\,\bar{@}\,\bar{\lambda}\bar{x}.\,\bar{k}\,\bar{@}\sim\bar{x}\\
[\![\{e\}]\!]_1 &= \bar{\lambda}\bar{k}.\,[\![e]\!]_1\,\bar{@}\,\bar{\lambda}\bar{x}.\,\bar{k}\,\bar{@}\,\{\bar{x}\}\\
[\![\lambda x.e]\!]_1 &= \bar{\lambda}\bar{k}.\,(\lambda z.\bar{k}\,\bar{@}\sim z)\langle\lambda x.\sim([\![e]\!]_1\,\bar{@}\,\bar{\lambda}\bar{z}.\langle\bar{z}\rangle)\rangle \quad \text{unless } e \text{ is a (future-stage) value,}\\
&\qquad\qquad \text{in which case the case } [\![v^1]\!]_1 \text{ above applies}
\end{aligned}
$$

Reflection	Reification
$\uparrow k=\bar{\lambda}\bar{x}.\,k\bar{x}$	$\downarrow\bar{k}=\lambda x.\,\bar{k}\,\bar{@}\,x$

Fig. 8. One-pass CPS translation for $\lambda_1^{\oslash}$.

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

$$[]^* = [] \qquad (\Gamma, x:\tau)^* = \Gamma^*,\, x:\tau^* \qquad (\Gamma,\langle x:\upsilon\rangle)^* = \Gamma^*,\langle x:\upsilon\rangle \tag{25}$$

Proposition 12 (translation preserves typing) We have the following three properties.

(1) If $\Gamma \vdash v^0 : \tau\ ;\ \tau_0$, then $\Gamma^* \vdash (\!|v^0|\!) : \tau^*$.
(2) Suppose $\Gamma \vdash e : \tau\ ;\ \tau_0$. Let Φ be a type environment in the target language and $\bar{k}$ be a meta-level function such that, for any type environment Δ and term t of the target language, if $\Gamma^*, \Delta \vdash t : \tau^*$, then $\Gamma^*, \Delta, \Phi \vdash \bar{k}\,\bar{@}\,t : \tau_0^*$. Then $\Gamma^*, \Phi \vdash [\![e]\!]_0\,\bar{@}\,\bar{k} : \tau_0^*$.

(3) Suppose $\Gamma \vdash e : \upsilon \ ; \tau_0 \ ; \upsilon_0$. Let Φ be a type environment in the target language and $\bar{k}$ be a meta-level function such that, for any type environment Δ and term t of the target language, if $\Gamma^*, \Delta \vdash t : \upsilon \ ;; \upsilon_0$ then $\Gamma^*, \Delta, \Phi \vdash \bar{k} \mathbin{\bar{@}} t : \tau_0^*$. Then $\Gamma^*, \Phi \vdash [\![e]\!]_1 \mathbin{\bar{@}} \bar{k} : \tau_0^*$.

If we ignore the difference of object language and metalanguage, the proposition can be stated in a much simpler way. For instance, part 2 is stated as: if $\Gamma \vdash e : \tau \ ; \tau_0$, then $\Gamma^* \vdash [\![e]\!]_0 : (\tau^* \to \tau_0^*) \to \tau_0^*$. In our one-pass CPS translation, $\bar{k}$ represents a meta-level function that (roughly) has type $\tau^* \to \tau_0^*$, and the assumption on $\bar{k}$ in the proposition is a precise statement for this intuition.

Proof The definitions in Figure 8 are mutually inductive, and $[\![v^0]\!]_0$ is defined in terms of $(\!|v^0|\!)$. Accordingly, we proceed by mutual induction on typing derivations, allowing part 2 to appeal to part 1 in the case where $e = v^0$.

For part 1, the interesting case is when $v^0 = \lambda x.e$. By inversion, let $\Gamma, x : \tau' \vdash e : \tau'' \ ; \tau_1$. Recall from Figure 8 that the translation in this case binds the dynamic continuation variable k. Note that $\uparrow k$ is a meta-level function such that $\Gamma^*, x : \tau'^*, \Delta, k : \tau''^* \to \tau_1^* \vdash \uparrow k \mathbin{\bar{@}} t : \tau_1^*$ whenever $\Gamma^*, x : \tau'^*, \Delta \vdash t : \tau''^*$. By the induction hypothesis (choosing $\Phi = k : \tau''^* \to \tau_1^*$), we have $\Gamma^*, x : \tau'^*, k : \tau''^* \to \tau_1^* \vdash [\![e]\!]_0 \mathbin{\bar{@}} \uparrow k : \tau_1^*$. Hence, $\Gamma^* \vdash \lambda x.\, \lambda k.\, [\![e]\!]_0 \mathbin{\bar{@}} \uparrow k : (\tau' \to \tau''/\tau_1)^*$, in which $\lambda x.\, \lambda k.\, [\![e]\!]_0 \mathbin{\bar{@}} \uparrow k$ is $(\!|\lambda x.e|\!)$ by definition, as desired.

For part 2, the interesting cases are as follows:

Case $e = v^0$: By the induction hypothesis for part 1, we have $\Gamma^* \vdash (\!|v^0|\!) : \tau^*$. Thus, by the assumption on $\bar{k}$, we have $\Gamma^*, \Phi \vdash \bar{k} \mathbin{\bar{@}} (\!|v^0|\!) : \tau_0^*$, in which $\bar{k} \mathbin{\bar{@}} (\!|v^0|\!)$ is $[\![v^0]\!]_0 \mathbin{\bar{@}} \bar{k}$ by definition, as desired.

Case $e = e_1 e_2$: By inversion, let $\Gamma \vdash e_1 : \tau' \to \tau/\tau_0 \ ; \tau_0$ and $\Gamma \vdash e_2 : \tau' \ ; \tau_0$ and $\bar{k}$ be a meta-level function such that $\Gamma^*, \Delta, \Phi \vdash \bar{k} \mathbin{\bar{@}} t : \tau_0^*$ whenever $\Gamma^*, \Delta \vdash t : \tau^*$. By the definition of $[\![e_1 e_2]\!]_0$ and the induction hypothesis for e_1 (choosing the same Φ), we need only show

$$\Gamma^*, \Delta_1, \Phi \vdash [\![e_2]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{x}.\, t_1 \bar{x} \downarrow \bar{k} : \tau_0^* \tag{26}$$

given $\Gamma^*, \Delta_1 \vdash t_1 : (\tau' \to \tau/\tau_0)^*$. By the induction hypothesis for e_2 (choosing Φ to be Δ_1, Φ), we need only show

$$\Gamma^*, \Delta_2, \Delta_1, \Phi \vdash t_1 t_2 \downarrow \bar{k} : \tau_0^* \tag{27}$$

given $\Gamma^*, \Delta_2 \vdash t_2 : \tau'^*$. Finally, given that $\downarrow \bar{k}$ is defined to be $\lambda x.\, \bar{k} \mathbin{\bar{@}} x$, we choose Δ to be $x : \tau^*, \Delta_2, \Delta_1$ and t to be x.

Case $e = \langle e' \rangle$: By inversion, let $\Gamma \vdash e' : \upsilon \ ; \tau_0 \ ; \upsilon_0$ and $\bar{k}$ be a meta-level function such that $\Gamma^*, \Delta, \Phi \vdash \bar{k} \mathbin{\bar{@}} t : \tau_0^*$ whenever $\Gamma^*, \Delta \vdash t : \langle \upsilon / \upsilon_0 \rangle$. Then $\Gamma^*, \Delta, \Phi \vdash \bar{k} \mathbin{\bar{@}} \langle t \rangle : \tau_0^*$ whenever $\Gamma^*, \Delta \vdash t : \upsilon \ ;; \upsilon_0$. By the induction hypothesis for part 3 (choosing the same Φ), we have $\Gamma^*, \Phi \vdash [\![e']\!]_1 \mathbin{\bar{@}} \bar{\lambda}\bar{x}.\, \bar{k} \mathbin{\bar{@}} \langle \bar{x} \rangle : \tau_0^*$, in which $[\![e']\!]_1 \mathbin{\bar{@}} \bar{\lambda}\bar{x}.\, \bar{k} \mathbin{\bar{@}} \langle \bar{x} \rangle$ is $[\![\langle e' \rangle]\!]_0 \mathbin{\bar{@}} \bar{k}$ by definition, as desired.

For part 3, the interesting cases are as follows:

Case $e = x$: By inversion, let $\Gamma \vdash x : \upsilon \ ; \tau_0 \ ; \upsilon_0$ and $\bar{k}$ be a meta-level function such that $\Gamma^*, \Delta, \Phi \vdash \bar{k} \mathbin{\bar{@}} t : \tau_0^*$ whenever $\Gamma^*, \Delta \vdash t : \upsilon \ ;; \upsilon_0$. Then $(\langle x : \upsilon \rangle) \in \Gamma$, so $(\langle x : \upsilon \rangle) \in \Gamma^*$ and so $\Gamma^* \vdash x : \upsilon \ ;; \upsilon_0$. We thus choose Δ to be $[]$ and t to be x and conclude $\Gamma^*, \Phi \vdash \bar{k} \mathbin{\bar{@}} x : \tau_0^*$, in which $\bar{k} \mathbin{\bar{@}} x$ is $[\![x]\!]_1 \mathbin{\bar{@}} \bar{k}$ by definition, as desired.

Case $e = \lambda x.e'$ where e' is not a value: By inversion, let $\Gamma, \langle x : \upsilon\rangle \vdash e' : \upsilon' \;;\; \langle \upsilon'/\upsilon_1\rangle \;;\; \upsilon_1$ and $\bar{k}$ be a meta-level function such that $\Gamma^*, \Delta, \Phi \vdash \bar{k}\ \bar{@}\ t : \tau_0^*$ whenever $\Gamma^*, \Delta \vdash t : \upsilon \to \upsilon'/\upsilon_1 \;;;\; \upsilon_0$. By the induction hypothesis (choosing the same Φ), because $\Gamma^*, \langle x : \upsilon\rangle, \Delta, \Phi \vdash \langle t\rangle : \langle \upsilon'/\upsilon_1\rangle$ whenever $\Gamma^*, \langle x : \upsilon\rangle, \Delta \vdash t : \upsilon' \;;;\; \upsilon_1$, we have $\Gamma^*, \langle x : \upsilon\rangle, \Phi \vdash [\![e']\!]_1\ \bar{@}\ \bar{\lambda}\bar{z}.\langle \bar{z}\rangle : \langle \upsilon'/\upsilon_1\rangle$. So, $\Gamma^*, \Phi \vdash \langle \lambda x. \sim([\![e']\!]_1\ \bar{@}\ \bar{\lambda}\bar{z}.\langle \bar{z}\rangle)\rangle : \langle (\upsilon \to \upsilon'/\upsilon_1)/\upsilon_0\rangle$. Finally, choose $\Delta = z : \langle (\upsilon \to \upsilon'/\upsilon_1)/\upsilon_0\rangle$ and $t = \sim z$ to show that $\Gamma^*, \Phi \vdash \lambda z.\bar{k}\ \bar{@}\ \sim z : \langle (\upsilon \to \upsilon'/\upsilon_1)/\upsilon_0\rangle \to \tau_0^*$.

Case $e = \{e'\}$: By inversion, let $\Gamma \vdash e' : \upsilon \;;\; \tau_0 \;;\; \upsilon$ and $\bar{k}$ be a meta-level function such that $\Gamma^*, \Delta, \Phi \vdash \bar{k}\ \bar{@}\ t : \tau_0^*$ whenever $\Gamma^*, \Delta \vdash t : \upsilon \;;;\; \upsilon_0$. Then $\Gamma^*, \Delta, \Phi \vdash \bar{k}\ \bar{@}\ \{t\} : \tau_0^*$ whenever $\Gamma^*, \Delta \vdash t : \upsilon \;;;\; \upsilon$. By the induction hypothesis (choosing the same Φ), we have $\Gamma^*, \Phi \vdash [\![e']\!]_1\ \bar{@}\ \bar{\lambda}\bar{x}.\bar{k}\ \bar{@}\ \{\bar{x}\} : \tau_0^*$, in which $[\![e']\!]_1\ \bar{@}\ \bar{\lambda}\bar{x}.\bar{k}\ \bar{@}\ \{\bar{x}\}$ is $[\![e]\!]_1\ \bar{@}\ \bar{k}$ by definition, as desired. □

We move on to show that our translation also preserves the dynamic semantics of $\lambda_1^{\mathcal{O}}$. To do so, we need a crucial lemma that relates the translations of a delimited context D^{ij} and of it plugged with a term e. Intuitively, this lemma says that the translation of $D^{ij}[e]$ puts e "in control" by applying the translation of e to the translation of $D^{ij}[x]$. After all, D^{ij} is a delimited *evaluation* context. The subsequent corollary then extends the lemma from delimited contexts to all contexts C^i.

Lemma 13 Let $\bar{k}$ be a meta-function from level-i target values to level-0 target terms, and let x be a fresh object variable at level j (so $(\!|x|\!) = x$ in case $j = 0$). Then

$$[\![D^{ij}[e]]\!]_i\ \bar{@}\ \bar{k} = [\![e]\!]_j\ \bar{@}\ \bar{\lambda}\bar{z}.([\![D^{ij}[x]]\!]_i\ \bar{@}\ \bar{k})[x := \bar{z}]. \tag{28}$$

(That is, the target terms on the two sides are related by canceling $\langle\ \rangle$ against $\sim$ as described at the beginning of this section.)

Proof By induction and case analysis on D^{ij}.

Case $\Box$: By definition of $[\![x]\!]_i$, then β and η reductions at the metalanguage level.

Case $D^{i1}[\sim\Box]$: We have

$$\begin{aligned}
\text{LHS} &= [\![D^{i1}[\sim e]]\!]_i\ \bar{@}\ \bar{k} \\
&= [\![\sim e]\!]_1\ \bar{@}\ \bar{\lambda}\bar{z}.([\![D^{i1}[x]]\!]_i\ \bar{@}\ \bar{k})[x := \bar{z}] && \text{by induction hypothesis} \\
&= [\![e]\!]_0\ \bar{@}\ \bar{\lambda}\bar{z}.([\![D^{i1}[x]]\!]_i\ \bar{@}\ \bar{k})[x := \sim\bar{z}] && \text{by definition of } [\![\sim e]\!]_1,
\end{aligned} \tag{29}$$

so

$$\begin{aligned}
\text{RHS} &= [\![e]\!]_0\ \bar{@}\ \bar{\lambda}\bar{z}.([\![D^{i1}[\sim x]]\!]_i\ \bar{@}\ \bar{k})[x := \bar{z}] \\
&= [\![e]\!]_0\ \bar{@}\ \bar{\lambda}\bar{z}.([\![D^{i1}[x]]\!]_i\ \bar{@}\ \bar{k})[x := \sim\bar{z}] && \text{by Equation (29) with } e = x, \\
&&& \quad \text{and definition of } [\![x]\!]_0 \\
&= \text{LHS} && \text{by Equation (29).}
\end{aligned} \tag{30}$$

Cases $D^{i0}[\langle\Box\rangle]$, $D^{i1}[\{\Box\}]$ and $D^{i0}[\mathsf{ifz}\ \Box\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]$ are similar.

Case $D^{i0}[v^0\Box]$: We have

$$\begin{aligned}
\text{LHS} &= [\![D^{i0}[v^0 e]]\!]_i\ \bar{@}\ \bar{k} \\
&= [\![v^0 e]\!]_0\ \bar{@}\ \bar{\lambda}\bar{z}.([\![D^{i0}[x]]\!]_i\ \bar{@}\ \bar{k})[x := \bar{z}] && \text{by induction hypothesis} \\
&= [\![e]\!]_0\ \bar{@}\ \bar{\lambda}\bar{z}.(\!|v^0|\!)\bar{z}(\lambda x.[\![D^{i0}[x]]\!]_i\ \bar{@}\ \bar{k}) && \text{by def. of } [\![v^0 e]\!]_0 \text{ and } [\![v^0]\!]_0,
\end{aligned} \tag{31}$$

so

$$\begin{aligned}\text{RHS} &= [\![e]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,([\![D^{i0}[v^0 x]]\!]_i \mathbin{\bar{@}} \bar{k})[x := \bar{z}] \\ &= [\![e]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,(\!|v^0|\!)\bar{z}(\lambda x.\,[\![D^{i0}[x]]\!]_i \mathbin{\bar{@}} \bar{k}) && \text{by Equation (31) with } e = x, \text{ and definition of } [\![x]\!]_0 \\ &= \text{LHS} && \text{by Equation (31).}\end{aligned} \tag{32}$$

Cases $D^{i0}[v^0 + \Box]$, $D^{i0}[(v^0, \Box)]$, $D^{i1}[v^1 + \Box]$, $D^{i1}[v^1 \Box]$, $D^{i1}[(v^1, \Box)]$, and $D^{i1}[\mathsf{ifz}\ v^1\ \mathsf{then}\ v_1^1\ \mathsf{else}\ \Box]$ are similar.

Case $D^{i0}[\Box e_2]$: We have

$$\begin{aligned}\text{LHS} &= [\![D^{i0}[e e_2]]\!]_i \mathbin{\bar{@}} \bar{k} \\ &= [\![e e_2]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,([\![D^{i0}[x]]\!]_i \mathbin{\bar{@}} \bar{k})[x := \bar{z}] && \text{by induction hypothesis} \\ &= [\![e]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{y}.\,[\![e_2]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{y}\bar{z}(\lambda x.\,[\![D^{i0}[x]]\!]_i \mathbin{\bar{@}} \bar{k}) && \text{by definition of } [\![e e_2]\!]_0 \\ &= [\![e]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{y}.\,([\![D^{i0}[f e_2]]\!]_i \mathbin{\bar{@}} \bar{k})[f := \bar{y}] && \text{by Equation (31) with } v^0 = f \\ &= \text{RHS}.\end{aligned} \tag{33}$$

Cases $D^{i0}[\Box + e_2]$, $D^{i0}[(\Box, e_2)]$, $D^{i1}[\Box + e_2]$, $D^{i1}[\Box e_2]$, $D^{i1}[(\Box, e_2)]$, $D^{i1}[\mathsf{ifz}\ v^1\ \mathsf{then}\ \Box\ \mathsf{else}\ e_2]$, and $D^{i1}[\mathsf{ifz}\ \Box\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]$ are similar. □

Corollary 14 Let x be an object variable at level 0 that does not occur in expressions $C^0[\{e\}]$ and $C^1[\lambda y.\,e]$ mentioned below.

If C^0 is a context and e is a present-stage expression, then

$$[\![C^0[\{e\}]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z} = ([\![C^0[\{x\}]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z})[x := [\![e]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z}], \tag{34}$$

in which the target term $[\![C^0[\{x\}]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z}$ is equal to the result of plugging x into some evaluation context.

If C^1 is a context and e is a future-stage expression that is not a value, then

$$[\![C^1[\lambda y.\,e]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z} = ([\![C^1[\lambda y.\,{\sim}x]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z})[x := [\![e]\!]_1 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\langle\bar{z}\rangle], \tag{35}$$

in which the target term $[\![C^1[\lambda y.\,{\sim}x]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z}$ is equal to the result of plugging x into some evaluation context.

Proof By mutual induction on C^0 and C^1, using Lemma 13 at each step. □

Proposition 15 (reduction preservation) Let e and e' be present-stage expressions. If $\{e\} \rightsquigarrow \{e'\}$, then $[\![e]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z} \rightsquigarrow^* [\![e']\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z}$. (Moreover, $[\![e]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z} = [\![e']\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z}$ only if $e = C^1[{\sim}\langle v^1\rangle]$ and $e' = C^1[v^1]$.)

Proof By case analysis on $\rightsquigarrow$.

Case $+$: We have $\{C^0[n_1 + n_2]\} \rightsquigarrow \{C^0[n]\}$ where $n = n_1 \dot{+} n_2$, and we need to show $[\![C^0[n_1 + n_2]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z} \rightsquigarrow^+ [\![C^0[n]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z}$. We perform case analysis on C^0:

- If $C^0 = D^{00}$, then what we need is $[\![D^{00}[n_1 + n_2]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z} \rightsquigarrow^+ [\![D^{00}[n_1 \dot{+} n_2]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z}$.
- If $C^0 = C^{0\prime}[\{D^{00}\}]$, then the first half of Corollary 14 gives us an evaluation context C in the target language (namely, $C[x] = [\![C^{0\prime}[\{x\}]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z}$) such that what we need is $C[[\![D^{00}[n_1 + n_2]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z}] \rightsquigarrow^+ C[[\![D^{00}[n_1 \dot{+} n_2]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z}]$.

So we just need to show $[\![D^{00}[n_1 + n_2]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z} \rightsquigarrow^+ [\![D^{00}[n_1 \dot{+} n_2]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z}$ as in the previous case.

- If $C^0 = C^1[\lambda x.\, D^{10}]$, then the second half of Corollary 14 gives us an evaluation context C in the target language (namely, $C[x] = [\![C^1[\lambda y. \sim x]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z}$) such that what we need is $C[[\![D^{10}[n_1 + n_2]]\!]_1 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\langle\bar{z}\rangle] \rightsquigarrow^+ C[[\![D^{10}[n_1 \dot{+} n_2]]\!]_1 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\langle\bar{z}\rangle]$. So we just need to show $[\![D^{10}[n_1 + n_2]]\!]_1 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\langle\bar{z}\rangle \rightsquigarrow^+ [\![D^{10}[n_1 \dot{+} n_2]]\!]_1 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\langle\bar{z}\rangle$.

To sum up this case analysis, we just need to show

$$[\![D^{i0}[n_1 + n_2]]\!]_i \mathbin{\bar{@}} \bar{k}' \rightsquigarrow^+ [\![D^{i0}[n_1 \dot{+} n_2]]\!]_i \mathbin{\bar{@}} \bar{k}' \tag{36}$$

where either $i = 0$ and $\bar{k}' = \bar{\lambda}\bar{z}.\,\bar{z}$ or $i = 1$ and $\bar{k}' = \bar{\lambda}\bar{z}.\,\langle\bar{z}\rangle$. We apply Lemma 13 to both sides, letting $\bar{k} = \bar{\lambda}\bar{z}.\,([\![D^{i0}[x]]\!]_i \mathbin{\bar{@}} \bar{k}')[x := \bar{z}]$:

$$[\![n_1 + n_2]\!]_0 \mathbin{\bar{@}} \bar{k} = {\downarrow}\bar{k}(n_1 + n_2) \rightsquigarrow_+ {\downarrow}\bar{k}(n_1 \dot{+} n_2) \rightsquigarrow_{\beta_v} \bar{k} \mathbin{\bar{@}} (n_1 \dot{+} n_2) = [\![n_1 \dot{+} n_2]\!]_0 \mathbin{\bar{@}} \bar{k}. \tag{37}$$

The fix, fst, and snd cases are similar.

Case β_v: Again by Corollary 14 and Lemma 13, we just need to show

$$[\![(\lambda x.\, e)\, v^0]\!]_0 \mathbin{\bar{@}} \bar{k} \rightsquigarrow^* [\![e[x := v^0]]\!]_0 \mathbin{\bar{@}} \bar{k}. \tag{38}$$

We have

$$[\![(\lambda x.\, e)\, v^0]\!]_0 \mathbin{\bar{@}} \bar{k} = (\lambda x.\, \lambda k.\, [\![e]\!]_0 \mathbin{\bar{@}} {\uparrow}k)(\!|v^0|\!){\downarrow}\bar{k} \rightsquigarrow^2_{\beta_v} ([\![e]\!]_0 \mathbin{\bar{@}} {\uparrow}{\downarrow}\bar{k})[x := (\!|v^0|\!)]. \tag{39}$$

The last term reduces to $([\![e]\!]_0 \mathbin{\bar{@}} \bar{k})[x := (\!|v^0|\!)]$ using the reduction (20) zero or more times, because our translation always applies the static continuation to a value, and the static continuation always uses its argument at most once. The two ifz cases use the same reasoning to reduce $[\![e_1]\!]_0 \mathbin{\bar{@}} {\uparrow}{\downarrow}\bar{k}$ and $[\![e_2]\!]_0 \mathbin{\bar{@}} {\uparrow}{\downarrow}\bar{k}$ to $[\![e_1]\!]_0 \mathbin{\bar{@}} \bar{k}$ and $[\![e_2]\!]_0 \mathbin{\bar{@}} \bar{k}$.

Case $\{\}$: Again we use Corollary 14 and Lemma 13:

$$[\![\{v^0\}]\!]_0 \mathbin{\bar{@}} \bar{k} = {\downarrow}\bar{k}(\!|v^0|\!) \rightsquigarrow_{\beta_v} \bar{k} \mathbin{\bar{@}} (\!|v^0|\!) = [\![v^0]\!]_0 \mathbin{\bar{@}} \bar{k}. \tag{40}$$

Case $\sim$: Again we use Corollary 14 and Lemma 13 (letting j be 1 rather than 0 this time):

$$[\![\sim\langle v^1\rangle]\!]_1 \mathbin{\bar{@}} \bar{k} = \bar{k} \mathbin{\bar{@}} \sim\langle v^1\rangle = \bar{k} \mathbin{\bar{@}} v^1 = [\![v^1]\!] \mathbin{\bar{@}} \bar{k}. \tag{41}$$

If the source reduction turns a future-stage λ-abstraction into a value, then one β_v-reduction per future-stage λ-abstraction is needed to match the CPS translation of the value.

Case 出^0: Letting $\bar{k} = \bar{\lambda}\bar{z}.\,([\![D^{00}[x]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z})[x := \bar{z}]$, we compute

$$\begin{aligned}
[\![D^{00}[\text{出}v^0]]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z} = (\!|\text{出}|\!)(\!|v^0|\!){\downarrow}\bar{k} &\rightsquigarrow^2_{\beta_v} (\!|v^0|\!)(\lambda x.\, \lambda k'.\, (\lambda x'.\, k'x')((\downarrow\bar{k})x))(\lambda x.\, x) \\
&\rightsquigarrow (\!|v^0|\!)(\lambda x.\, \lambda k'.\, (\lambda x'.\, k'x')(\bar{k} \mathbin{\bar{@}} x))(\lambda x.\, x) \\
&= [\![v^0(\lambda x.\, \{D^{00}[x]\})]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z}.
\end{aligned} \tag{42}$$

Case 出^1: Letting $\bar{k} = \bar{\lambda}\bar{z}.\,([\![\langle D^{10}[x]\rangle]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z})[x := \bar{z}]$, we compute

$$\begin{aligned}
[\![D^{10}[\text{出}v^0]]\!]_1 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\langle\bar{z}\rangle = (\!|\text{出}|\!)(\!|v^0|\!){\downarrow}\bar{k} &\rightsquigarrow^2_{\beta_v} (\!|v^0|\!)(\lambda x.\, \lambda k'.\, (\lambda x'.\, k'x')((\downarrow\bar{k})x))(\lambda x.\, x) \\
&\rightsquigarrow (\!|v^0|\!)(\lambda x.\, \lambda k'.\, (\lambda x'.\, k'x')(\bar{k} \mathbin{\bar{@}} x))(\lambda x.\, x) \\
&= [\![v^0(\lambda x.\, \{\langle D^{10}[x]\rangle\})]\!]_0 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\bar{z} \\
&= [\![\sim(v^0(\lambda x.\, \{\langle D^{10}[x]\rangle\}))]\!]_1 \mathbin{\bar{@}} \bar{\lambda}\bar{z}.\,\langle\bar{z}\rangle.
\end{aligned} \tag{43}$$

The last equality is why we equate $\langle\sim e\rangle$ with e in the target language. □

Values	$v^i ::= n \mid \mathsf{fix} \mid (v^i, v^i) \mid \mathsf{fst} \mid \mathsf{snd} \mid 出 \mid \langle v^{i+1} \rangle \mid x$	
	$v^0 \mathrel{+}= \lambda x.e$	
	$v^i \mathrel{+}= v^i + v^i \mid \lambda x.v^i \mid v^i v^i \mid \mathsf{ifz}\ v^i\ \mathsf{then}\ v^i\ \mathsf{else}\ v^i \mid \{v^i\}$	when $i \geq 1$
	$v^i \mathrel{+}= {\sim} v^{i-1}$	when $i \geq 2$
Frames	$F^i ::= \Box + e \mid v^i + \Box \mid \Box e \mid v^i \Box \mid (\Box, e) \mid (v^i, \Box) \mid \mathsf{ifz}\ \Box\ \mathsf{then}\ e\ \mathsf{else}\ e$	
	$F^i \mathrel{+}= \mathsf{ifz}\ v^i\ \mathsf{then}\ \Box\ \mathsf{else}\ e \mid \mathsf{ifz}\ v^i\ \mathsf{then}\ v^i\ \mathsf{else}\ \Box \mid \{\Box\}$	when $i \geq 1$
Delimited contexts	$D^{ij} ::= D^{ij}[F^j] \mid D^{i(j+1)}[{\sim}\Box]$	
	$D^{ii} \mathrel{+}= \Box$	
	$D^{ij} \mathrel{+}= D^{i(j-1)}[\langle \Box \rangle]$	when $j \geq 1$
Contexts	$C^j ::= D^{0j} \mid C^0[\{D^{0j}\}] \mid C^i[\lambda x.D^{ij}]$	when $i \geq 1$

Fig. 9. Values and contexts of $\lambda^{\oslash}$. We write $+\!=$ to add alternatives to a preceding BNF rule.

$$C^i\left[\lambda x.D^{i0}[出 v^0]\right] \rightsquigarrow C^i\left[\lambda x.{\sim}^i(v^0(\lambda y.\{\langle D'^0[y]\rangle^i\}))\right] \qquad \text{where } y \text{ is fresh} \qquad (出^i)$$

Fig. 10. Operational semantics: small-step reduction $e \rightsquigarrow e'$ particular to $\lambda^{\oslash}$. Other reduction rules are the same as those in $\lambda_1^{\oslash}$. Here $\langle\ \rangle^i$ and ${\sim}^i$ stand for i levels of brackets and escapes; $i \geqslant 1$.

7 Multilevel calculus with control effects

This section generalizes $\lambda_1^{\oslash}$ to the full language $\lambda^{\oslash}$ with an arbitrary number of levels, removing the restriction on nesting of brackets and escapes and making the language, its type system in particular, more uniform. The generalization only affects the number of levels for expressions; the syntax of expressions remains the same, see Section 3 and Figure 1. Level 0 still refers to the present stage, at which reductions may occur. The language $\lambda^{\oslash}$ permits more than one future-stage level by allowing brackets and escapes to nest. Brackets increase the level of the expression and escape decreases it. The language $\lambda^{\oslash}$ thus can express not only code generators but also generators of code generators, and so on. The level superscripts are now arbitrary natural numbers. Figure 9 defines values and contexts of $\lambda^{\oslash}$, generalizing those of $\lambda_1^{\oslash}$ to multiple future levels.

For example, according to the definition of delimited contexts in Figure 9, $\Box$ is a D^{22}, so ${\sim}\Box$ is a D^{21} and so ${\sim}{\sim}\Box$ is a D^{20}. Similarly, because $\Box$ is a D^{00} as well as a D^{11}, we have that $\langle\Box\rangle$ is a D^{01} as well as a D^{12}. Putting these derivations together, the parse tree below shows that $\langle\lambda x.\langle\lambda y.{\sim}{\sim}\Box\rangle\rangle$ is an evaluation context C^0:

$$\dfrac{\dfrac{\dfrac{\overline{D^{00} ::= \Box}}{D^{01} ::= \langle\Box\rangle} \quad \dfrac{\overline{D^{11} ::= \Box}}{D^{12} ::= \langle\Box\rangle}}{C^1 ::= \langle\Box\rangle} \quad \dfrac{\dfrac{\overline{D^{22} ::= \Box}}{D^{21} ::= {\sim}\Box}}{D^{20} ::= {\sim}{\sim}\Box}}{C^2 ::= \langle\lambda x.\langle\Box\rangle\rangle} \qquad (44)$$
$$\overline{C^0 ::= \langle\lambda x.\langle\lambda y.{\sim}{\sim}\Box\rangle\rangle}$$

The operational semantics of $\lambda^{\oslash}$ is, with one exception, identical to that of $\lambda_1^{\oslash}$. After all, in both calculi, reductions are only performed at level 0. The sole exception is to generalize the $出^1$ reduction of $\lambda_1^{\oslash}$ to the $出^i$ reduction as shown in Figure 10.

Type variables	α, β
Types	$\tau ::= \mathsf{int} \mid \tau \to \tau'/\tau_0 \mid \langle \tau/\tau_0 \rangle \mid (\tau, \tau') \mid \alpha$
Answer-type sequences	$T_i ::= \tau_0, \ldots, \tau_i$
Judgments	$\Gamma \vdash e : \tau \,;\, T_i$
Environments	$\Gamma ::= [] \mid \Gamma, \langle x : \tau \rangle^i$

$$\frac{\Gamma, \langle x:\tau\rangle^i \vdash e : \tau' \,;\, \langle \tau'/\tau_i' \rangle^{\langle i \rangle}, \langle \tau'/\tau_i' \rangle^{\langle i-1 \rangle}, \ldots, \langle \tau'/\tau_i' \rangle^{\langle 1 \rangle}, \tau_i'}{\Gamma \vdash (\lambda x.e) : \tau \to \tau'/\tau_i' \,;\, T_i} \qquad \frac{\Gamma \vdash e : \tau_i \,;\, T_{i-1}, \tau_i}{\Gamma \vdash \{e\} : \tau_i \,;\, T_{i-1}, \tau_i'}$$

$$\frac{\Gamma \vdash e : \tau \,;\, T_i, \tau_{i+1}}{\Gamma \vdash \langle e \rangle : \langle \tau/\tau_{i+1} \rangle \,;\, T_i} \qquad \frac{\Gamma \vdash e : \langle \tau/\tau_{i+1} \rangle \,;\, T_i}{\Gamma \vdash {\sim}e : \tau \,;\, T_i, \tau_{i+1}}$$

Fig. 11. The type system of $\lambda^{\oslash}$ and selected typing rules. The notation $\tau^{\langle n \rangle}$ is explained in the text.

The type system of $\lambda^{\oslash}$ is essentially the same as that of $\lambda_1^{\oslash}$; Figure 11 shows the crucial parts. Terms like $\langle\langle 42 \rangle\rangle$ with nested brackets are now allowed and inhabit types with nested brackets like $\langle\langle \mathsf{int} \,/\, \mathsf{int} \rangle/\, \mathsf{int} \rangle$. In type environments, the notation $\langle x : \tau \rangle^i$ (or just $x : \tau$ if $i = 0$) associates the level-i variable x with the type τ. We generalize answer types to sequences of types $T_i = \tau_0, \ldots, \tau_i$; each type τ_j in a sequence tracks the control effects that may occur at the level j. A typing judgment $\Gamma \vdash e : \tau \,;\, T_i$ for a level-i expression e includes the answer-type sequence T_i. The typing rules of $\lambda^{\oslash}$ simply generalize those of $\lambda_1^{\oslash}$; Figure 11 shows a sample. The type system of $\lambda^{\oslash}$ is more uniform however: There is only one form of judgment, which works for the present and the future stages. We no longer have to employ the judgment schema $\Gamma \vdash e : \tau \,;\, \tau_0 \; [; \upsilon_0]$; strictly speaking, $\lambda^{\oslash}$ has roughly half the number of typing rules compared to $\lambda_1^{\oslash}$.

The rule for future-stage abstraction is the only one with nontrivial generalization, giving more insight into our restriction of effects within the closest future-stage binder. (This restriction is similar to that in the region-based type-and-effect system (Talpin & Jouvelot, 1992).) Recall that, to prevent scope extrusion, we put an implicit present-stage reset under each future-stage binder. In the case of many future stages, we place many implicit resets, for levels 0 through $i-1$ (inclusive), under each level-i binder. For example, the body e of a level-1 abstraction is implicitly delimited as ${\sim}\{\langle e \rangle\}$; the body e of a level-2 abstraction is implicitly delimited as ${\sim}\{{\sim}\{\langle\langle e \rangle\rangle\}\}$. These implicit resets explain the answer-type sequence for the body of the level-i λ. In the first rule in Figure 11, the sequence is written $\langle \tau'/\tau_i' \rangle^{\langle i \rangle}, \langle \tau'/\tau_i' \rangle^{\langle i-1 \rangle}, \ldots, \langle \tau'/\tau_i' \rangle^{\langle 1 \rangle}, \tau_i'$ using the notation $\tau^{\langle i \rangle}$ inductively defined as follows:

$$\tau^{\langle 1 \rangle} = \tau, \qquad \tau^{\langle i+1 \rangle} = \langle \tau^{\langle i \rangle} / \tau^{\langle i \rangle} \rangle. \tag{45}$$

For example, at level 2, we can conclude $\Gamma \vdash (\lambda x.e) : \tau \to \tau'/\tau_2' \,;\, \tau_0, \tau_1, \tau_2$ provided that $\Gamma, \langle\langle x : \tau \rangle\rangle \vdash e : \tau' \,;\, \langle\langle \tau'/\tau_2' \rangle/\langle \tau'/\tau_2' \rangle\rangle, \langle \tau'/\tau_2' \rangle, \tau_2'$.

The formal properties of the multilevel calculus $\lambda^{\oslash}$ straightforwardly generalize those of $\lambda_1^{\oslash}$ stated in Sections 4.2 and 6 – and so do their proofs. To be certain, we have mechanized the type soundness proofs for $\lambda^{\oslash}$ in Twelf. Therefore, we omit the proofs, referring the reader for details to the well-commented code `circle-shift-n.elf` accompanying the paper.

8 Related work

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

8.1 Side effects in code 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. (Dussart and Thiemann (1996) in their Section 1.1 also criticized the approach of expanding source programs monadically.)

Danvy and Filinski's (1990, 1992) CPS translations and Bondorf's (1992) 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 they are but reformulated as combinators (Thiemann, 1999). It thus assures them sound; in particular, it is the first to accept Danvy and Filinski's (1990, 1992) and Lawall and Danvy's (1994) uses of delimited control.

In contrast, we are not aware of any sound type system for code generators that accepts Sumii and Kobayashi's (2001) specializer, which performs let-insertion using mutable state rather than delimited control to speed up specialization, or Dussart and Thiemann's (1996) specializer, which uses *first-class* mutable state (Morrisett, 1993) as well as first-class continuations. For both of those specializers, code generation is preceded by a sophisticated type-based binding-time analysis.

8.2 Domain-specific optimizations

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 & Kiselyov, 2011), and simplifying complex arithmetic for Fast Fourier Transform (Frigo & Johnson, 2005; Kiselyov & Taha, 2005). To better support these domain-specific optimizations, two approaches have been developed in the literature.

Side effects in specializer input. 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 the specializer to produce efficient code: by expressing unspecialized optimizations (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 (1999) 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 into *regions* (Talpin & Jouvelot, 1992; Tofte *et al*., 2004). Regions limit the references' lifetimes statically, much as our answer types do in the simulation of state by delimited control in Section 3.2. However, our simulation is relatively simplistic in that it allows access to only one mutable cell at a time, namely the cell at the nearest delimiter. To prevent scope extrusion, Thiemann and Dussart's (1999) binding-time analysis 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. In general, the use of regions to encapsulate monadic effects is a large and productive research area (Kagawa, 2001; Fluet & Morrisett, 2006; Ganz, 2006).

Optimizing compilers of imperative languages can determine the extent of possible mutations by control-flow analyses, but typically do not express the results of analysis in the language or make them available to the programmer for inspection or control, as our type system does. In particular, whereas Thiemann and Dussart's (1999) 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.

Custom code generators. 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 binding-time analysis) on a source program. Swadi *et al*.'s (2006) and Carette and Kiselyov's (2011) 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^{α}_{1v} 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 Section 6 here. (Choi *et al.* 2011 present another unstaging translation.) So far, then, it seems simpler to combine staging and effects by translating effects rather than staging away.

9 Conclusions

We have presented the first multilevel 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 and 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 that 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. As discussed in Section 3.4, we could then express loop-invariant code motion and generate assertions to be checked as early as possible. It might also help us to simultaneously access multiple pieces of state at different generated scopes, not just one piece as in Section 3.2. At the very least, we would be able to throw exceptions as we generate code, for example, when attempting to specialize Gibonacci (Section 2) to a negative *n*. (We can add an ad hoc extension to our system permitting effects to propagate beyond the closest binder provided the answer type is a base type.)

Another good way to enrich our language is to add delimited control to a richer language (like the language of Taha and Nielsen 2003) with `run` and cross-stage persistence. Finally, as discussed in Section 4.1, our language can be made much more comfortable to use by adding polymorphism over answer types (Asai & Kameyama, 2007).

Supplementary material

For supplementary material for this article, please visit `http://dx.doi.org/10.1017/S0956796811000256`

Acknowledgments

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

References

Asai, K. (2009) On typing delimited continuations: Three new solutions to the printf problem. *Higher-Order Symb. Comput.* **22**(3), 275–291.

Asai, K. & Kameyama, Y. (2007) Polymorphic delimited continuations. In *Proceedings of APLAS'07*, LNCS, vol. 4807, pp. 239–254.

Balat, V., Di Cosmo, R. & Fiore, M. P. (2004) Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *Proceedings of Annual Symposium on Principles of Programming Languages* (*POPL*), pp. 64–76.

Bondorf, A. (1992). Improving binding times without explicit CPS-conversion. In *Proceedings of LISP & Functional Programming*, pp. 1–10.

Bondorf, A. & Danvy, O. (1991) Automatic autoprojection of recursive equations with global variables and abstract data types. *Sci. Comput. Program.* **16**(2), 151–195.

Calcagno, C., Moggi, E. & Taha, W. (2000) Closed types as a simple approach to safe imperative multi-stage programming. In *Proceedings of ICALP*, LNCS, vol. 1853, pp. 25–36.

Calcagno, C., Moggi, E. & Taha, W. (2004) ML-like inference for classifiers. In *Proceedings of ESOP*, LNCS, vol. 2986, pp. 79–93.

Carette, J. (2006) Gaussian Elimination: A case study in efficient genericity with MetaOCaml. *Sci. Comput. Program.* **62**(1), 3–24 (special issue on the First MetaOCaml Workshop 2004).

Carette, J. & Kiselyov, O. (2011) Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Sci. Comput. Program.* **76**(5), 349–375.

Choi, W., Aktemur, B., Yi, K. & Tatsuta, M. (2011) Static analysis of multi-staged programs via unstaging translation. In *Proceedings of POPL '11: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, Ball, T. & Sagiv, M. (eds). New York: ACM Press, pp. 81–92.

Cohen, A., Donadio, S., Garzarán, M. J., Herrmann, C. A., Kiselyov, O. & Padua, D. A. (2006) In search of a program generator to implement generic transformations for high-performance computing. *Sci. Comput. Program.* **62**(1), 25–46.

Czarnecki, K., O'Donnell, J. T., Striegnitz, J. & Taha, W. (2004) DSL implementation in MetaOCaml, Template Haskell, and C++. In *Proceedings of DSPG 2003*, LNCS, vol. 3016, pp. 51–72.

Danvy, O. (1998) Functional unparsing. *J. Funct. Program.* **8**(6), 621–625.

Danvy, O. & Filinski, A. (1989) *A Functional Abstraction of Typed Contexts*. Tech. Rep. 89/12, DIKU, University of Copenhagen, Denmark. Available at: `http://www.daimi.au.dk/~danvy/Papers/fatc.ps.gz` Accessed 8 November 2011.

Danvy, O. & Filinski, A. (1990) Abstracting control. In *Proceedings of LISP & Functional Programming*, Nice, France, June 1990, pp. 151–160.

Danvy, O. & Filinski, A. (1992) Representing control: A study of the CPS transformation. *Math. Struct. Comput. Sci.* **2**(4), 361–391.

Davies, R. (1996) A temporal logic approach to binding-time analysis. In *Proceedings of LICS*, New Brunswick, New Jersey, July 27–30, pp. 184–195.

Davies, R. & Pfenning, F. (2001) A modal analysis of staged computation. *J. ACM* **48**(3), 555–604.

Dussart, D. & Thiemann, P. (1996). *Imperative Functional Specialization*. Tech. Rep. WSI-96-28. Universität Tübingen.

Eckhardt, J., Kaiabachev, R., Pašalić, E., Swadi, K. N. & Taha, W. (2005) Implicitly heterogeneous multi-stage programming. In *Proceedings of GPCE*, LNCS, vol. 3676, pp. 275–292.

Elliott, C. (2004) Programming graphics processors functionally. In *Proceedings of Haskell Workshop*, Snowbird, UT, USA, September 22, pp. 45–56.

Felleisen, M. (1991) On the expressive power of programming languages. *Sci. Comput. Program.* **17**(1–3), 35–75.

Felleisen, M., Friedman, D. P., Kohlbecker, E. E. & Duba, B. F.. (1986) Reasoning with continuations. In *Proceedings of the 1st Symposium on Logic in Computer Science*, Cambridge, MA, USA, June 16–18, pp. 131–141.

Filinski, A. (1994) Representing monads. In *Proceedings of POPL*, Portland, Oregon, USA, January 17–21, pp. 446–457.
Fluet, M. & Morrisett, J. G. (2006) Monadic regions. *J. Funct. Program.* **16**(4–5), 485–545.
Frigo, M. & Johnson, S. G. (2005) The design and implementation of FFTW3. *Proc. IEEE* **93**(2), 216–231.
Ganz, S. E. (2006). *Encapsulation of State with Monad Transformers*. Ph.D. thesis, Computer Science Department, Indiana University.
Gomard, C. K. & Jones, N. D. (1991) A partial evaluator for the untyped lambda calculus. *J. Funct. Program.* **1**(1), 21–69.
Hammond, K. & Michaelson, G. (2003) Hume: A domain-specific language for real-time embedded systems. In *Proceedings of GPCE*, LNCS, vol. 2830, pp. 37–56.
Igarashi, A. & Iwaki, M. (2007) Deriving compilers and virtual machines for a multi-level language. In *Proceedings of APLAS*, LNCS, vol. 4807, pp. 206–221.
Kagawa, K. (2001) Monadic encapsulation with stack of regions. In *Proceedings of FLOPS*, LNCS, vol. 2024, pp. 264–279.
Kameyama, Y., Kiselyov, O. & Shan, C.-c. (2008) Closing the stage: From staged code to typed closures. In *Proceedings of PEPM*, San Francisco, CA, USA, pp. 147–157.
Kameyama, Y., Kiselyov, O. & Shan, C.-c. (2009) Shifting the stage: Staging with delimited control. In *Proceedings of PEPM*. New York: ACM Press, pp. 111–120.
Kameyama, Y., Kiselyov, O. & Shan, C.-c. (2010) Mechanizing multilevel metatheory with control effects. In *Proceedings of 5th ACM SIGPLAN Workshop on Mechanizing Metatheory*. Available at: `http://www.cis.upenn.edu/~bcpierce/wmm/wmm10-program.html` Accessed 8 November 2011.
Kamin, S. (1996) Standard ML as a meta-programming language. Available at: `http://loome.cs.uiuc.edu/pubs.html` Accessed 8 November 2011.
Kiselyov, O. (2010) Delimited control in OCaml, abstractly and concretely: System description. In *Proceedings of FLOPS, LNCS*, vol. 6009, pp. 304–320. Extended version to appear in Theor. Comput. Sci.
Kiselyov, O., Shan, C.-c. & Sabry, A. (2006) Delimited dynamic binding. In *Proceedings of ICFP*, Portland, OR, USA, pp. 26–37.
Kiselyov, O. & Taha, W. (2005) Relating FFTW and split-radix. In *Proceedings of ICESS*, LNCS, vol. 3605, pp. 488–493.
Lawall, J. L. & Danvy, O. (1994) Continuation-based partial evaluation. In *Proceedings of LISP & Functional Programming*, Austin, TX, USA, August 5–8, pp. 227–238.
Lengauer, C. & Taha, W. (eds). (2006) Special issue on the 1st MetaOCaml workshop (2004), Sci. Comput. Program. **62**(1).
Leone, M. & Lee, P. (1998) Dynamic specialization in the Fabius system. *ACM Comput. Surv*, **30**(3es), article 23:1–23:6.
Leroy, X. & Pessaux, F. (2000) Type-based analysis of uncaught exceptions. *ACM Tran. Prog. Lang. Syst.* **22**(2), 340–377.
Masuko, M. & Asai, K. (2009) Direct implementation of shift and reset in the MinCaml compiler. In *Proceedings of ACM SIGPLAN Workshop on ML*. New York: ACM Press, pp. 49–60.
McAdam, B. J. (2001) Y in practical programs. *Proceedings of Workshop on Fixed Points in Computer Science*. Available at: `http://www.dsi.uniroma1.it/~labella/absMcAdam.ps` Accessed 8 November 2011.
MetaOCaml. (2006) MetaOCaml. Available at: `http://www.metaocaml.org` Accessed 8 November 2011.
Michie, D. (1968) "Memo" functions and machine learning. *Nature* 218: 19–22.
Minsky, Y. (2008) Bind without tears. Available at: `http://ocaml.janestreet.com/?q=node/23` Accessed 8 November 2011.
Moreau, L. (1998) A syntactic theory of dynamic binding. *Higher-Order Symb. Comput.* **11**(3), 233–279.

Morrisett, J. G. (1993) Refining first-class stores. In *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages*, pp. 73–87.

Nielson, F. & Nielson, H. R. (1988) Automatic binding time analysis for a typed λ-calculus. In *Proceedings of POPL*, San Diego, CA, USA, pp. 98–106.

Parigot, M. (1992) $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In *Proceedings of LPAR*, LNAI, vol. 624, pp. 190–201.

Pašalić, E., Taha, W. & Sheard, T. (2002) Tagless staged interpreters for typed languages. In *Proceedings of ICFP*, pp. 157–166.

Peyton Jones, S. L. (2003) The Haskell 98 language and libraries. *J. Funct. Program.* **13**(1), 1–255.

Püschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., Singer, B. W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R. W. & Rizzolo, N. (2005) SPIRAL: Code generation for DSP transforms. *Proc. IEEE* **93**(2), 232–275.

Sørensen, M. H. B., Glück, R. & Jones, N. D. (1994) Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In *Proceedings of ESOP*, LNCS, vol. 788, pp. 485–500.

Sumii, E. and Kobayashi, N. (2001) A hybrid approach to online and offline partial evaluation. *Higher-Order Symb. Comput.* **14**(2–3), 101–142.

Swadi, K., Taha, W. & Kiselyov, O. (2005) Dynamic programming benchmark. Available at: `http://www.metaocaml.org/examples/dp/` Accessed 8 November 2011.

Swadi, K., Taha, W., Kiselyov, O. & Pašalić, E. (2006) A monadic approach for avoiding code duplication when staging memoized functions. In *Proceedings of PEPM*, Charleston, SC, USA, January 9–10, pp. 160–169.

Taha, W. (2000). A sound reduction semantics for untyped CBN multi-stage computation. In *Proceedings of PEPM*, Boston, MA, USA, pp. 34–43.

Taha, W. (2005) Resource-aware programming. In *Proceedings of ICESS*, LNCS, vol. 3605, pp. 38–43.

Taha, W. & Nielsen, M. F. (2003) Environment classifiers. In *Proceedings of POPL*, New Orleans, LA, USA, January 15–17, pp. 26–37.

Talpin, J.-P. & Jouvelot, P. (1992) Polymorphic type, region and effect inference. *J. Funct. Program.* **2**(3), 245–271.

Thielecke, H. (2003) From control effects to typed continuation passing. In *Proceedings of POPL*, New Orleans, LA, USA, January 15–17, pp. 139–149.

Thiemann, P. (1999) Combinators for program generation. *J. Funct. Program.* **9**(5), 483–525.

Thiemann, P. & Dussart, D. (1999) Partial evaluation for higher-order languages with state. Available at: `http://www.informatik.uni-freiburg.de/~thiemann/papers/mlpe.ps.gz` Accessed 8 November 2011.

Tofte, M., Birkedal, L., Elsman, M. & Hallenberg, N. (2004) A retrospective on region-based memory management. *Higher-Order Symb. Comput.* **17**(3), 245–265.

Wadler, P. L. (1992) Comprehending monads. *Math. Struct. Comput. Sci.* **2**(4), 461–493.

Whaley, R. C. & Petitet, A. (2005) Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Softw. Pract. Exp.* **35**(2), 101–121.