Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform

Tiark Rompf Ingo Maier Martin Odersky

Programming Methods Laboratory (LAMP)
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
{firstname.lastname}@epfl.ch

Abstract

We describe the implementation of first-class polymorphic delimited continuations in the programming language Scala. We use Scala's pluggable typing architecture to implement a simple type and effect system, which discriminates expressions with control effects from those without and accurately tracks answer type modification incurred by control effects. To tackle the problem of implementing first-class continuations under the adverse conditions brought upon by the Java VM, we employ a selective CPS transform, which is driven entirely by effect-annotated types and leaves pure code in direct style. Benchmarks indicate that this high-level approach performs competitively.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures

General Terms Languages, Theory

Keywords Delimited continuations, selective CPS transform, control effects, program transformation

1. Introduction

Continuations, and in particular delimited continuations, are a versatile programming tool. Most notably, we are interested in their ability to suspend and resume sequential code paths in a controlled way without syntactic overhead and without being tied to VM threads.

Classical (or *full*) continuations can be seen as a functional version of the infamous GOTO-statement (Strachey and Wadsworth 2000). Delimited (or *partial*, or *composable*) continuations are more like regular functions and less like GOTOs. They do not embody the entire rest of the computation, but just a partial rest, up to a programmer-defined outer bound. Unlike their undelimited counterparts, delimited continuations will actually return control to the caller after they are invoked, and they may also return values. This means that delimited continuations can be called multiple times in succession, and the program can proceed at the call site afterwards. This ability makes delimited continuations strictly more powerful than regular ones. Operationally speaking, delimited continuations do not embody the entire control stack but just stack fragments, so they can be used to recombine stack fragments in interesting and possibly complicated ways.

To access and manipulate delimited continuations in direct-style programs, a number of control operators have been proposed, which can be broadly classified as static or dynamic, according to whether the extent of the continuations they capture is determined statically or not. The dynamic variant is due to Felleisen (1988); Felleisen et al. (1988) and the static variant to Danvy and Filinski (1990, 1992). The static variant has a direct, corresponding CPS-formulation which makes it attractive for an implementation using a static code transformation and thus, this is the variant underlying the implementation described in this paper. We will not go into the details of other variants here, but refer to the literature instead (Dyvbig et al. 2007; Shan 2004; Biernacki et al. 2006); suffice it to note that the two main variants, at least in an untyped setting, are equally expressive and have been shown to be macro-expressible (Felleisen 1991) by each other (Shan 2004; Kiselyov 2005). Applying the type systems of Asai and Kameyama (2007); Kameyama and Yonezawa (2008), however, renders the dynamic control operators strictly more expressive since strong normalization holds only for the static variant (Kameyama and Yonezawa 2008).

In Danvy and Filinski's model, there are two primitive operations, `shift` and `reset`. With `shift`, one can access the current continuation and with `reset`, one can demarcate the boundary up to which continuations reach: A `shift` will capture the control context up to, but not including, the nearest dynamically enclosing `reset` (Biernacki et al. 2006; Shan 2007).

Despite their undisputed expressive power, continuations (and in particular delimited ones) have not yet found their way into the majority of programming languages. Full continuations are standard language constructs in Scheme and popular ML dialects, but most other languages do not support them natively. This is partly because efficient support for continuations is assumed to require special provisions from the runtime system (Clinger et al. 1999), like the ability to capture and restore the run-time stack, which are not available in all environments. In particular, popular VM's such as the JVM or the .NET CLR do not provide this low-level access to the run-time stack. One way to overcome these limitations is to simulate stack inspection with exception handlers and/or external data structures (Pettyjohn et al. 2005; Srinivasan 2006).

Another avenue is to use monads instead of continuations to express custom-defined control flow. Syntactic restrictions imposed by monadic style can be overcome by supporting more language constructs in the monadic level, as is done in F#'s workflow expressions. Nevertheless, the fact remains that monads or workflows impose a certain duplication of syntax constructs that need to be

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.

made available on both monadic and direct style levels. Besides that, it is common knowledge that delimited continuations are able to express any definable monad (Filinski 1994, 1999).

In this paper we pursue a third, more direct alternative: transform programs using delimited continuations into CPS using a type-directed selective CPS transform. Whole program CPS transforms were previously thought to be too inefficient to be practical, unless accompanied by tailor-made compiler backends and runtimes (Appel 1992). However, we show that the more localized CPS-transforms needed for delimited continuations can be implemented on stock VMs in ways that are competitive in terms of performance.

1.1 Contributions

To the best of our knowledge, we are the first to implement direct-style `shift` and `reset` operators with full answer type polymorphism in a statically type-safe manner and integrated into a widely available language.

We implement a simple type and effect system which discriminates pure from impure expressions, while accurately tracking answer type modification and answer type polymorphism. We thereby extend the work of Asai and Kameyama (2007) to a slightly different notion of purity, which identifies all uses of `shift` in a program and thus is a suitable basis for applying a selective CPS transform (Nielsen 2001). To the best of our knowledge, we are the first to use a selective CPS transform to implement delimited continuations.

With our implementation, we present evidence that the standard CPS transform, when applied selectively, is a viable means to efficiently implement static control operators in an adverse environment like the JVM, under the condition that closures are available in the host language.

1.2 Related Work

Filinski (1994) presented an ML-implementation of `shift` and `reset` (using `callcc` and mutable state), which has fixed answer types. Gasbichler and Sperber (2002) describe a direct implementation in Scheme48, which, of course, is not statically typed. Dyvbig et al. (2007) presented a monadic framework for delimited continuations in Haskell, which includes support for `shift` and `reset` among other control operators and supports multiple typed prompts. This framework does not allow answer type modification, though, and the control operators can only be used in monadic style (e.g. using Haskell's `do` notation) but not in direct-style. Kiselyov et al. (2006) presented a direct-style implementation in (bytecode) OCaml, which is inspired by Dyvbig et al.'s framework. The OCaml implementation does not support answer type modification, though, and the type system does not prevent using `shift` outside the dynamic scope of a `reset`. In this case, a runtime exception will occur. All of the above implementations cannot express Asai's type-safe implementation of `printf` (Asai 2007). Kiselyov (2007) gave an adaption of Asai and Kameyama (2007)'s type system (powerful enough to express `printf`) in Haskell, which is fully type safe and provides answer type polymorphism, but cannot be used in direct-style (in fact, due to the use of parameterized monads (Atkey 2006), `do` notation cannot be used either). Asai and Kameyama (2007) did not provide a publicly available implementation of their calculus.

On the JVM, continuations have been implemented using a form of generalized stack inspection (Pettyjohn et al. 2005) in Kilim (Srinivasan 2006; Srinivasan and Mycroft 2008). These continuations can in fact be regarded as delimited, but there is no published account of their delimited nature. Kilim also tracks control effects using a `@pausable` annotation on methods but there are no explicit or definable answer types.

Danvy and Filinski (1989) presented a monomorphic type system for `shift` and `reset`, which was extended by Asai and Kameyama (2007) to provide answer type polymorphism. A type and effect system for full continuations has been presented by Thielecke (2003). None of these allow to identify all uses of `shift` (including trivial ones like in `shift(k => k(3))`) in a given source program. The selective CPS transform has been introduced by Nielsen (2001), but it has not been applied in the context of delimited continuations.

1.3 Overview

The rest of this paper is structured as follows. Section 2 gives a short overview of the Scala language and the language subset relevant to our study. Section 3 is the main part of this paper and describes the typing rules and program transformations which constitute the implementation of delimited continuations in Scala. Section 4 presents programming examples, Section 5 performance figures, and Section 6 concludes.

2. The Host Language

This section gives a quick overview of the language constructs of Scala as far as they are necessary to understand the material in this paper. Scala is different from most other statically typed languages in that it fuses object-oriented and functional programming. Most features from traditional functional languages such as Haskell or ML are also supported by Scala, but sometimes in a slightly different way.

Value definitions in Scala are written

```
val pat = expr
```

where `pat` is a pattern and `expr` is an expression. They correspond to let bindings. Monomorphic function definitions are written

```
def f(params): T = expr
```

where `f` is the function's name, `params` is its parameter list, `T` is its return type and `expr` is its body. Regular by-value parameters are of the form `x: T`, where `T` is a type. Scala also permits by-name parameters, which are written `x: => T`. It is possible to leave out the parameter list of a function completely. In this case, the function's body is evaluated every time the function's name is used.

Definitions in Scala programs are always nested inside classes or objects.[1] A simple class definition is written

```
class C[tparams](params) extends Ds { defs }
```

This defines a class `C` with type parameters given by `tparams`, value parameters given by `params`, superclasses and -traits given by `Ds` and definitions given by `defs`. All components except the class name can be omitted. Classes are instantiated to objects using the `new` operator. One can also define an object directly with almost the same syntax as a class:

```
object O extends Ds { defs }
```

Such an object is an instance of an anonymous class with the given parent classes `Ds` and definitions `defs`. The object is created lazily, the first time it is referenced.

The most important form of type in Scala is a reference to a class `C`, or `C[Ts]` if `C` is parameterized. Unary function types from `S` to `T` are type instances of class `Function1[S, T]`, but one usually uses the abbreviated syntax `S => T` for them. By-name function

[1] Definitions outside classes are supported in the Scala REPL and in Scala scripts but they cannot be accessed from outside their session or script.

types can be written `(=> S) => T`. Most of Scala's libraries are written in an object-oriented style, where operations apply to an implicit `this` receiver object. For instance, a `List` class would support operations `map` and `flatMap` in the following way:

```
class List[T] {
  ...
  def map[U](f: T => U) = this match {
    case Nil => Nil
    case x :: xs => f(x) :: xs.map(f)
  }
  def flatMap[U](f: T => List[U]) = this match {
    case Nil => Nil
    case x :: xs => f(x) ::: xs.flatMap(f)
  }
}
```

Here, `::` is list cons and `:::` is list concatenation. The implementations above also show Scala's approach to pattern matching using `match` expressions. Similar to Haskell and ML, pattern matching blocks that enclose a number of `case`s in braces can also appear outside `match` expressions; they are then treated as function literals.

Most forms of expressions in Scala are written similarly to Java and C, except that the distinction between statements and expressions is less strict. Blocks `{ ... }`, for example, can appear as expressions, including as function arguments. Another departure from Java is support for function literals, which are written with an infix `=>`. For instance, `(x: Int) => x + 1` represents the incrementation function on integers. All binary operations in Scala are treated as method calls. `x op y` is treated as `x.op(y)` for every operator identifier `op`, no matter whether `op` is symbolic or alphanumeric.

In fact, `map` and `flatMap` correspond closely to the operations of a monad. `flatMap` is monadic `bind` and `map` is bind with a `unit` result. Together, they are sufficient to express all monadic expressions as long as injection into the monad is handled on a per-monad basis. Therefore, all that needs to be done to implement monadic unit is to provide a corresponding constructor operation that, in this case, builds one-element lists. Similarly to Haskell and F#, Scala supports monad comprehensions, which are called `for`-expressions. For instance, the expression

```
for (x <- xs; y <- f(x)) yield g(x, y)
```

is expanded to

```
xs.flatMap(x => f(x).map(y => g(x, y)))
```

Definitions as well as parameters can be marked as **implicit**. Implicit parameters that lack an actual argument can be instantiated from an implicit definition that is accessible at the point of call and that matches the type of the parameter. Implicit parameters can simulate the key aspects of Haskell's type classes (Moors et al. 2008). An **implicit** unary function can also be used as a conversion, which implictly maps its domain type to its range.

Definitions and types can be annotated. Annotations are user-defined metadata that can be attached to specific program points. Some annotations are visible at run-time where they can be accessed using reflection. Others are consumed at compile-time by compiler plugins. The Scala compiler has a standardized plugin architecture (Nielsen 2008) which lets users add additional type checking and transformation passes to the compiler.

Syntactically, annotations take the form of a class constructor preceded by an `@`-sign. For instance, the type

```
String @cps[Int, List[Int]]
```

is the type `String`, annotated with an instance of the type `cps` applied to type arguments `Int` and `List[Int]`.

```
reset {
  val x = shift { k: (Int=>Int) =>
    "done here"
  }
  println(x)
}
```

No output (continuation not invoked)

```
reset {
  val x = shift { k: (Int=>Int) =>
    k(7)
  }
  println(x)
}
```

Output: 7

```
val x = reset {
  shift { k: (Int=>Int) =>
    k(7)
  } + 1
} * 2
println(x)
```

Output: 16

```
val x = reset {
  shift { k: (Int=>Int) =>
    k(k(k(7)))
  } + 1
} * 2
println(x)
```

Output: 20

```
val x = reset {
  shift { k: (Int=>Int) =>
    k(k(k(7))); "done"
  } + 1
}
println(x)
```

Output: “done”

```
def foo() = {
  1 + shift(k => k(k(k(7))))
}
def bar() = {
  foo() * 2
}
def baz() = {
  reset(bar())
}
println(baz())
```

Output: 70

Figure 1. Examples: `shift` and `reset`

In this paper, we study the addition of control operators `shift` and `reset` to this language framework, which together implement static delimited continuations (Danvy and Filinski 1990, 1992). The operational semantics of `shift` is similar to that of `callcc` in languages like Scheme or ML, except that a continuation is only captured up to the nearest enclosing `reset` and the capturing is always abortive (i.e. the continuation must be invoked explicitly). Figure 1 presents some examples to illuminate the relevant cases.

3. Implementation

Broadly speaking, there are two ways to implement continuations (see (Clinger et al. 1999) for a more detailed account). One is to stick with a stack-based execution architecture and to reify the current continuation by making a copy of the stack, which is reinstated

when the continuation is invoked. This is the approach taken by many language implementations that are in direct control of the runtime system. Direct implementations of delimited continuations using an incremental stack/heap strategy have also been described (Gasbichler and Sperber 2002). In the Java world, stack-copying has been used to implement continuations on the Ovm virtual machine (Dragos et al. 2007). For Scala, though, this is not a viable option, since Scala programs need to run on plain, unmodified, JVMs, which do not permit direct access or modification of stack contents. A variant of direct stack inspection is generalized stack inspection (Pettyjohn et al. 2005), which uses auxiliary data structures to simulate continuation marks that are not available on the JVM or CLR architectures. That approach is picked up and refined by Kilim (Srinivasan 2006; Srinivasan and Mycroft 2008), which transforms compiled programs at the bytecode-level, inserting copy and restore instructions to save the stack contents into a separate data structure (called a *fiber*) when a continuation is to be accessed.

The other approach is to transform programs into continuation-passing-style (CPS) (Appel and Jim 1989; Danvy and Filinski 1992). Unfortunately, the standard CPS-transform is a whole-program transformation. All explicit or implicit `return` statements are replaced by function calls and all state is kept in closures, completely bypassing the stack. For a stack-based architecture like the JVM, of course, this is not a good fit.

On the other hand, regarding manually written CPS code shows that only a small number of functions in a program actually need to pass along continuations. What we are striving for is thus a *selective* CPS transform (Nielsen 2001) that is applied only where it is actually needed, and allows us to stick to a regular, stack-based runtime discipline for the majority of code. As a side effect, this by design avoids the performance problems associated with implementations of delimited continuations in terms of undelimited ones (Balat and Danvy 1997; Gasbichler and Sperber 2002). In general, a CPS transform is feasible only if the underlying architecture supports constant-space tail-calls, which is the case for the .NET CLR but not yet for the JVM[2]. So far, we have not found this a problem in practice. One reason is that for many use-cases of delimited continuations, call depth tends to be rather small. Moreover, some applications lend themselves to uses of `shift` and `reset` as parts of other abstractions, which allow a transparent inclusion of a trampolining facility, in fact introducing a back-door tail-call optimization.

3.1 Syntax-Directed Selective CPS Transform

Taking a step back, we consider how we might implement delimited continuations as user-level Scala code. The technique we use comes as no surprise and is a straightforward generalization of the continuation monad to one that is parametric in the answer types. On a theoretical level, parameterized monads have been studied in (Atkey 2006).

As a first step, we define a wrapper class to hold `shift` blocks and provide methods to extend and compose them. The method `flatMap` is the monadic bind operation:

```
class Shift[+A,-B,+C](val fun: (A => B) => C) {

  def map[A1](f: (A => A1)): Shift[A1,B,C] = {
    new Shift((k: (A1 => B)) =>
      fun((x:A) => k(f(x))))
  }

  def flatMap[A1,B1,C1<:B](f: (A =>
          Shift[A1,B1,C1])): Shift[A1,B1,C] = {
    new Shift((k: (A1 => B1)) =>
        fun((x:A) => f(x).fun(k)))
    }
  }
```

Note the `+/-` variance annotations on the type parameters of class `Shift`, which make the class covariant in parameters `A` and `C` and contravariant in `B`. This makes `Shift` objects consistent with the subtyping behavior of the parameter `fun`.

We go on by defining `reset` to operate on `Shift` objects, invoking the body of a given `shift` block with the identity function to pass the result back into the body (which is the standard CPS definition of `reset`):

```
def reset[A,C](c: Shift[A,A,C]) = c.fun(x:A => x)
```

With these definitions in place, we can use delimited continuations by placing `Shift` blocks in `for` comprehensions, which are Scala's analog to the `do` notation in Haskell:

```
val ctx = for {
  x <- new Shift((k:Int=>Int) => k(k(k(7))))
} yield (x + 1)

reset(ctx) // 10
```

This works because during parsing, the Scala compiler desugars the `for` comprehension into invocations of `map` and `flatMap`:

```
val ctx = new Shift((k:Int=>Int) => k(k(k(7))))
                                      .map(x => x + 1)
reset(ctx) // 10
```

So for all practical matters, we have a perfectly workable selective CPS transform, albeit a *syntax-directed* one, i.e. one which is carried out by the parser on the basis of purely syntactic criteria, more specifically the placement of the keywords `for` and `yield`. Being forced to use `for` comprehensions everywhere continuations are accessed does not make for a pleasant programming style, though. Instead, we would like our CPS to be *type-directed*, i.e. carried out by the compiler on the basis of expression types.

3.2 Effect-Annotated Types

The motivation for this approach is to transparently mix code that must be transformed with code that does not. Therefore, we have to disguise the type `Shift[A,B,C]` as something else, notably something that is compatible with type `A` because `A` is the argument type of the expected continuation (recall the definition of `Shift`). Thus, we make use of Scala's pluggable typing facilities and introduce a *type annotation* `@cps[-B,+C]`, with the intention that any expression of type `A @cps[B,C]` should be translated to an expression of type `Shift[A,B,C]`.

The approach of using annotated types to track control effects has a close correspondence to the work on polymorphic delimited continuations (Asai and Kameyama 2007). It has been noted early (Danvy and Filinski 1989) that in the presence of control operators, standard typing judgements of the form $\Gamma \vdash e : \tau$, which associate a single result type τ with an expression e, are insufficient to accurately describe the result of evaluating the expression e. The reason is that evaluating e may change the *answer type* of the enclosing computation. In the original type system by Danvy and Filinski (1989), typing judgements thus have the form

$$\Gamma; \alpha \vdash e : \tau; \beta$$

meaning that "if e is evaluated in a context represented by a function from τ to α, the type of the result will be β" or equivalently "In a context where the (original) result type was α, the type of e is τ and the new type of the result will be β".

[2] Tail-call support for the JVM has been proposed by JSR 292 (Rose 2008)

Asai and Kameyama (2007) present a polymorphic extension of this (monomorphic) type system and prove a number of desirable properties, which include strong type soundness, existence of principal types and an efficient type inference algorithm. A key observation is that if e does not modify its context, α and β will be identical and if $\Gamma; \alpha \vdash e : \tau; \alpha$ is derivable for any α, the expression does not have any measurable control effect. In other words, *pure* expressions (e.g. values) are intuitively characterized as being polymorphic in the answer type and not modifying it (Thielecke 2003).

Pure expressions such as lambda abstractions (or other values) should thus be allowed to be used polymorphically in the language. The ability to define functions that are polymorphic in how they modify the answer type when invoked plays a crucial role e.g. in the implementation of type-safe `printf` (Asai 2007). Asai and Kameyama therefore use two kinds of typing judgements to distinguish pure from impure expressions, and they require that only pure expressions be used in right-hand sides of `let`-bindings, in order to keep their (predicative) type system sound. The kinds of judgements used are

$$\Gamma \vdash_p e : \tau \qquad \Gamma; \alpha \vdash e : \tau; \beta$$

for pure and impure expressions, respectively. Expressions classified as pure are variables, constants, lambda abstractions and uses of `reset`. In addition, pure expressions can be treated as impure ones that do not change the answer type:

$$\frac{\Gamma \vdash_p e : \tau}{\Gamma; \alpha \vdash e : \tau; \alpha}$$

Instead of the standard function types $\sigma \to \tau$, types of the form $\sigma/\alpha \to \tau/\beta$ are used (denoting a change in the answer type from α to β when the function is applied).

3.3 Pure is not Pure

For our goal of applying a selective CPS transform, we need a slightly different notion of purity. Since we have to transform all expressions that actually access their continuation (and only those), we have to be able to identify them accurately. Neither the intuitive notion of purity nor the purity judgement of Asai and Kameyama does provide this classification. For example, the expression `shift(k => k(3))`, which needs to be transformed, would be characterized as pure in the intuitive sense (it is polymorphic in the answer type and does not modify it) but the purity judgement is not applicable. The expression `id(3)`, however, which should not be CPS-transformed, is intuitively pure but impure as defined by applicability of the purity judgement, as are function applications in general.

We thus define an expression as pure if and only if it does not reify its continuation via `shift` in the course of its evaluation. In order to adapt the effect typing to this modified notion of purity we use a slightly different formulation. We keep the standard typing judgements of the form $\Gamma \vdash e : \tau$, but we enrich the types themselves. That is, τ can be either A, denoting a pure type, or A `@cps[`B`,`C`]`, denoting an impure type that describes a change of the answer type from B to C. We will write $A\ \alpha$ when we talk about both pure and impure types.

We present typing rules for a selected subset of the Scala language in Figure 2. Impure types are introduced by `shift` expressions (SHIFT).[3] Impure types are eliminated by `reify` expressions

$$\frac{\Gamma \vdash f : (A \Rightarrow B) \Rightarrow C}{\Gamma \vdash \texttt{shift}(f) : A\ \texttt{@cps[}B\texttt{,}C\texttt{]}} \quad \text{(SHIFT)}$$

$$\frac{\Gamma \vdash c : A\ \texttt{@cps[}B\texttt{,}C\texttt{]}}{\Gamma \vdash \texttt{reify}(c) : \texttt{Shift[}A\texttt{,}B\texttt{,}C\texttt{]}} \quad \text{(REIFY)}$$

$$\frac{\Gamma \vdash f : (A \Rightarrow (B\ \gamma))\ \alpha \quad \Gamma \vdash e : A\ \beta \qquad \delta = \mathit{comp}(\alpha\, \beta\, \gamma)}{\Gamma \vdash f(e) : B\ \delta} \quad \text{(APP-VALUE)}$$

$$\frac{\Gamma \vdash f : ((\Rightarrow A\ \beta) \Rightarrow (B\ \gamma))\ \alpha \quad \Gamma \vdash e : A\ \beta \qquad \delta = \mathit{comp}(\alpha\, \gamma)}{\Gamma \vdash f(e) : B\ \delta} \quad \text{(APP-NAME)}$$

$$\frac{\Gamma \vdash f : ((\Rightarrow A\ \texttt{@cps[}C\texttt{,}C\texttt{]}) \Rightarrow (B\ \gamma))\ \alpha \quad \Gamma \vdash e : A \qquad \delta = \mathit{comp}(\alpha\, \gamma)}{\Gamma \vdash f(e) : B\ \delta}$$

(APP-DEMOTE)

$$\frac{\Gamma, x : A, f : (A \Rightarrow B\ \beta) \vdash e : B\ \beta \quad \Gamma, f : (A \Rightarrow B\ \beta) \vdash \{\overline{r}\} : C\ \gamma}{\Gamma \vdash \texttt{def}\ f(x\texttt{:}A)\texttt{:}\ B\ \beta\ \texttt{=}\ e\texttt{;}\ \overline{r}\ \ \texttt{:}\, C\ \gamma}$$

(DEF-CBV)

$$\frac{\Gamma, x : A\ \alpha, f : ((\Rightarrow A\ \alpha) \Rightarrow B\ \beta) \vdash e : B\ \beta \quad \Gamma, f : ((\Rightarrow A\ \alpha) \Rightarrow B\ \beta) \vdash \{\overline{r}\} : C\ \gamma}{\Gamma \vdash \texttt{def}\ f(x\texttt{:}\Rightarrow A\ \alpha)\texttt{:}\ B\ \beta\ \texttt{=}\ e\texttt{;}\ \overline{r}\ \ \texttt{:}\, C\ \gamma}$$

(DEF-CBN)

$$\frac{\Gamma \vdash e : A\ \alpha \qquad \Gamma, x : A \vdash \{\overline{r}\} : B\ \beta \qquad \delta = \mathit{comp}(\alpha\ \beta)}{\Gamma \vdash \texttt{val}\ x\texttt{:}\ A\ \texttt{=}\ e\texttt{;}\ \overline{r}\ \ \texttt{:}\, B\ \delta}$$

(VAL)

$$\frac{\Gamma \vdash \overline{s : A\ \alpha} \qquad \Gamma \vdash e : B\ \beta \qquad \delta = \mathit{comp}(\overline{\alpha}\ \beta)}{\Gamma \vdash \{\overline{s}\texttt{;}\ e\} : B\ \delta} \quad \text{(SEQ)}$$

Figure 2. Typing rules for a selected subset of Scala expressions. Lowercase letters are expressions, uppercase letters types without annotations, greek letters are either annotations or no annotations.

$$\mathit{comp}(\epsilon) = \epsilon$$

$$\mathit{comp}(\texttt{@cps[}B\texttt{,}C\texttt{]}) = \texttt{@cps[}B\texttt{,}C\texttt{]}$$

$$\frac{\mathit{comp}(\overline{\alpha}) = \texttt{@cps[}V\texttt{,}W\texttt{]} \qquad W <: B}{\mathit{comp}(\texttt{@cps[}B\texttt{,}C\texttt{]}\,\overline{\alpha}) = \texttt{@cps[}V\texttt{,}C\texttt{]}}$$

$$\frac{(A \Rightarrow B) \Rightarrow C\ <:\ (U \Rightarrow V) \Rightarrow W}{A\ \texttt{@cps[}B\texttt{,}C\texttt{]}\ <:\ U\ \texttt{@cps[}V\texttt{,}W\texttt{]}}$$

Figure 3. Composition of control effects and subtyping between annotated types. Lowercase letters are expressions, uppercase letters are types without annotations, ϵ denotes the empty sequence of annotations, other greek letters are either annotations or no annotations.

[3] The given typing of `shift(`f`)`, which requires f to be a pure function, is slightly different from the usual presentation, which would allow f to have non-trivial control effects itself. This is no limitation, however, since the usual definition of `shift` wraps its body with an implicit `reset`. Thus, adding an explicit `reset` around the body of an impure f will make the expression well-typed and achieve the standard behavior.

(REIFY), which allow a Scala program to directly access the `Shift` object that results from CPS-transforming an expression. We show in Section 3.4 how to express `reset` in terms of `reify`.

As opposed to the type system of Asai and Kameyama, which provides no distinction between pure and impure *functions*, we can distinguish the two cases by looking at the return type. If no `@cps` annotation is present, and only then, the function is considered pure. Functions with a single by-value parameter are of the type A `=>` $(B\ \beta)$ (DEF-CBV). That the effect of applying a function is coupled to its return type is consistent with the intuitive assumption that the return type describes what happens when the function is applied. The formal parameter is not allowed to have an effect. This is also intuitively consistent, because for a by-value parameter, any effect the evaluation of an argument might have will occur at the call site (APP-VALUE), so inside the function, accessing the argument will be effect-free. On the other hand, functions in Scala can also have by-name parameters. A function with a single by-name parameter will have the type $(\texttt{=>}A\ \alpha)$ `=>` $(B\ \beta)$ (DEF-CBN), which is consistent with the assumption that the effect of evaluating the argument now happens inside the function body (APP-NAME). If a function taking an impure parameter is applied to a pure expression, the argument is demoted to impure provided that the parameter type does not demand changing the answer type (APP-DEMOTE). The typing rules for other kinds of expressions (e.g. conditionals) are similar in spirit to those presented for function applications.

Since functions can be polymorphic in their answer type modification, we allow right-hand-sides of `def` statements to be impure. We also allow impure expressions in `val` definitions (these are monomorphic in Scala) that occur inside methods, but the identifier will have a pure type since the effect occurs during evaluation of the right hand side and has already happened once the identifier is assigned (VAL). The effect is instead accounted to the enclosing block (SEQ). In Scala, `val` definitions are also used to define object or class level fields. Contrary to those inside methods, these `val` definitions are required to be pure since we cannot capture a continuation while constructing an object.

Subtyping between impure types takes the annotations into account, with proper variances as induced by the corresponding CPS representation (see Figure 3). There is no subtyping or general subsumption between pure and impure types, but pure expressions may be treated as impure (and thus polymorphic in the answer type) where required, as defined by the typing rules in Figure 2. This is a subtle difference that allows us to keep track of the purity of each expression and prevents the loss of accuracy associated with Asai and Kameyama's subsumptive treatment of pure expressions as impure ones. The CPS transform will detect these conversion cases in the code and insert `shift`s where necessary to demote pure expressions to impure ones. In addition, impure expressions can be treated as pure ones in all *by-value* places, i.e. those where the expression is reduced to a value. The expression's effect (which happens during evaluation) is then accounted to the enclosing expression.

And this is exactly what will drive the selective CPS conversion: Every use of an impure expression as a pure one must reify the context as a continuation and explicitly pass it to the translated impure expression.

When we say "accounted to the enclosing expression", we are actually a bit imprecise. The correct way to put it is that every expression's cumulated control effect (which may be none) is a composition of its by-value subexpressions' control effects in the order of their evaluation. For such a composition to exist, the answer types must be compatible (according to the standard rules, also manifest in the type constraints of the class `Shift` and its method `flatMap`). During composition, pure expressions are treated neutrally. This is how we achieve answer type polymorphism in our system. If no composition exists, a type error is signaled. The rules that define the composition relation are given in Figure 3.

$$\frac{f : (A \texttt{ => } B) \texttt{ => } C}{[\![\texttt{shift}(f)]\!] = \texttt{new Shift}[A,B,C](f)}$$
(SHIFT)

$$\frac{c : A \texttt{ @cps}[B,C]}{[\![\texttt{reify}(c)]\!] = [\![c]\!]}$$
(REIFY)

$$\frac{e : A \texttt{ @cps}[B,C] \quad \{[\![\overline{r}]\!]\} : U \texttt{ @cps}[V,W]}{[\![\texttt{val } x\texttt{: } A = e\texttt{; } \overline{r}]\!] = [\![e]\!]\texttt{.flatMap}(x\texttt{:}A \texttt{ => } \{[\![\overline{r}]\!]\})}$$
(VAL-IMPURE)

$$\frac{e : A \texttt{ @cps}[B,C] \quad \{[\![\overline{r}]\!]\} : U}{[\![\texttt{val } x\texttt{: } A = e\texttt{; } \overline{r}]\!] = [\![e]\!]\texttt{.map}(x\texttt{:}A \texttt{ => } \{[\![\overline{r}]\!]\})}$$
(VAL-PURE)

$$[\![\texttt{def } f(x\texttt{:}A) = e\texttt{; } \overline{r}]\!] = \texttt{def } f(x\texttt{:}A) = [\![e]\!]\texttt{; } [\![\overline{r}]\!]$$
(DEF)

$$[\![s\texttt{; } \overline{r}]\!] = s\texttt{; } [\![\overline{r}]\!] \qquad [\![\{\overline{r}\}]\!] = \{[\![\overline{r}]\!]\}$$
(SEQ)

Figure 4. Type-directed selective CPS transform for a subset of Scala. Lowercase italic letters denote untransformed expressions, uppercase letters expression sequences or types. Rules are applied deterministically from top to bottom.

3.4 Type-Directed Transformation

We will define the selective CPS transform in two steps and start with the one that comes last. A subset of the transformation rules is shown in Figure 4.

We denote the transformation itself by $[\![.]\!]$, and we let Scala programs access transformed expressions with the primitive `reify` (REIFY). Invocations of `shift` are translated to creations of `Shift` objects (SHIFT). If an impure expression appears on the right-hand-side of a value definition, which is followed by a sequence of expressions, then the right-hand-side is translated to a `Shift` object, upon which `flatMap` or `map` is invoked with the translated remaining expressions wrapped up into an anonymous function (VAL-IMPURE,VAL-PURE). Whether `flatMap` or `map` is used depends on whether the remaining expressions are translated to a `Shift` object or not. The semantics of `shift` require to flatten out nested `Shift` objects because otherwise, multiple nested `reset` handlers would be needed to get at the result of a sequence of `shift` blocks. In this case, all but the first `shift` would escape the enclosing `reset`[4]. The use of `map` is an optimization. We could as well wrap the remaining code into a stub `shift` that behaves as identity and then use `flatMap`. But that would introduce unnecessary "administrative" redexes, which customary CPS-transform algorithms go to great lengths to avoid (Danvy et al. 2007). Right-hand sides of function definitions are translated independently of the context (DEF). Finally, block expressions $\{\ldots\}$

[4] This is in fact Felleisen's model (Felleisen 1988)

$$[\![\{\overline{r};\ e\}]\!] = \{[\![\overline{r}]\!]_{Inline};\ [\![e]\!]\}$$

$$[\![s;\ \overline{r}]\!]_{Inline} = [\![s]\!]_{Inline};\ [\![\overline{r}]\!]_{Inline}$$

$$\frac{[\![e]\!] = \overline{r};\ g \qquad g : A\ \texttt{@cps[}B\texttt{,}C\texttt{]}}{[\![e]\!]_{Inline} = \overline{r};\ \texttt{val}\ x\texttt{:}\ A\ \texttt{=}\ g;\ x}$$

$$\frac{[\![f]\!]_{Inline} = \overline{r};\ g \qquad [\![e]\!]_{Inline} = \overline{s};\ h}{[\![f\,(e)\,]\!] = \overline{r};\ \overline{s};\ g\,(h)} \quad \text{(BY-VALUE APPLY)}$$

$$\frac{[\![f]\!]_{Inline} = \overline{r};\ g \qquad [\![e]\!]_{Inline} = \overline{s};\ h}{[\![f\,(e)\,]\!] = \overline{r};\ g\,(\{\overline{s};\ h\})} \quad \text{(BY-NAME APPLY)}$$

Figure 5. Selective ANF transform (only selected rules shown). Lowercase italic letters denote untransformed expressions, uppercase letters expression sequences or types.

are translated by applying the other rules to the enclosed expression sequence, possibly skipping a prefix of non-CPS terms (SEQ).

Applying the transformation rules given in Figure 4, we can transform code like

```
reset(reify {
  val x = shift(k => k(k(k(7))))
  x + 1
})
```

into the following:

```
reset(new Shift(k => k(k(k(7)))).map(x => x + 1))
```

We are still somewhat restricted, though, in that CPS expressions may only appear in value definitions. Fortunately, we can reach this form by a pre-transform step, which assigns synthetic names to the relevant intermediate values, lifting them into value definitions. In analogy to the selective CPS transform, we can describe this step as a selective ANF transform (*administrative normal form* (Flanagan et al. 1993)). We present a subset of the transformation rules in Figure 5.

For the ANF pre-transform, we use two mutually recursive functions, $[\![.]\!]$ and $[\![.]\!]_{Inline}$ that map expressions to expression sequences ($[\![.]\!]_{Inline}$ is extended pointwise to expression sequences). The latter is used to lift nested CPS-expressions and insert a `val` definition *inline*, preceding the parent expression. Since we do not want to introduce value definitions for expressions that are already in tail position, we use either transformation depending on the context. Again, we illustrate the main principle by considering function applications. We consider application of functions with a single by-value parameter first. The function and the argument are nested expressions and thus transformed using $[\![.]\!]_{Inline}$, each of them yielding a statement sequence followed by an expression. The Scala semantics demand that the function be evaluated first, so the result is the function's statements, followed by the argument's statements, followed by applying the expressions. When considering functions with by-name parameters, by contrast, the statements that result from transforming the corresponding argument must not be inserted preceding the application. In this case, the whole resulting sequence is wrapped up in a block and passed as an argument to the transformed function. For other kinds of Scala expressions like conditionals, pattern matching, etc., the transformation works accordingly, depending on the context whether the by-name or by-value style is used.

Note that in Figure 5, the insertion of new value definitions is triggered by a `@cps` annotation on the *result* of transforming the expression in question. While this is a sound premise in the description at hand, we actually make sure in the implementation that the expression *itself* is annotated accordingly. This is done by an *annotation checker*, which hooks into the typer phase of the Scala compiler, promoting CPS annotations outwards to expressions that have nested CPS expressions in positions where $[\![.]\!]_{Inline}$ will be applied. In the actual implementation, the selective ANF transform is also slightly more complex than described here. One reason is that we have to accommodate for possibly erroneous programs. Therefore, the actual transform takes two additional parameters, namely an *expected* `@cps` annotation (or *none* if a pure expression is expected) for the current expression sequence and an *actual* one, which is built up as we go along. When reaching the end of an expression sequence, these two must either match, or, if an annotation is expected but none is present, an implicit `shift` is inserted that behaves as identity.

Summing up the transformation steps, we implement `reset` in terms of `reify`, using a by-name parameter:

```
def reset[A,C](ctx: => A @cps[A,C]) = {
  reify(ctx).fun(x:A => x)
}
```

Finally, we can express our working example as

```
reset {
  shift(k => k(k(k(7)))) + 1
}
```

which is exactly what was intended.

4. Programming Examples

There are many well-known use cases for delimited continuations and most of them can be implemented in Scala straightforwardly.

4.1 Type-Safe `printf`

As a first example, we present the Scala implementation of type-safe `printf` (Danvy 1998; Asai 2007):

```
val int = (x: Int) => x.toString
val str = (x: String) => x

def format[A,B](toStr: A => String) = shift {
  k: (String => B) => (x:A) => k(toStr(x))
}

def sprintf[A](str: =>String @cps[String,A]) = {
  reset(str)
}

val f1 = sprintf[String]("Hello World!")
val f2 = sprintf("Hello " +
  format[String,String](str) + "!")
val f3 = sprintf("The value of " +
  format[String,Int=>String](str) +
  " is " + format[Int,String](int) + ".")

println(f1)
println(f2("World"))
println(f3("x")(3))
```

This example is instructive for its use of both answer type modification and answer type polymorphism. As we can see in the code above, `format` takes a function that converts a value of type A to a string. In addition, it modifies the answer type from any type B to a function from A to B. In a context whose result type is `String`, invoking `format(int)` will change the answer

type to `Int => String`; an additional integer argument has to be provided to turn the result into a string. Unfortunately, Scala's local type inference cannot reconstruct all the type parameters here so we must give explicit type arguments for uses of `format`.

4.2 Direct-Style Monads

Another interesting example is the use of monads in direct-style programming (Filinski 1994, 1999). As has been shown by Filinski, delimited continuations can express any definable monad. We identify monadic types structurally by the existence of a *bind* operation (`flatMap`), making use of type constructor polymorphism (Moors et al. 2008) to describe its required signature:

```
type Monadic[+U, C[_]] = {
  def flatMap[V](f: U => C[V]): C[V]
}
```

We go on by defining an adapter class that allows to *reflect* monadic values, passing the current continuation to the underlying monadic bind operation:

```
class Reflective[+A, C[_]](xs: Monadic[A,C]) {
  def reflect[B](): A @cps[C[B], C[B]] = {
    shift { k:(A => C[B]) =>
      xs.flatMap(k)
    }
  }
}
```

Defining an implicit conversion for iterables, we can e.g. use the list monad in direct-style. The unit constructor `List` is used here as Filinski's *reify* operation:

```
implicit def reflective[A](xs:Iterable[A]) =
  new Reflective[A,Iterable](xs)

reset {
  val left = List("x","y","z")
  val right = List(4,5,6)

  List((left.reflect[Any], right.reflect[Any]))
}
// result: cartesian product
```

The same mechanism applies to other monads, too. Using the option monad, for example, we can build a custom exception handling mechanism and the state monad could be used as an alternative to thread-local variables.

4.3 Concurrency Primitives

Using delimited continuations, we can implement a rich variety of primitives for concurrent programming. Among others, these include bounded and unbounded buffers, rendezvous cells, futures, single-assignment variables, actor mailboxes, and join patterns. Without going into the details of the implementation, we show how our implementation of extensible join patterns or *dynamic functional nets* (Fournet and Gonthier 1996; Odersky 2000; Rompf 2007) can integrate join-calculus based programming into Scala. The following code implements, with a common interface, synchronous rendezvous cells and asynchronous reference cells backed by a one-place buffer:

```
abstract class ReferenceCell[A] {
  val put = new (A ==> Unit)
  val get = new (Unit ==> A)
}

// synchronous reference cell (no buffering)
class SyncRefCell[A] extends ReferenceCell[A] {
  join {
    case put(x <== return_put)
                <&> get(_ <== return_get) =>
      println("exchanging " + x)
      return_put() <&> return_get(x)
  }
}

// asynchronous reference cell (1-place buffer)
class AsyncRefCell[A] extends ReferenceCell[A] {
  val empty = new (Unit ==> Unit)
  val item = new (A ==> Unit)
  join {
    case put(x <== return_put)
                <&> empty(_ <== _) =>
      return_put() <&> item(x)
  }
  join {
    case get(_ <== return_get)
                <&> item(x <== _) =>
      println("exchanging " + x)
      return_get(x) <&> empty()
  }
  spawn {
    empty()
  }
}
```

4.4 Actors

Scala Actors provide an implementation of the actor model (Agha and Hewitt 1987) on the JVM (Haller and Odersky 2009). To make efficient use of VM threads, actors, when waiting for incoming messages, can suspend in event-based mode with an explicitly passed continuation closure instead of blocking the underlying thread (Haller and Odersky 2006). This is accomplished by the primitive `react` that takes a message handler (the continuation closure), and suspends the current actor in event mode. The underlying (pool) thread is released to execute other runnable actors. Using `react`, however, imposes some restrictions on the program structure. In particular, no code *following* a `react` is ever executed, only the explicitly provided closure.

For example, consider implementing a communication protocol using actors. It would be tempting to handle the connection setup in a separate method:

```
def establishConnection() = {
  server ! SYN
  react {
    case SYN_ACK =>
      server ! ACK
  }
}
```

which is then used as part of a more complex actor behavior:

```
actor {
  establishConnection()
  transferData()
  ...
}
```

But unfortunately, this does not work as is. The use of `react` inside `establishConnection` precludes the execution of `transferData`. To make this example work, one would have to

use explicit `andThen` combinators to chain the individual pieces of behavior together. In the presence of complex control structures, programming in this style quickly becomes cumbersome. In addition, the type system does not enforce the use of combinators so errors will manifest only at runtime.

Using delimited continuations, we can simplify programming event-based actors significantly. Moreover, we can do so without changing the implementation of the existing primitives, thereby maintaining the high degree of interoperability with standard Java threads (Haller and Odersky 2009). This approach, which has been suggested by Philipp Haller, introduces a higher-order function `proceed` that can be applied to `react`, such that the message handling closure is extended with the current continuation:

```
def proceed[A, B](fun: PartialFunction[A, Unit] =>
                                          Nothing):
  PartialFunction[A, B] => B @cps[Unit, Unit] =
    (cases: PartialFunction[A, B]) =>
      shift((k: B => Unit) => fun(cases andThen k))
```

Wrapping each `react` with a `proceed` and inserting a `reset` to delineate the actor behavior's outer bound we can actually code the above example as follows. It is worth mentioning that leaving out the `reset` would cause the compiler to signal a type error, since an impure expression would occur in a pure context:

```
def establishConnection() = {
  server ! SYN
  proceed(react) {
    case SYN_ACK =>
      server ! ACK
  }
}
actor {
  reset {
    establishConnection()
    transferData()
    ...
  }
}
```

Alternatively, the implementations of `react` and `actor` could be modified to make use of the necessary control operators directly. But using `proceed` is a good example of incorporating delimited continuations into existing code in a backwards-compatible way.

4.5 Functional Reactive Programming

Functional reactive programming (FRP) is an effort to integrate reactive programming concepts into functional programming languages (Elliott and Hudak 1997; Courtney et al. 2003; Cooper and Krishnamurthi 2006). The two fundamental abstractions of FRP are signals and event streams[5]. A signal represents a continuous time-varying value; it holds a value at every point in time. An event stream, on the other hand, is discrete; it yields a value only at certain times. Signals and event streams can be composed through combinators, some of which are known from functional collections, such as `map`, `flatMap`, or `filter`.

Our implementation of a reactive library in Scala takes the basic ideas of FRP and extends it with support for imperative programming. One key abstraction to achieve this is called *behaviors*. A behavior can be used to conveniently react to complex event patterns in an imperative way. To give an idea how behaviors work, we take an example from our user interface toolkit whose event handling details are implemented exclusively with our reactive programming library. The interactive behavior of a button widget can be implemented as follows:

```
behavior {
  next(mouse.leftDown)
  showPressed(true)
  val t = loop {
    showPressed(next(mouse.hovers.changes))
  }
  next(mouse.leftUp)
  t.done()
  showPressed(false)
  if (mouse.hovers.now) performClick()
}
```

The first action of the behavior above is to wait until the left mouse button is pressed down. Method `next` blocks the current behavior and returns the next message that becomes available in a given event stream. In our case, the behavior drops that message and then updates the button view and starts a loop. A loop is a child behavior that is automatically terminated when the current cycle of the parent behavior ends. The loop updates the button view whenever the mouse enters or leaves the button area. We do this by waiting for changes in the boolean signal `mouse.hovers`, which indicates whether the mouse currently hovers over the button widget. The call to `changes` converts that boolean signal to an event stream that yields boolean messages. We use the event message to determine whether the mouse button is currently over the button. In the parent behavior, in parallel to the loop, we wait for the left mouse button to be released. On release, we terminate the loop by calling `done`, which causes the child behavior to stop after it has processed all pending events. Note that this does not lead to race conditions since, in constrast to actors, behaviors are executed sequentially and should not be accessed from different threads. Eventually, we update the button view and perform a click if the mouse button has been released while inside the bounds of the button widget.

The use of the CPS transform API is hidden inside `behavior`, `next`, and `loop`. Methods `behavior` and `loop` delimit the continuation scope while method `next` captures the continuation and passes it to an event stream observer which invokes the continuation on notification.

4.6 Asynchronous IO

Using a similar model, we can use scalable asynchronous IO primitves in a high-level declarative programming style. Below, we consider the Scala adaptation of an asynchronous webserver presented in (Rompf 2007). The basic mechanism is to request a stream of callbacks matching a set of keys from a *selector*. This stream can be iterated over and transformed into a higher-level stream by implementing the standard iteration methods (e.g. `foreach`) used by Scala's `for` comprehensions. A stream of sockets representing incoming connections can be implemented like this:

```
def acceptConnections(sel: Selector, port: Int) =
  new Object {
    def foreach(body: (SocketChannel =>
                          Unit @suspendable)) = {
      val serverSock = ServerSocketChannel.open()
      for (key <- callbacks(serverSock, sel,
                    SelectionKey.OP_ACCEPT)) {
        body(serverSock.accept())
      }
    }
  }
```

[5] We use different terminology than most FRP implementations, who use the term *behavior* for signals. In our implementation, this term is reserved for a different concept as discussed below.

Note that the client handler (the parameter `body` of `foreach`) might capture a continuation and thus interrupt the accepting of new connections. The annotation `@suspendable` expresses the common case of a control effect that occurs in a context with answer type `Unit` and keeps the answer type unchanged:

```
type suspendable = cps[Unit,Unit]
```

In the same way as above, we can implement a stream of events indicating incoming data on a specific socket:

```
def readBytes(selector: Selector, socketChannel:
                                        SocketChannel) =
  new Object {
    def foreach(body: (ByteBuffer =>
                           Unit @cps[Unit,Unit])) =
      val buf = ByteBuffer.allocateDirect(4096)
      for (key <- callbacks(socketChannel,
                  selector, SelectionKey.OP_READ)) {
        buf.flip()
        body(buf)
        buf.clear()
      }
  }
```

Adapting this stream of incoming data, we can build a stream of incoming requests, which invokes its handler only when a new, complete request has been parsed. Using these building blocks, we can define the main server loop:

```
val sel = createAsyncSelector()
for (socketChannel <- acceptConnections(sel, 80)) {
  spawn {
    println("Connect: " + socketChannel)

    for (req <- readRequests(sel, socketChannel)) {
      val res = handleRequest(req)
      writeResponse(sel, socketChannel, res)
    }

    println("Disconnect: " + socketChannel)
  }
}
```

Using `spawn` to offload handling of requests to a thread pool, this server loop, despite its strictly sequential appearance, can handle large numbers of concurrently active connections using scalable asynchronous IO with only few native platform threads.

5. Performance

In assessing the performance of our implementation of delimited continuations, we focus on comparing running times with other means of implementing continuations on the JVM first. The approach of Kilim (Srinivasan 2006; Srinivasan and Mycroft 2008) reportedly exhibits very good performance and is thus a natural target for a head-to-head comparison.

All numbers were taken on a late-2008 MacBook Pro (Intel Core 2 Duo, 2.4GHz, 4GB RAM) running MacOS X 10.5.5 and Java 1.6.0_07 (64 Bit). Scala code was compiled using a pre-release build of scalac 2.8.0 with option -optimize. The Kilim version used was 0.5. Numbers shown are median values of 5 consecutive measurements.

5.1 Actors

On top of its byte-code transformation, Kilim also provides an implementation of actors, which is known to outperform the Scala actor framework on a number of benchmarks. It must be noted however, that Kilim's actors lack several important features of Scala's

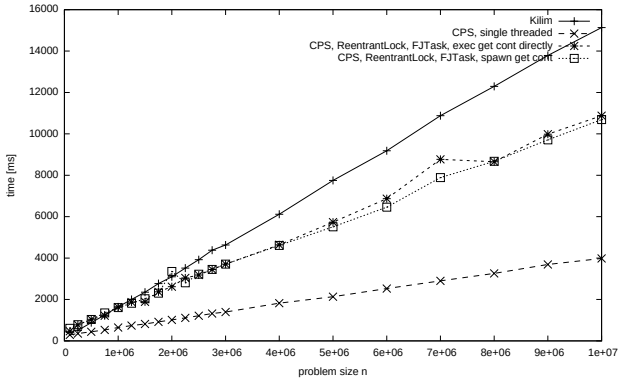

Figure 6. Ping-Pong actor benchmark. Two actors, exchanging n messages.

actors such as message pattern matching or actor linking. In our first benchmark, we compare Kilim actors to a vastly stripped-down reimplementation of Scala actors, where blocking reads on mailboxes are implemented using `shift`. The benchmark used is the "Ping-Pong" example included in both the Kilim and the Scala distribution, consisting of two actors that alternatingly exchange a fixed number of messages. The measured results are shown in Figure 6. Test runs were done using a single-threaded actor implementation without any locking and with two slightly different thread-safe implementations using standard Java locking primitives and the FJTask library (Lea 2000) as a thread pool. The thread-safe implementations differ in whether the continuation of a read operation is directly executed on the calling thread if data is available or whether it is submitted to the thread pool. The data shown in Figure 6 indicates a speedup of our thread-safe implementations over Kilim's of about 30%. The single-threaded case performs slightly less than three times faster than the thread-safe one.

5.2 Generators

For the next benchmark, the aim was to exclude any effects that might result from using multiple threads. We turn our attention to generators, which are also included in the Kilim distribution. Using generators, a possibly infinite sequence of values can be generated in *push* mode, demand-driven by one or more clients that seek to *pull* items out of the generator. An implementation using continuations is straightforward. For every data item that is to be generated, the continuation is captured and stored in a mutable variable, as is the data item. The client of the generator can access the generated value and, once it is ready to retrieve the next item, invoke the stored continuation, which will trigger generation of the next item.

We present performance measurements for two different styles of generators. One will generate values using a strictly linear, tail-recursive or iterative call pattern and thus use only constant stack space, while the other one exhibits a tree-like call pattern with logarithmic stack depth. The results are shown in Figures 7 and 8, respectively. While we can see a 30% speedup in the linear case (similar to the actors benchmark), the speedup for the tree-like case is more significant and amounts to more than a factor of seven.

5.3 Samefringe

Another direction for performance evaluation is to assess how well solutions using direct-style control operators perform in relation to other solutions. We consider the *samefringe*-problem (Gabriel 1991) and compare Scala implementations using iterators, lists, streams and delimited continuations in Figure 9. The task is to com-

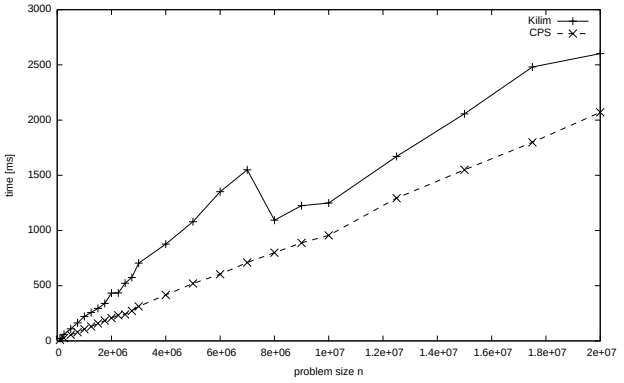

Figure 7. Generator benchmark. Generating numbers $0 \ldots n$ linearly. Call depth $O(1)$.

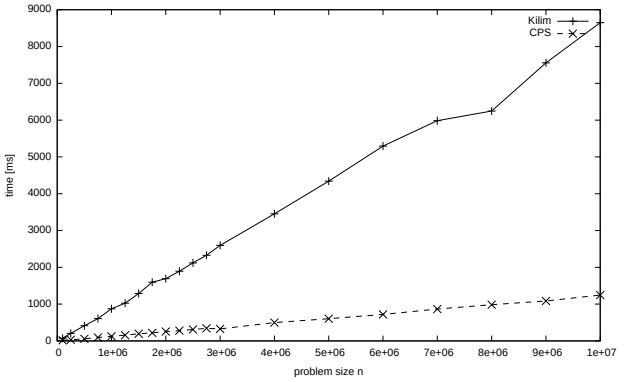

Figure 8. Generator benchmark. Generating numbers $0 \ldots n$ recursively, in a binary-search like fashion. Call depth $O(\log n)$.

pare the in-order sequences of leaves of two binary trees. For lists and streams (which are lists where the tail is computed lazily), there are two implementations each. The first one uses a modified tree traversal function to propagate a partial list downwards, to which the leaves are prepended using `cons`. The other one does not pass any partially constructed information downwards, so partial structures have to be combined using `append`. For the continuation-based case, there are also two implementations. One is almost identical to the generators described above and the other, purportedly less efficient one differs in that it will not save the captured continuations directly into a mutable variable but leave it to another `shift` in the client code to access them.

Regarding the results displayed in Figure 9, we see that streams and lists using `cons` perform best, followed within a factor of two by the faster continuations implementation and within a factor of three by the slower one. Streams and lists using `append` perform worse than continuations. That streams perform best here is no surprise, since the way they are implemented with by-name function arguments leads to a byte-code translation which is very similar to a hand-optimized CPS implementation, which avoids creating intermediate `Shift` objects as is done in the translation of the direct-style control operators.

It is instructive, though, to do another test run with more limiting memory restrictions (see Figure 10). Here we see that lists do not scale to these conditions, even though they performed well in the previous run. Streams and continuations are the only mechanisms that provide a solution in this case, since all other implementations terminate prematurely with an `OutOfMemoryError`. The continuation-based solutions still exhibit a running time that is not too far off the optimal case and still perform better than streams using `append`.

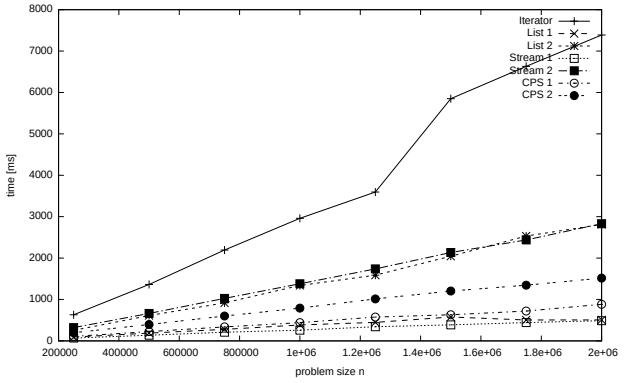

Figure 9. Samefringe benchmark. Comparing a fully balanced binary tree of size n to itself. Leaves are integers. Run with `-Xms2G -Xmx2G`.

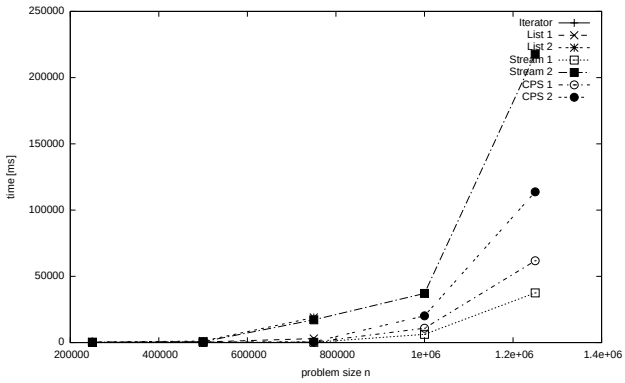

Figure 10. Samefringe benchmark. Comparing a fully balanced binary tree of size n to itself. Leaves are integers. Run with `-Xms128M -Xmx128M`. Missing data points indicate OutOfMemoryErrors. For $1.5 \cdot 10^6$ items, the tree could no longer be generated due to memory exhaustion.

6. Conclusion

In this paper, we have described an efficient way of implementing delimited continuations and the associated static control operators `shift` and `reset` under the adverse conditions of the JVM. In doing so, we employ a selective CPS transform, which is driven by a type and effect system that tracks uses of control operators. We have applied delimited continuations in several different contexts ranging from asynchronous IO to actor-based concurrency to reactive programming for user interfaces. Our experiences and performance evaluation indicate that the technique is practical and performs adequately.

Acknowledgments

We would like to thank Philipp Haller for suggesting the use of `proceed` to make Scala actors continuation aware. We also thank the reviewers for their comments.

References

Agha, Gul, and Carl Hewitt. 1987. Concurrent programming using actors. In *Object-oriented concurrent programming*, 37–53. MIT Press, Cambridge, MA, USA.

Appel, Andrew W. 1992. *Compiling with continuations*. Cambridge University Press, New York, NY, USA.

Appel, Andrew W., and Trevor Jim. 1989. Continuation-passing, closure-passing style. In *Proc. POPL'89*, 293–302.

Asai, Kenichi. 2007. On typing delimited continuations: Three new solutions to the printf problem. Tech. Rep. OCHA-IS 07-1, Department of Information Science, Ochanomizu University, Tokyo, Japan. Available from: `http://pllab.is.ocha.ac.jp/~asai/papers/`.

Asai, Kenichi, and Yukiyoshi Kameyama. 2007. Polymorphic delimited continuations. In *Proc. APLAS'07*, vol. 4807 of *LNCS*, 91–108.

Atkey, Robert. 2006. Parameterised notions of computation. In *Proc. MSFP'06*, 31–45. Electronic Workshops in Computing, British Computer Society.

Balat, Vincent, and Olivier Danvy. 1997. Strong normalization by run-time code generation. Tech. Rep. BRICS RS-97-43, Department of Computer Science, University of Aarhus, Denmark.

Biernacki, Dariusz, Olivier Danvy, and Chung-chieh Shan. 2006. On the static and dynamic extents of delimited continuations. *Science of Computer Programming* 60(3):274–297.

Clinger, William D., Anne H. Hartheimer, and Eric M. Ost. 1999. Implementation Strategies for First-Class Continuations. *Higher-Order and Symbolic Computation* 12(1):7–45.

Cooper, Gregory H., and Shriram Krishnamurthi. 2006. Embedding dynamic dataflow in a call-by-value language. In *Proc. ESOP'06*, 294–308.

Courtney, Antony, Henrik Nilsson, and John Peterson. 2003. The Yampa arcade. In *Proc. ACM SIGPLAN workshop on Haskell*, 7–18.

Danvy, Olivier. 1998. Functional unparsing. *J. Funct. Program.* 8(06):621–625.

Danvy, Olivier, and Andrzej Filinski. 1989. A Functional Abstraction of Typed Contexts. Tech. Rep., DIKU University of Copenhagen, Denmark.

———. 1990. Abstracting control. In *Proc. LFP'90*, 151–160.

———. 1992. Representing Control: A Study of the CPS Transformation. *Mathematical Structures in Computer Science* 2(4):361–391.

Danvy, Olivier, Kevin Millikin, and Lasse R. Nielsen. 2007. On one-pass CPS transformations. *J. Funct. Program.* 17(6):793–812.

Dragos, Iulian, Antonio Cunei, and Jan Vitek. 2007. Continuations in the Java Virtual Machine. In *Proc. ICOOOLPS'07*.

Dyvbig, R. Kent, Simon Peyton-Jones, and Amr Sabry. 2007. A monadic framework for delimited continuations. *J. Funct. Program.* 17(6):687–730.

Elliott, Conal, and Paul Hudak. 1997. Functional reactive animation. In *Proc. ICFP'97*.

Felleisen, Matthias. 1991. On the expressive power of programming languages. *Science of Computer Programming* 17(1-3):35–75.

Felleisen, Matthias, Mitch Wand, Daniel Friedman, and Bruce Duba. 1988. Abstract continuations: a mathematical semantics for handling full jumps. In *Proc. LFP'88*, 52–62.

Felleisen, Mattias. 1988. The theory and practice of first-class prompts. In *Proc. POPL'88*, 180–190.

Filinski, Andrzej. 1994. Representing monads. In *Proc. POPL'94*, 446–457.

———. 1999. Representing layered monads. In *Proc. POPL'99*, 175–188.

Flanagan, Cormac, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In *Proc. PLDI'93*, vol. 28(6), 237–247.

Fournet, Cédric, and Georges Gonthier. 1996. The reflexive CHAM and the join-calculus. In *Proc. POPL'96*, 372–385.

Gabriel, Richard P. 1991. The design of parallel programming languages. In *Artificial intelligence and mathematical theory of computation: papers in honor of john mccarthy*, 91–108. Academic Press Professional, San Diego, CA, USA.

Gasbichler, Martin, and Michael Sperber. 2002. Final shift for call/cc:: direct implementation of shift and reset. In *Proc. ICFP'02*, 271–282.

Haller, Philipp, and Martin Odersky. 2006. Event-based programming without inversion of control. In *Proc. JMLC'06*, vol. 4228 of *LNCS*, 4–22.

———. 2009. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci* 410(2-3):202–220.

Kameyama, Yukiyoshi, and Takuo Yonezawa. 2008. Typed dynamic control operators for delimited continuations. In *Proc. FLOPS'08*, vol. 4989 of *LNCS*, 239–254.

Kiselyov, Oleg. 2005. How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible. Tech. Rep. TR611, Department of Computer Science, Indiana University.

———. 2007. Genuine shift/reset in haskell98. Announcement and explanations posted on the Haskell mailing list on 12/12/2007. Implementation available from: `http://okmij.org/ftp/Haskell/ShiftResetGenuine.hs`.

Kiselyov, Oleg, Chung-chieh Shan, and Amr Sabry. 2006. Delimited dynamic binding. In *Proc. ICFP'06*, 26–37.

Lea, Doug. 2000. A Java fork/join framework. In *Proc. ACM Java Grande*, 36–43.

Moors, Adriaan, Frank Piessens, and Martin Odersky. 2008. Generics of a higher kind. In *Proc. OOPSLA'08*, 423–438.

Nielsen, Anders Bach. 2008. Scala compiler phase and plug-in initialization. Available from: `http://lampsvn.epfl.ch/svn-repos/scala/lamp-sip/compiler-phase-init/sip-00002.xhtml`.

Nielsen, Lasse R. 2001. A selective CPS transformation. Tech. Rep. RS-01-30, BRICS, Department of Computer Science, Aarhus University.

Odersky, Martin. 2000. Functional Nets. In *Proc. European Symposium on Programming Languages and Systems*, 1–25.

Pettyjohn, Greg, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. 2005. Continuations from generalized stack inspection. *SIGPLAN Not.* 40(9):216–227.

Rompf, Tiark. 2007. Design and implementation of a programming language for concurrent interactive systems. Master's thesis, Institute of Software Technology and Programming Languages, University of Lübeck, Germany. Available from: `http://vodka.nachtlicht-media.de/docs.html`.

Rose, John. 2008. JSR 292: Supporting dynamically typed languages on the Java platform. `http://jcp.org/en/jsr/detail?id=292`.

Shan, Chung-chieh. 2004. Shift to control. In *Proc. ACM SIGPLAN workshop on Scheme and functional programming*, 99–107.

———. 2007. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation* 20(4):371–401.

Srinivasan, Sriram. 2006. A thread of one's own. In *New horizons in compilers workshop, hipc, bangalore*.

Srinivasan, Sriram, and Alan Mycroft. 2008. Kilim: Isolation-typed actors for Java. In *Proc. ECOOP'08*, 104–128.

Strachey, Christopher, and Christopher P. Wadsworth. 2000. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation* 13(1):135–152.

Thielecke, Hayo. 2003. From control effects to typed continuation passing. In *Proc. POPL'03*, 139–149.