Reasoning about Continuations with Control Effects

Pierre Jouvelot [1,2]
David K. Gifford [2]

Abstract

We present a new static analysis method for first-class continuations that uses an *effect system* to classify the control domain behavior of expressions in a typed polymorphic language. We introduce two new control effects, `goto` and `comefrom`, that describe the control flow properties of expressions. An expression that does not have a `goto` effect is said to be *continuation following* because it will always call its passed return continuation. An expression that does not have a `comefrom` effect is said to be *continuation discarding* because it will never preserve its return continuation for later use. Unobservable control effects can be *masked* by the effect system. Control effect soundness theorems guarantee that the effects computed statically by the effect system are a conservative approximation of the dynamic behavior of an expression.

The effect system that we describe performs certain kinds of control flow analysis that were not previously feasible. We discuss how this analysis can enable a variety of compiler optimizations, including parallel expression scheduling in the presence of complex control structures, and stack allocation of continuations. The effect system we describe has been implemented as an extension to the FX-87 programming language.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-83-K-0125.

[1]CAI, Ecole des Mines, 60 bvd Saint-Michel, 75272, PARIS, France (E-mail: `JOUVELOT@FREMP11.bitnet`)

[2]LCS, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139, USA (E-mail: `GIFFORD@BROKAW.LCS.MIT.EDU`)

Categories and Subject Descriptions: D.1.3 [**Programming Techniques**] – Concurrent Programming: *Effect systems*; D.1.m [**Programming Techniques**] – Miscellaneous: *First-class continuations*; D.3.1 [**Programming Languages**] – Formal Definitions and Theory; D.3.3 [**Programming Languages**] – Language Constructs: *Control structures, effect systems*; D.3.4 [**Programming Languages**] – Processors: *Compilers, optimization.*

General Terms: Languages, Theory, Verification.

Additional Key Words and Phrases: effect systems, type systems, control effects, effect masking, control flow analysis, FX-87.

1 Introduction

First-class continuations add a great deal of expressive power to a programming language as they permit the implementation of a wide variety of control structures, including jumps, error-handlers, and coroutines [F87]. With this power comes substantial semantic [MR88] and implementation [CHO88] complexities. Thus it would be very useful to be able to precisely identify which expressions in a program use first-class continuations and in what manner.

We present a new static method for control flow analysis that performs certain kinds of analysis that were not previously feasible. Specifically, we have developed the first static method of determining which expressions may not exhibit sequential control flow in a programming language with first-class continuations.

Our static analysis technique is based on the use of an *effect system* [LG88] to classify the possible control domain behavior of expressions. An effect system is based on a kinded type system for the second-order lambda calculus [M79]. Kinds are the "types" of *descriptions* which include types and effects. Our type and effect system has three base kinds: *types*, which describe the value that an expression may return; *effects*, which describe the side-effects that an expression may have; and

regions, which are used to describe where side-effects may occur. An expression that does not have an observable effect is said to be `pure`. Expressions that are `pure` are referentially transparent.

Types, effects and regions are closely interrelated; in particular, a function type incorporates a *latent effect* that describes the side-effects that the function may perform when it is applied, and a reference type incorporates a region that describes where the reference is allocated. The kind system is used to verify the well-formedness of descriptions; the type and effect system is used to verify the well-formedness of expressions.

We can use an effect system for control flow analysis by introducing two types of control effects, `goto` and `comefrom`, that describe the control flow properties of expressions. An expression that does not have a `goto` effect is said to be *continuation following* because it will always call its return continuation in the usual way. An expression that does not have a `comefrom` effect is said to be *continuation discarding* because it will never preserve its return continuation for later use. This "double negation" style of definitions is necessary when one wants to express conservative approximations of run-time program behaviors.

Unobservable control effects can be *masked* by the effect system. Our masking rule applies to expressions that are externally well-behaved, even if they use continuations internally. Control effect soundness theorems guarantee that the effects computed statically by the effect system are a conservative approximation of the dynamic behavior of an expression.

We show how our effect system can be used with the procedure `call-with-current-continuation` inspired from [R86], hereafter noted `cwcc`. This procedure allows first-class access to the current continuation. Simpler control structures based on labels and jumps can be treated in a similar way.

Control effects are useful to the programmer, the language designer and the compiler writer:

- Control effects let the programmer specify, in machine-verifiable form, the expected run-time control behavior of a given program, thus increasing documentation, modularity and maintainability of programs. Control effects also provide a programmer with a new framework in which to reason about languages with first-class continuations. Moreover, when unobservable control effects are masked, a programmer knows that an expression will be well-behaved.

- Control effects let the language designer limit the use of continuations to simplify the semantics of the language. For instance, by saying that top-level definitions are not allowed to have `comefrom` or `goto` effects, the problem of a variable "redefinition" by a "return" inside its `define` form is avoided. In the same manner, mutations that are performed by taking advantage of the implementation of recursive definitions by `letrec` [B89] can be prohibited.

- Control effects let the compiler writer perform safe optimizations in the presence of first-class continuations. For instance, if a given `cwcc` expression has a masked control effect, then the internal continuation will be used only as a "downward funarg" [S78] and thus the expression's continuation structure (control frames) can be stack allocated.

Control effects also allow sequential semantics to be preserved in the presence of both first-class continuations and automatic compile-time detection of parallelism. When compiling for a parallel target machine, the compiler can guarantee sequential semantics (which is intimately related to the notion of continuations, which represent the state of a sequential evaluation) by considering that control effects interfere with all effects.

In the remainder of this paper we describe the kernel language KFX of FX-87 (Section 2), integrate control effects into KFX (Section 3), state two control effect soundness theorems (Section 4), give precise conditions when it is possible to mask unobservable control effects (Section 5), survey related work (Section 6), and summarize our results (Section 7).

2 KFX - A Kernel Language for FX-87

For pedagogical purposes we will study control effects in the context of KFX, the kernel language of FX-87. FX-87 [GJLS87][LG88] is a polymorphic typed language that allows side-effects and first-class functions. Its syntax and most of its standard operations are strongly inspired by Scheme [R86] which will be used in most of our examples. The language KFX has the following Kind, Description (Region, Effect and Type) and Expression domains (where `I` is the domain of identifiers):

```
K  ::=  region
        effect
        type

R  ::=  I
        @I        region constant
```

```
F  ::=  I
        pure                  no effect
        (write R)             write on R
        (read R)              read on R
        (alloc R)             allocation on R
        (maxeff F0 F1)        combination

T  ::=  I
        (subr F (T) T)        function
        (poly (I K) T)        polymorphic type
        (ref T R)             reference to T in
                                  region R
D  ::=  R
        F
        T

E  ::=  I
        (lambda (I T) E)      lambda
                                  abstraction
        (E0 E1)               application
        (plambda (I K) E)     polymorphic
                                  abstraction
        (proj E D)            projection
        (new R T E)           allocation
        (get E)               dereference
        (set E0 E1)           mutation
```

In the effect domain, a `pure` expression is referentially transparent. Impure expressions can `write`, `read` or allocate (`alloc`) memory in regions which denote sets of memory locations. Combined effects are introduced by `maxeff`.

In the type domain, the `subr` type of a function encodes its latent effect `F`, the type of its argument and its return type. `Poly` represents the type of polymorphic values abstracted over kind `K`. `(Ref T R)` is the type of a reference in the region `R` to a value of type `T`.

In the expression domain, just as `lambda` abstracts `E` over the ordinary variable `I` of type `T`, `plambda` abstracts `E` over the description variable `I` of kind `K` to yield a polymorphic value. A polymorphic value is instantiated with the `proj` construct. `New` allocates (i.e., has an `alloc` latent effect) a reference in region `R` to the value of `E` of type `T`, `get` reads and returns the value stored into the reference `E` and `set` writes `E1` in the reference `E0`. As an example, we give below the code of the polymorphic `twice` function that applies the function `f` twice on its argument `x`:

```
(plambda (t type)
  (plambda (e effect)
    (lambda (f (subr e (t) t))
      (lambda (x t)
        (f (f x))))))
```

Note that `twice` is abstracted over the type `t` of the argument of `f` and its latent effect `e`. The type of `twice` is:

```
twice : (poly (t type)
          (poly (e effect)
            (subr pure
                  ((subr e (t) t))
                  (subr e (t) t))))
```

The type and effect rules for application, abstraction, polymorphic abstraction[1] and projection follow. Just as ":" is used to denote the "type of" relation, "!" is used to denote the "effect of" relation. T is the type and kind assignment function that maps variables to their type or kind.

$$\frac{\begin{array}{lcll}\langle \texttt{E0}, T\rangle & : & \texttt{(subr}\ e\ \texttt{(}t_1\texttt{)}\ t_2\texttt{)} & ! \quad e_0 \\ \langle \texttt{E1}, T\rangle & : & t_1 & ! \quad e_1\end{array}}{\langle \texttt{(E0 E1)}, T\rangle : t_2 \ ! \ \texttt{(maxeff}\ e\ \texttt{(maxeff}\ e_0\ e_1\texttt{))}}$$

$$\frac{\langle \texttt{E}, T[t_1/\texttt{I}]\rangle : t_2 \ ! \ e}{\langle \texttt{(lambda (I}\ t_1\texttt{) E)}, T\rangle : \texttt{(subr}\ e\ \texttt{(}t_1\texttt{)}\ t_2\texttt{)} \ ! \ \texttt{pure}}$$

$$\frac{\langle \texttt{E}, T[k/\texttt{I}]\rangle : t \ ! \ \texttt{pure}}{\langle \texttt{(plambda (I}\ k\texttt{) E)}, T\rangle : \texttt{(poly (I}\ k\texttt{)}\ t\texttt{)} \ ! \ \texttt{pure}}$$

$$\frac{\langle \texttt{E}, T\rangle : \texttt{(poly (I}\ k\texttt{)}\ t\texttt{)} \ ! \ e}{\langle \texttt{(proj E}\ k'\texttt{)}, T\rangle : t[k'/\texttt{I}] \ ! \ e}$$

where $(f[y/x])z$ is y if z is x and fz otherwise (the same notation on types is used to denote syntactic substitution). The remaining effect and type rules of KFX are presented in [LG88].

The standard semantics [S86] of KFX is defined on type-erased KFX expressions. A program is type-erased by reducing `plambda` and `proj` expressions to their embedded expressions and eliminating all type information (see [L87] for a formal definition of type erasure).

The domain equations for locations, basic values, closures, values, environment, stores, results and continuations in our standard semantics are as follows:

l : Loc
b : $Basic$
p : $Clo = Val \to Cont \to Store \to Answer$
v : $Val = Basic + Clo + Loc$
u : $Env = \texttt{I} \to Val$
s : $Store = Loc \to Val$
a : $Answer = Val$
k : $Cont = Val \to Store \to Answer$

where $+$ means disjoint sum and $\to$ is right-associative.

[1] We suppose alpha-renaming to avoid name capture.

The definition of Eval, of type E → *Env* → *Cont* → *Store* → *Answer*, is fairly simple [S86]:

$$\begin{aligned}
\mathcal{E}[\![\ \texttt{I}\ ]\!]uk &= k(u[\![\ \texttt{I}\ ]\!]) \\
\mathcal{E}[\![\ \texttt{(lambda (I) E)}\ ]\!]uk &= k(\lambda e.\mathcal{E}[\![\ \texttt{E}\ ]\!]u[e/\texttt{I}]) \\
\mathcal{E}[\![\ \texttt{(E0 E1)}\ ]\!]uk &= \mathcal{E}[\![\ \texttt{E0}\ ]\!]u(\lambda e_0.\\
&\qquad \mathcal{E}[\![\ \texttt{E1}\ ]\!]u(\lambda e_1.\\
&\qquad\quad e_0 e_1 k)) \\
\mathcal{E}[\![\ \texttt{(new E)}\ ]\!]uk &= \mathcal{E}[\![\ \texttt{E}\ ]\!]u(\lambda es.kl(s[e/l])) \\
&\qquad \text{where } l = newLoc() \\
\mathcal{E}[\![\ \texttt{(get E)}\ ]\!]uk &= \mathcal{E}[\![\ \texttt{E}\ ]\!]u(\lambda es.k(se)s) \\
\mathcal{E}[\![\ \texttt{(set E0 E1)}\ ]\!]uk &= \mathcal{E}[\![\ \texttt{E0}\ ]\!]u(\lambda e_0.\\
&\qquad \mathcal{E}[\![\ \texttt{E1}\ ]\!]u(\lambda e_1 s_1.\\
&\qquad\quad ke_1(s_1[e_1/e_0])))
\end{aligned}$$

where *newLoc* is a function that allocates fresh memory locations. Note that, contrarily to the Scheme `set!` special form, the function `set` evaluates its first argument, which is a reference. For clarity, we omitted injection and projection operations in sum domains.

3 Control Effects Describe Potential Behavior

The material of the previous section is a simplified presentation of the effect system that is the basis of the FX-87 programming language [GJLS87]. This static system was mainly designed to cleanly deal with *memory* side-effects within a typed polymorphic language which allows first-class functions. We show below how this framework can be easily extended to deal with the a-priori unrelated *control* side-effects that are caused by first-class continuations. This shows the power and flexibility of an effect system.

We introduce first-class continuations into KFX with the Scheme function `cwcc`. The function `cwcc` passes a procedural representation of its return continuation to its argument, which must be a one-argument function. In order to add `cwcc` to KFX we need merely to bind `cwcc` to its value in the initial environment u. So, we have :

$$u[\![\ \texttt{cwcc}\ ]\!] = \lambda ek.e(\lambda e'k'.ke')k$$

The `cwcc` function gives the programmer access to continuations. As first-class values, these continuations can subsequently be stored into global variables or returned, allowing a given expression to be evaluated more than once (to return a result more than once) or to escape from its current continuation (to discard its current continuation by applying another one).

The `cwcc` function cannot be treated as pure. The `comefrom` effect corresponds to the capture of a continuation and it is introduced by providing `cwcc` with a `(comefrom r)` latent effect, where `r` is a region. The region restricts the domain of influence of a given continuation-capturing expression (see below). The following Scheme program shows that a compiler cannot common subexpression eliminate calls to `cwcc` since calling `f1` would perform `horrible-effects` on the store; these effects would not occur if `f2` were called. Thus the `cwcc` calls cannot be `pure`.

```
(let* ((f1 (cwcc (lambda (x) x)))
       (x (horrible-effects))
       (f2 (cwcc (lambda (x) x))))
  ...)
```

The `goto` effect corresponds to the "escaping" call of a continuation and is introduced by providing a continuation with a `(goto r)` latent effect, where `r` is the region used in the `cwcc` expression that created the continuation. The following Scheme program shows that any effect can be exercised by the evaluation of a continuation:

```
(let ((x (cwcc (lambda (f)
                 (cwcc (lambda (g) (f g)))
                 (h)
                  f))))
  (horrible-effects)
  ...
  (x 0)
  ...)
```

In this example, between the evaluations of `(f g)` and `(h)`, `(horrible-effects)` will be executed; thus these expressions cannot be reordered. The effects of `(horrible-effects)` will also be performed after any call to `x`, since it is bound to the continuation that binds the result of the call of `cwcc` to `x`.

In summary, the KFX type and effect rules can be used to compute control effects by assigning `cwcc` the following type[2]:

```
cwcc: (poly (r region)
        (poly (t type)
          (poly (e effect)
            (subr (maxeff (comefrom r) e)
                  ((subr e
                         ((subr (goto r)
                                (t)
                                void))
                         t))
                  t))))
```

[2]Without loss of generality, we assume that the anti-aliasing rule of FX-87 [GJLS87] is eliminated.

The function `cwcc` is abstracted over the region `r` in which the continuation is located, the type `t` of the eventual result and the latent effect `e` of the function passed as argument. The latent effect of `cwcc` denotes the combined effects of calling its argument and capturing a continuation in `r`. The argument to `cwcc` is a function that expects as argument the captured continuation seen as a function with a `(goto r)` latent effect. The type `void` indicates that a call to the continuation will never return to its caller.

The control effects of an expression can be masked if the effects cannot be observed outside of the expression. For instance, if we follow the typing rules of KFX, then we deduce that the following expression

```
(+ (cwcc (lambda (f) (f 0))) 1)
```

has both `(goto r)` and `(comefrom r)` effects for some region `r`. However, it is easy to prove that this expression is well-behaved; it will call its return continuation (it is *continuation following*) and will not preserve its continuation (it is *continuation discarding*). Its control effects can thus be masked as we discuss further in section 5; in consequence, the continuation `f` will not need to be heap allocated, thus improving run-time performance and saving on memory consumption.

4 Soundness of comefrom and goto effects

We now need to show that the `comefrom` and `goto` effects we compute for an expression are a conservative estimate of the expression's dynamic behavior. Without loss of generality, we restrict our soundness proofs to terminating expressions.

Definition. An expression `E` is *continuation following* iff, for every environment u and store s, there exist two continuations k_1 and k_2 such that

$$\mathcal{E}[\![\ \texttt{E}\ ]\!]uk_1s \neq \mathcal{E}[\![\ \texttt{E}\ ]\!]uk_2s$$

The definition of equality we use is extensional. The evaluation of a continuation-following expression ends by calling the continuation it is passed and never escapes via some continuation available in the environment or store.

Theorem (Goto Soundness). *If there does not exist a region* `r` *such that* `E` *has effect* `(goto r)`*, then* `E` *is continuation following.*

Proof sketch. By induction on the domain of expressions. Every construct of KFX is continuation following (a look at the semantic definition shows that every equation of $\mathcal{E}$ ends by calling the continuation it is passed), except the application `(E0 E1)`. There are two ways we can generate a function in KFX:

- if e_0 is obtained by evaluating a lambda expression, then it is continuation following if its body is,
- otherwise e_0 comes from a continuation created by a `cwcc` expression, and then the expression isn't continuation following. But, this continuation has latent effect `(goto r)` where `r` is the region that `cwcc` has been projected onto. Therefore, the application will have effect `(goto r)`. □

With this theorem, we know that the `goto` effect information collected by the KFX type system is a conservative (i.e. safe) approximation of the run-time behavior of a program.

We now prove the soundness of `comefrom` effects. This aspect is a bit more complex, so we need some preliminary definitions.

Definition. A *continuation collecting semantics* of a programming language L is a non-standard semantics of L in which every evaluation of an expression `E` returns an ordered pair made of the standard value of `E` and the set of intermediate continuations captured while evaluating `E` (i.e. continuations made available by calls to `cwcc`).

Definition. $\mathcal{E}_c$ is the continuation collecting semantics of $\mathcal{E}$.

The precise definition of $\mathcal{E}_c$ is an extension of $\mathcal{E}$, where the set of continuations is passed along the computation and updated by `cwcc`. For every function F used in the semantics of KFX, there is an equivalent one in the continuation collecting semantics of KFX; we will note it F_c. Note that the continuations k_c successively take the value to be passed, the set of (potentially available) continuations and the current store. For instance, here are the equations defining $\mathcal{E}_c$ for identifier, lambda expression and application (j denotes the current set of intermediate continuations):

$$\begin{aligned}
\mathcal{E}_c[\![\ \texttt{I}\ ]\!]u_cjk_c &= k_c(u_c[\![\ \texttt{I}\ ]\!])j \\
\mathcal{E}_c[\![\ \texttt{(lambda (I) E)}\ ]\!]u_cjk_c &= k_c(\lambda ej.\\
&\qquad \mathcal{E}_c[\![\ \texttt{E}\ ]\!]u_c[e/\texttt{I}]j)j \\
\mathcal{E}_c[\![\ \texttt{(E0 E1)}\ ]\!]u_cjk_c &= \mathcal{E}_c[\![\ \texttt{E0}\ ]\!]u_cj(\lambda e_0j_0.\\
&\qquad \mathcal{E}_c[\![\ \texttt{E1}\ ]\!]u_cj_0(\lambda e_1j_1.\\
&\qquad\quad e_0e_1j_1k_c))
\end{aligned}$$

and the definition for `cwcc`:

$$u_c\ \texttt{cwcc} = \lambda ejk_c.e(\lambda e'j'k'_c.k_ce'j')(\{k_c\} \cup j)k_c$$

Definition. An *expression context* `Ec` is an expression with a "hole" in it, identified by `[]`. It can be filled with `E` by writing `Ec[E]`.

Definition. An expression context `Ec` is *well-behaved* iff for every environment u_c, store s_c and continuation k_c, there exist an expression `E` and a value v such that

$$\mathcal{E}_c[\![\ \texttt{Ec[E]}\ ]\!]u_c\{\}k_cs_c = v, \{\}$$

where x, y is the ordered pair made of x and y and $\{\}$ the empty set. A well-behaved expression context doesn't capture any continuation on its own.

Definition. An expression `E` is *continuation discarding* iff for every well-behaved expression context `Ec`, environment u_c, store s_c, continuation k_c, value v and list of continuations j:

$$\mathcal{E}_c[\![\ \texttt{Ec[E]}\ ]\!]u_c\{\}k_cs_c = v, j \implies v \notin j$$

The idea behind this definition is that it isn't possible, from the evaluation of `E`, to recover some continuation caught in some global variable or returned by `E`. For instance, the following expression :

```
(begin (cwcc (lambda (f)
               (set x (lambda () f))))
       (h))
```

isn't continuation discarding, since the well-behaved context `(begin [] ((get x)))` returns a caught continuation if we replace the hole by the previous expression.

Theorem (Comefrom Soundness). *If there does not exist a region* `r` *such that* `E` *has effect* `(comefrom r)`*, then E is continuation discarding.*

Proof sketch. The proof is by induction on the equations of $\mathcal{E}_c$. The only way the set of caught continuations can be extended is by a call to `cwcc`, which implies, following KFX typing rules, that there is a region `r` such that this call has a `(comefrom r)` effect. □

With this theorem, we know that the `comefrom` effect information collected by the KFX type system is a conservative (i.e. safe) approximation of the run-time behavior of a program.

5 Control Effect Masking

FX-87 effect masking detects and eliminates effects of an expression that are not observable (e.g., mutation of locally allocated references) under the following conditions:

- if no free variable uses the region `r` in its type, then read and write effects on `r` can be masked,
- if `r` does not appear in the type of the result of the expression, then alloc effects on `r` can be masked.

Effect masking can be easily extended to deal with control effects. Control effect masking soundness a direct extension of FX-87 type soundness and effect masking soundness [GJLS87]. We use the following lemma:

Lemma. *When evaluating* `E` *in the environment* u*, continuation* k *and store* s*, the value passed to* k *is determined by the values of the free variables and the storage locations accessible via these free variables.*

Proof sketch. By induction on $\mathcal{E}$ equations with the use of the "location invariance" property of [L87] which expresses that the choice of bound locations does not influence the final result of the evaluation. □

Theorem. *A* `(goto r)` *effect of an expression* `E` *can be masked if* `E` *does not import variables or return values in the type of which* `r` *appears.*

Proof sketch: We have to show that if the preceding condition is verified, then `E` is continuation following, i.e. that for every u and s there exist k_1 and k_2 such that $\mathcal{E}[\![\ \texttt{E}\ ]\!]uk_1s \neq \mathcal{E}[\![\ \texttt{E}\ ]\!]uk_2s$. By the previous lemma, the value of `E` passed to k_1 and k_2 is defined only by the values of the free variables of `E`, which appear in u, and accessible (i.e. in the returned value) locations, in s. If `r` does not appear in the type of any of these values, then, by application of FX-87 type soundness, no `(goto r)` effect can be performed by an external (i.e. not defined in `E`) continuation. By goto effect soundness, `E` is then continuation-following, and choosing $k_1 = (\lambda es.0)$ and $k_2 = (\lambda es.1)$ will suffice. □

Theorem. *A* `(comefrom r)` *effect of an expression* `E` *can be masked if* `E` *does not import variables or return values which have* `r` *in their type.*

Proof sketch. In the same way, we have to show that if the preceding condition is verified, then `E` is continuation discarding. We want to prove that for every `Ec`, u_c, s_c and k_c, v is not a member of j where v and j are as in the definition of "continuation discarding". As before, `Ec[E]` can only refer to free variables of `E` or its returned value. If `r` does not appear in the type of any of these, then, by application of FX-87 type soundness, no continuation effectively caught by `cwcc` forms inside of `E` projected on `r` can be returned by `Ec[E]`. □

Not surprisingly these control effect masking rules are very similar to the ones already in FX-87 for memory effects. One can view a call to `cwcc` as allocating some data in the heap region to store the current stack and a call to a continuation as reading a stored stack and writing to the current stack. This analogy is not quite right since the `goto` effect requires a more stringent rule as seen above. This can be demonstrated by the following (admittedly contrived) Scheme program which shows an expression with a `goto` effect that cannot be masked, even though this expression has no free variables:

```
(let ((x (cwcc (lambda (f)
                 (let ((y (cons f f)))
                   (cwcc (lambda (g)
                           (set-cdr! y g)
                           (f y)))
                   ((car y) y))))))
   (cwcc (lambda (h)
           (set-car! x h)
           ((cdr x) x))))
```

In this example, by calling `f`, we first bind `x` to the pair `y` that contains `f` and the continuation `g` that precedes the evaluation of `((car y) y)`. The first element of `x` is mutated with `h` that corresponds to the rest of the program and `x`'s binding expression is reentered at the point where `((car y) y)` is evaluated; this evaluation is not continuation-following. Although there are no free variables in the expression `x` is bound to, the `goto` effect of the call `((car y) y)` cannot be masked; first-class continuations allow the programmer to temporarily export and mutate local locations, thus requiring the supplementary check on the regions appearing in return type (which is also, by construction, the type of values passed to the captured continuation).

The major practical benefit of control effect masking is the possibility of compile-time detection of externally well-behaved expressions that internally use complex control structures. This allows a programmer to formally reason about his program as if he were not using first-class continuations, thus easing program correctness proofs. In addition, this allows compile-time optimizations such as parallel evaluation of expressions that have internal control effects or recognition of stack-allocatable continuations, in which case `cwcc` can be implemented with a more efficient *catch-throw* mechanism [M74].

When an expression has a masked control effect, it may use full-fledged continuations internally. If control effects are kept at run time, control effects can be used to dynamically optimize these internal continuations as well. The only control frames that have to be dumped into the heap when a continuation in region `r` is captured are the ones that are flagged with the same `(comefrom r)` effect. These are precisely the control frames that correspond to the dynamic extent of the captured continuation up to the point where the control effect is masked. We know that the control frames that appear after this point won't be needed. Note that this optimization is not of utmost importance in a system that uses a stack/heap strategy for the implementation of continuations [CHO88]. It can however be useful in an implementation that supports parallel evaluation of expressions since it avoids stack sharing between different processes.

6 Related Work

A related approach to our compile-time analysis of programs uses the notion of "abstract interpretation" [CC77]. Results in the area of functional languages [AH87] with higher-order functions and polymorphism cannot yet equal the outcome of our type and effect system. For instance, the only analysis of continuations we found [H87] does not address first-class continuations.

[F88a] proposed the λ-v-CS formal system to deal with higher-order functions in the presence of imperative constructs. Continuations can be described in such a framework and properties about them can be proved. However, these proofs are not currently automated and therefore cannot be used in compiling systems.

Thus, the compile time analysis of programming languages with first-class continuations appears to be new. Most of the literature dealing with continuations generally focuses on their run-time characteristics and applications to various problems like coroutines [HFW86], multiprocessing [W80] or backtracking [SM72].

The possibility of stack-allocation of continuation structure, which is a major asset of the control effect masking we presented above, is often presented, e.g. in [HF87], but with an "interpreter view" of the problem. The detection is done at run time (e.g., by checking that a continuation is not used in a way incompatible with its implementation). Our system performs a safe approximation of this optimization at compile time.

Control effect masking allows the run-time system to limit the amount of control information that has to be dumped in the heap when a continuation is captured; the "prompts" proposed in [F88b] or the "shift/reset" of [DF89] offer the same kind of facility, but they have to be introduced by the programmer.

We showed elsewhere [JG89a, JG89b] how effect systems can be also used to describe *communication* side-effects arising when communicating parallel processes are added to a language that uses and effect systems. Communication effects describe transmission operations

on shared channels between processes. Control and communication effects show the power of an effect system to statically describe run-time behavior of programs.

7 Conclusion

We presented a new static system that enables the use of first-class continuations in a typed polymorphic language. Simpler control structures, for example those based on gotos and labels, can be straightforwardly treated by the same framework.

Our new effect system introduces two new effect constructors, called `goto` and `comefrom`. Every expression that does not have a `goto` (`comefrom`) effect is continuation following (discarding). Expressions that do not have control effects are well-behaved in the sense that they observe normal sequential execution semantics. We showed how a precise definition of these notions can be given with respect to the FX-87 standard and continuation collecting semantics. We also proved two effect soundness theorems.

The control effect masking rules enable us to detect expressions that are externally well-behaved even though they use continuations internally. Such expressions can be optimized, e.g. continuations can be stack allocated at compile time. Well-behaved expressions can be scheduled to execute in parallel subject to their other effects.

We have also found it relatively easy to implement our effect system as an extension to FX-87. As discussed above, we only needed to add three constants to our existing store effect system in order to handle the complications introduced by `cwcc`.

Acknowledgments

We want to express our gratitude to Olivier Danvy for his insightful comments on the optimization of continuations and to one of the anonymous referees for encouraging us to look for reference [C73] "if only for hack value".

References

[**AH87**] Abramsky, C., and Hankin, C. T. Eds. *Abstract Interpretation for Declarative Languages.* J. Wiley and Sons, 1987.

[**B89**] Bawden, A. Submission to the Scheme electronic mailing list `scheme@mc.lcs.mit.edu`. March 2, 1989.

[**C73**] Clark, R. L. A Linguistic Contribution to GOTO-Less Programming. *Datamation*, December 1973, 62-63. Reprinted in *Communications of the ACM 4*, 27 (1984), 349-350.

[**CC77**] Cousot, P., and Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction and Approximation of Fixed Points. In *Proceedings of the 4th Annual ACM Conference on Principles of Programming Languages.* ACM, New York, 1977, pp. 238-252.

[**CHO88**] Clinger, W. D., Hartheimer, A. H., and Ost, E. O. Implementation Strategies for Continuations. in *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming.* ACM, New York, 1988, pp. 124-131.

[**DF89**] Danvy, O., and Filinski, A. *A Functional Abstraction of Typed Contexts.* DIKU Report 89/5, University of Copenhagen, 1989.

[**F87**] Friedman, D. P., and Haynes, C. T. Constraining Control. In *Proceedings of the 12th Annual ACM Conference on Principles of Programming Languages.* ACM, New York, 1985, pp. 245-254.

[**F88a**] Felleisen, M. λ-v-CS: An Extended λ-Calculus for Scheme. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming.* ACM, New York, 1988, pp. 72-85.

[**F88b**] Felleisen, M. The Theory and Practice of First-Class Prompts. In *Proceedings of the 15th Annual ACM Conference on Principles of Programming Languages.* ACM, New York, 1988, pp. 180-190.

[**GJLS87**] Gifford, D. K., Jouvelot, P., Lucassen, J. M., and Sheldon, M. A. *The FX-87 Reference Manual.* MIT/LCS/TR-407, 1987.

[**H87**] Hughes, J. Strictness Analysis by Abstract Interpretation of Continuations. In [AH87].

[**HF87**] Haynes, C. T., and Friedman, D. P. Embedding Continuations in Procedural Objects. *ACM*

Trans. on Prog. Lang. and Syst. 9, 4 (1987), 582-598.

[**HFW86**] Haynes, C. T., Friedman, D. P., and Wand, M. Obtaining Coroutines with Continuations. *Comput. Lang. 11*, 3/4 (1986), 143-153.

[**JG88**] Jouvelot, P. and Gifford, D. K. The FX-87 Interpreter. In *Proceedings of the 2nd IEEE International Conference on Computer Languages*. IEEE, New York, 1988, pp. 65-72.

[**JG89a**] Jouvelot, P. and Gifford, D. K. Parallel Functional Programming: The FX-87 Project. To appear in *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. North Holland, Amsterdam, 1989.

[**JG89b**] Jouvelot, P. and Gifford, D. K. *Communication Effects for Message-Based Concurrency*. MIT/LCS/TM-386, 1989.

[**L87**] Lucassen, J. M. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD Dissertation, MIT/LCS/TR-408, 1987.

[**LG88**] Lucassen, J. M., and Gifford, D. K. Polymorphic Effect Systems. In *Proceedings of the 15th Annual ACM Conference on Principles of Programming Languages*. ACM, New York, 1988, pp. 47-57.

[**M74**] Moon, D. *MacLISP Reference Manual, Revision 0*. MIT Project MAC, 1974.

[**M79**] McCracken, N. J. *An Investigation of a Programming Language with a Polymorphic Type Structure*. PhD Dissertation, Syracuse University, 1979.

[**MR88**] Meyer, A. R., and Riecke, J. Continuations May Be Unreasonable. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*. ACM, New York, 1988, pp. 63-71.

[**R86**] Rees, J. A., and Clinger, W. Eds. *The Revised^3 Report on the Algorithmic Language Scheme*. MIT/AI Memo 848a, 1986.

[**S78**] Steele, G. L., and Sussman, G. J. *The Art of the Interpreter or, The Modularity Complex (Parts Zero, One and Two)*. MIT/AI Memo 453, 1978.

[**S86**] Schmidt, D. A. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[**SM72**] Sussman, G. J., and McDermott, D. From PLANNER to CONNIVER – A Genetic Approach. In *Proceedings of the Fall Joint Computer Conference*. AFIPS Press, Reston, 1972, pp 1171-1179.

[**W80**] Wand, M. Continuation-based Multiprocessing. In *Proceedings of the 1980 ACM Conference on Lisp and Functional Programming*. ACM, New York, 1980, pp. 19-28.