Programming Language Engineering
Master of Computer Science

Faculty of Science and Bio-Engineering Sciences
Vrije Universiteit Brussel

Section 9: Advanced Features

Theo D'Hondt
Software Languages Lab

Advanced Features

* Proper Tail Recursion

version 10

* First-class Continuations

version 11

Proper Tail Recursion

```
(begin
  (define (fibonacci p q r)
     (if (> p 1)
        (fibonacci (- p 1) r (+ q r))
        r))
  (fibonacci 15 1 1))
```

stack size

40

30

20

10

0

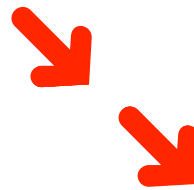
version 9

version 10

Proper Tail Recursion (cont'd)

```
static EXP_type evaluate_expression(EXP_type Expression)
```

add tailcall status to all
tailcall sensitive
evaluation functions



```
static EXP_type evaluate_expression(EXP_type Expression,
                                    EXP_type Tail_call)
```

Proper Tail Recursion (cont'd)

```
typedef
  struct CNT { CEL_type hdr;
               NBR_type tid;
               CNT_type cnt;
               EXP_type tcs;
               EXP_type exp[]; } CNT;
```

```
static EXP_type continue_with(EXP_type Value)
  { CNT_type continuation;
    CCC_type function;
    EXP_type call_status;
    NBR_type thread_id;
    continuation = Thread_Peek();
    thread_id   = continuation->tid;
    call_status = continuation->tcs;
    function = Thread_Retrieve(thread_id);
    return function(Value,
                    call_status); }
```

```
static EXP_type continue_application(EXP_type Procedure,
                                     EXP_type Tail_call)
  { ... }
```

Proper Tail Recursion (cont’d)

embed tailcall status in continuation frame

```
typedef
  struct CNT { CEL_type hdr;
               NBR_type tid;
               CNT_type cnt;
               EXP_type tcs;
               EXP_type exp[]; } CNT;
```

```
static EXP_type continue_with(EXP_type Value)
  { CNT_type continuation;
    CCC_type function;
    EXP_type call_status;
    NBR_type thread_id;
    continuation = Thread_Peek();
    thread_id   = continuation->tid;
    call_status = continuation->tcs;
    function = Thread_Retrieve(thread_id);
    return function(Value,
                    call_status); }
```

```
static EXP_type continue_application(EXP_type Procedure,
                                     EXP_type Tail_call)
  { ... }
```

Proper Tail Recursion (cont'd)

embed tailcall status in continuation frame

```
typedef
  struct CNT { CEL_type hdr;
               NBR_type tid;
               CNT_type cnt;
               EXP_type tcs;
               EXP_type exp[]; } CNT;
```

```
static EXP_type continue_with(EXP_type Value)
  { CNT_type continuation;
    CCC_type function;
    EXP_type call_status;
    NBR_type thread_id;
    continuation = Thread_Peek();
    thread_id   = continuation->tid;
    call_status = continuation->tcs;
    function = Thread_Retrieve(thread_id);
    return function(Value,
                    call_status); }
```

extract tailcall status from continuation frame

```
static EXP_type continue_application(EXP_type Procedure,
                                     EXP_type Tail_call)
  { ... }
```

Proper Tail Recursion (cont'd)

embed tailcall status in continuation frame

```
typedef
  struct CNT { CEL_type hdr;
               NBR_type tid;
               CNT_type cnt;
               EXP_type tcs;
               EXP_type exp[]; } CNT;
```

```
static EXP_type continue_with(EXP_type Value)
  { CNT_type continuation;
    CCC_type function;
    EXP_type call_status;
    NBR_type thread_id;
    continuation = Thread_Peek();
    thread_id   = continuation->tid;
    call_status = continuation->tcs;
    function = Thread_Retrieve(thread_id);
    return function(Value,
                        call_status); }
```

extract tailcall status from continuation frame

transmit tailcall status to continuation function

```
static EXP_type continue_application(EXP_type Procedure,
                                     EXP_type Tail_call)
  { ... }
```

Proper Tail Recursion (cont'd)

```
static EXP_type evaluate_body(VEC_type Body,
                              VEC_type Environment,
                              VEC_type Frame,
                              EXP_type Tail_call)
  { prepare_environment_restore_with_poke(Tail_call);
    Environment_Set_Environment_And_Frame(Environment,
                                          Frame);
    return evaluate_sequence(Body,
                             Main_True); }
```

```
static NIL_type prepare_environment_restore_with_poke(EXP_type Tail_call)
  { bOD_type body_thread;
    if (is_FLS(Tail_call))
      { body_thread = (bOD_type)Thread_Poke(Continue_body,
                                            Main_False,
                                            bOD_size);
        body_thread->env = Environment_Get_Environment();
        body_thread->frm = Environment_Get_Frame();
        return; }
    Thread_Zap(); }
```

Proper Tail Recursion (cont'd)

```
static EXP_type evaluate_body(VEC_type Body,
                              VEC_type Environment,
                              VEC_type Frame,
                              EXP_type Tail_call)
  { prepare_environment_restore_with_poke(Tail_call);
    Environment_Set_Environment_And_Frame(Environment,
                                          Frame);
    return evaluate_sequence(Body,
                             Main_True); }
```

application of tailcall hint

```
static NIL_type prepare_environment_restore_with_poke(EXP_type Tail_call)
  { bOD_type body_thread;
    if (is_FLS(Tail_call))
      { body_thread = (bOD_type)Thread_Poke(Continue_body,
                                            Main_False,
                                            bOD_size);
        body_thread->env = Environment_Get_Environment();
        body_thread->frm = Environment_Get_Frame();
        return; }
    Thread_Zap(); }
```

Proper Tail Recursion (cont'd)

```
static EXP_type evaluate_body(VEC_type Body,
                              VEC_type Environment,
                              VEC_type Frame,
                              EXP_type Tail_call)
  { prepare_environment_restore_with_poke(Tail_call);
    Environment_Set_Environment_And_Frame(Environment,
                                          Frame);
    return evaluate_sequence(Body,
                             Main_True); }
```

application of tailcall hint

origin of tailcall hint

```
static NIL_type prepare_environment_restore_with_poke(EXP_type Tail_call)
  { bOD_type body_thread;
    if (is_FLS(Tail_call))
      { body_thread = (bOD_type)Thread_Poke(Continue_body,
                                            Main_False,
                                            bOD_size);
        body_thread->env = Environment_Get_Environment();
        body_thread->frm = Environment_Get_Frame();
        return; }
    Thread_Zap(); }
```

Proper Tail Recursion (cont'd)

```
static EXP_type evaluate_body(VEC_type Body,
                              VEC_type Environment,
                              VEC_type Frame,
                              EXP_type Tail_call)
  { prepare_environment_restore_with_poke(Tail_call);
    Environment_Set_Environment_And_Frame(Environment,
                                          Frame);
    return evaluate_sequence(Body,
                             Main_True); }
```

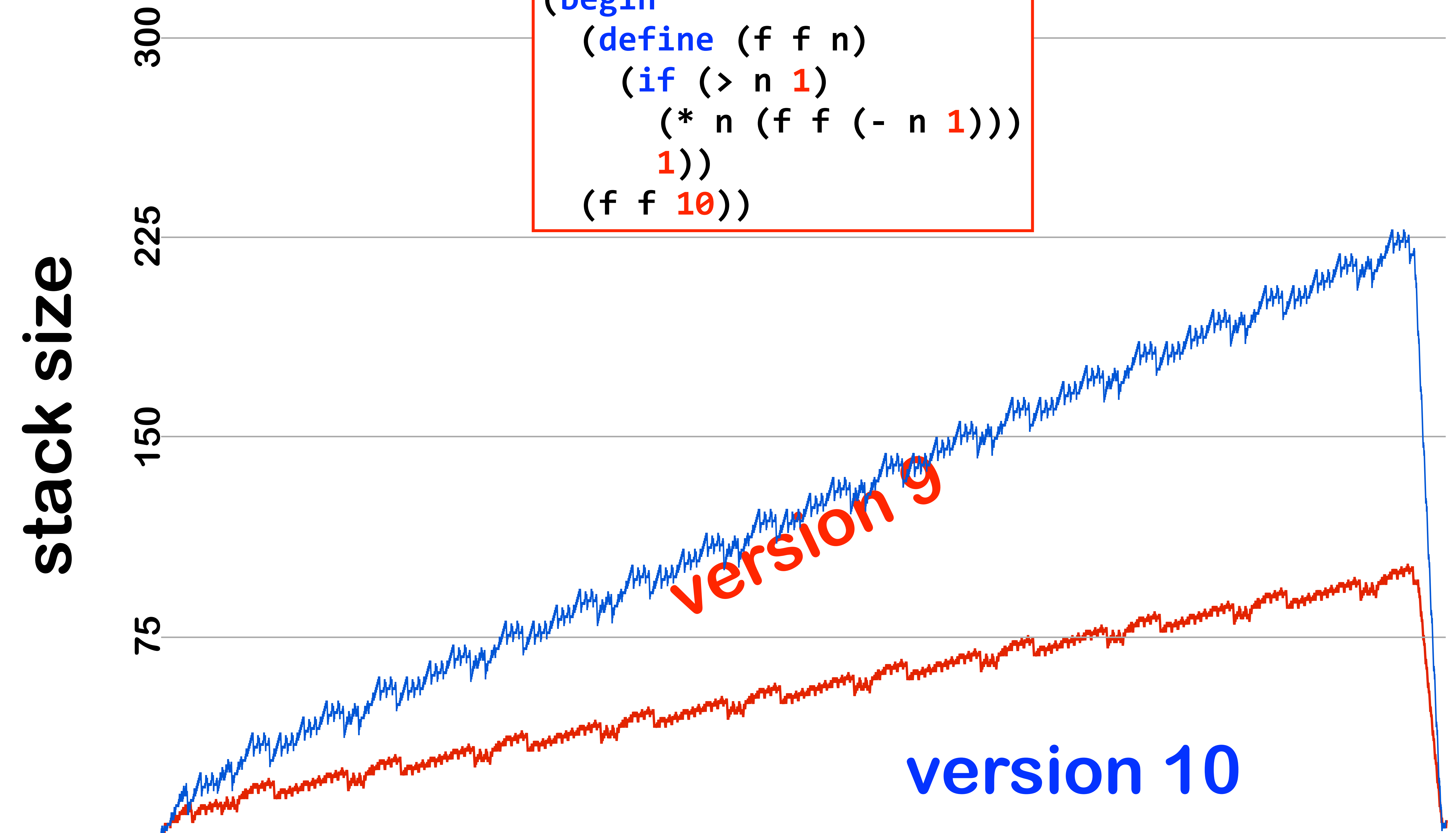
application of tailcall hint

origin of tailcall hint

elimination of stack increment on tailcall hint

```
static NIL_type prepare_environment_restore_with_poke(EXP_type Tail_call)
  { bOD_type body_thread;
    if (is_FLS(Tail_call))
      { body_thread = (bOD_type)Thread_Poke(Continue_body,
                                            Main_False,
                                            bOD_size);
        body_thread->env = Environment_Get_Environment();
        body_thread->frm = Environment_Get_Frame();
        return; }
    Thread_Zap(); }
```

Proper Tail Recursion (cont'd)



First-class Continuations

```
(begin
  (define *first-task* ())
  (define *last-task* ())
  (define *return* ())

  (define (enqueue cont)
    (if (null? *last-task*)
      (begin
        (set! *last-task* (list cont))
        (set! *first-task* *last-task*))
      (begin
        (set-cdr! *last-task* (list cont))
        (set! *last-task* (cdr *last-task*)))))

  (define (dequeue)
    (if (not (null? *first-task*))
      (begin
        (define cont (car *first-task*))
        (set! *first-task* (cdr *first-task*))
        (if (null? *first-task*)
          (set! *last-task* ()))
          (cont #f))
        (*return* #f)))

  (define (schedule task)
    (enqueue (lambda (ignore)
               (task)
               (dequeue))))

  (define (yield)
    (call-cc
      (lambda (cont)
        (enqueue cont)
        (dequeue))))

  (define (start)
    (call-cc
     (lambda (cont)
       (set! *return* cont)
       (dequeue))))

 (define (print op id)
  (display op)
  (display " task ")
  (display id)
  (newline))

(define (report id prime)
  (display id)
  (display " finds a prime: ")
  (display prime)
  (newline)
  (yield))

(define (primes id n)
  (define mask (make-vector n #t))
  (define limit (sqrt n))
  (print "start" id)
  (define i 2)
  (while (<= i limit)
    (if (vector-ref mask i)
      (begin
        (report id i)
        (define j (+ i i))
        (while (< j n)
          (vector-set! mask j #f)
          (set! j (+ j i)))))
    (set! i (+ i 1)))
  (while (< i n)
    (if (vector-ref mask i )
      (report id i))
    (set! i (+ i 1)))
  (print "stop" id))

(display "begin ParallelPrimes")
(newline)
(schedule (lambda () (primes 'A 12)))
(schedule (lambda () (primes 'B 30)))
(schedule (lambda () (primes 'C 20)))
(start)
(display "end ParallelPrimes")
(newline))
```

First-class Continuations

```
(begin
  (define *first-task* ())
  (define *last-task* ())
  (define *return* ())

  (define (enqueue cont)
    (if (null? *last-task*)
      (begin
        (set! *last-task* (list cont))
        (set! *first-task* *last-task*))
      (begin
        (set-cdr! *last-task* (list cont))
        (set! *last-task* (cdr *last-task*)))))

  (define (dequeue)
    (if (not (null? *first-task*))
      (begin
        (define cont (car *first-task*))
        (set! *first-task* (cdr *first-task*))
        (if (null? *first-task*)
          (set! *last-task* ()))
        (cont #f))
      (*return* #f)))

  (define (schedule task)
    (enqueue (lambda (ignore)
               (task)
               (dequeue))))

  (define (yield)
    (call-cc
      (lambda (cont)
        (enqueue cont)
        (dequeue))))

  (define (start)
    (call-cc
     (lambda (cont)
       (set! *return* cont)
       (dequeue))))

   (define (print op id)
    (display op)
    (display " task ")
    (display id)
    (newline))

  (define (report id prime)
    (display id)
    (display " finds a prime: ")
    (display prime)
    (newline)
    (yield))

  (define (primes id n)
    (define mask (make-vector n #t))
    (define limit (sqrt n))
    (print "start" id)
    (define i 2)
    (while (<= i limit)
      (if (vector-ref mask i)
        (begin
          (report id i)
          (define j (+ i i))
          (while (< j n)
            (vector-set! mask j #f)
            (set! j (+ j i)))))
      (set! i (+ i 1)))
    (while (< i n)
      (if (vector-ref mask i )
        (report id i))
      (set! i (+ i 1)))
    (print "stop" id))

  (display "begin ParallelPrimes")
  (newline)
  (schedule (lambda () (primes 'A 12)))
  (schedule (lambda () (primes 'B 30)))
  (schedule (lambda () (primes 'C 20)))
  (start)
  (display "end ParallelPrimes")
  (newline))
```

cfr. cpSlip...

as easy in C?

First-class Continuations

```
(begin
  (define *first-task* ())
  (define *last-task* ())
  (define *return* ())

  (define (enqueue con
    (if (null? *last-t
      (begin
        (set! *last-ta
        (set! *first-t
      (begin
        (set-cdr! *last-task* (list cont))
        (set! *last-task* (cdr *last-task*)))))

  (define (dequeu
    (if (not (nul
      (begin
        (define c
        (set! *fi
        (if (null
          (set! *
          (cont #
        (*return* #f)))

  (define (schedule task)
    (enqueue (lambda (ignore)
               (task)
               (dequeue))))

  (define (yield)
    (call-cc
      (lambda (cont)
        (enqueue cont)
        (dequeue))))

  (define (start)
    (call-cc
     (lambda (cont)
       (set! *return* cont)
       (dequeue))))
```

```
(define (print op id)
  (display op)
  (display " task ")

  (yield))

 (define (primes id n)
                                  #t))

        (define j (+ i i))
        (while (< j n)
          (vector-set! mask j #f)
          (set! j (+ j i)))))
      (set! i (+ i 1)))
    (while (< i n)
      (if (vector-ref mask i )
        (report id i))
      (set! i (+ i 1)))
    (print "stop" id))

 (display "begin ParallelPrimes")
 (newline)
 (schedule (lambda () (primes 'A 12)))
 (schedule (lambda () (primes 'B 30)))
 (schedule (lambda () (primes 'C 20)))
 (start)
 (display "end ParallelPrimes")
 (newline))
```

cfr. cpSlip ...

as easy in C?

First-class Continuations (cont'd)

```
typedef
  struct CNT { CEL_type hdr;
               NBR_type tid;
               CNT_type cnt;
               EXP_type tcs;
               EXP_type exp[]; } CNT;
```

change of terminology

First-class Continuations (cont'd)

```
typedef
  struct THR { CEL_type hdr;
               NBR_type tid;
               THR_type thr;
               EXP_type tcs;
               EXP_type exp[]; } THR;
```

change of terminology

"continuation" becomes "thread"

First-class Continuations (cont'd)

```
typedef
  struct CNT { CEL_type hdr;
               VEC_type env;
               VEC_type frm;
               THR_type thr; } CNT;
BYT_type   is_CNT(EXP_type);
CNT_type make_CNT(VEC_type,
                  VEC_type,
                  THR_type);
```

First-class Continuations (cont'd)

```
typedef
  struct CNT { CEL_type hdr;
               VEC_type env;
               VEC_type frm;
               THR_type thr; } CNT;
BYT_type   is_CNT(EXP_type);
CNT_type make_CNT(VEC_type,
                  VEC_type,
                  THR_type);
```

"continuation" refers to "1st class continuation"

First-class Continuations (cont’d)

capture the environment + frame

```
typedef
  struct CNT { CEL_type hdr;
               VEC_type env;
               VEC_type frm;
               THR_type thr; } CNT;
BYT_type   is_CNT(EXP_type);
CNT_type make_CNT(VEC_type,
                  VEC_type,
                  THR_type);
```

First-class Continuations (cont'd)

capture
the thread

```
typedef
  struct CNT { CEL_type hdr;
               VEC_type env;
               VEC_type frm;
               THR_type thr; } CNT;
BYT_type   is_CNT(EXP_type);
CNT_type make_CNT(VEC_type,
                  VEC_type,
                  THR_type);
```

First-class Continuations (cont'd)

```c
static const TXT_type ccc_string = "call-cc";
```

```c
static EXP_type call_cc_native(VEC_type Vector,
                               EXP_type Tail_call)
  { CNT_type continuation;
    EXP_type procedure;
    PAI_type arguments;
    THR_type thread;
    UNS_type raw_size;
    VEC_type environment,
             frame;
    raw_size = size_VEC(Vector);
    if (raw_size != 1)
      return Main_Error_Text(EX1_error_string,
                             ccc_string);
    procedure = Vector[1];
    environment = Environment_Get_Environment();
    frame = Environment_Global_Frame();
    thread = Thread_Mark();
    continuation = make_CNT(environment,
                            frame,
                            thread);
    arguments = make_PAI(continuation,
                         Main_Null);
    return Evaluate_Apply(procedure,
                          arguments,
                          Tail_call); }
```

```c
static NIL_type initialize_call_cc_native(NIL_type)
  { special_native_define(ccc_string,
      call_cc_native); }
```

First-class Continuations (cont'd)

```
static const TXT_type ccc_string = "call-cc";
```

```
static EXP_type call_cc_native(VEC_type Vector,
                               EXP_type Tail_call)
  { CNT_type continuation;
    EXP_type procedure;
```

identify procedural argument of call-cc

```
    raw_size = size_VEC(Vector);
    if (raw_size != 1)
      return Main_Error_Text(EX1_error_string,
                             ccc_string);
    procedure = Vector[1];
    environment = Environment_Get_Environment();
    frame = Environment_Global_Frame();
    thread = Thread_Mark();
    continuation = make_CNT(environment,
                            frame,
                            thread);
    arguments = make_PAI(continuation,
                         Main_Null);
    return Evaluate_Apply(procedure,
                          arguments,
                          Tail_call); }
```

```
static NIL_type initialize_call_cc_native(NIL_type)
  { special_native_define(ccc_string,
                          call_cc_native); }
```

First-class Continuations (cont'd)

```
static const TXT_type ccc_string = "call-cc";
```

```
static EXP_type call_cc_native(VEC_type Vector,
                               EXP_type Tail_call)
  { CNT_type continuation;
    EXP_type procedure;
    PAI_type arguments;
    THR_type thread;
    UNS_type raw_size;
    VEC_type environment,
             frame;
    raw_size = size_VEC(Vector);
    if (raw_size != 1)
      return Main_Error_Text(EX1_error_string,
                             ccc_string);
    procedure = Vector[1];
    environment = Environment_Get_Environment();
    frame = Environment_Global_Frame();
    thread = Thread_Mark();
    continuation = make_CNT(environment,
                            frame,
                            thread);
    arguments = make_PAI(continuation,
                         Main_Null);
    return Evaluate_Apply(procedure,
                          arguments,
                          Tail_call); }
```

construct continuation

```
static NIL_type initialize_call_cc_native(NIL_type)
  { special_native_define(ccc_string,
    call_cc_native); }
```

First-class Continuations (cont'd)

```
static const TXT_type ccc_string = "call-cc";
```

```
static EXP_type call_cc_native(VEC_type Vector,
                               EXP_type Tail_call)
  { CNT_type continuation;
    EXP_type procedure;
    PAI_type arguments;
    THR_type thread;
    UNS_type raw_size;
    VEC_type environment,
             frame;
    raw_size = size_VEC(Vecto
    if (raw_size != 1)
      return Main_Error_Text(EX1_error_string,
                             ccc_string);
    procedure = Vector[1];
    environment = Environment_Get_Environment();
    frame = Environment_Global_Frame();
    thread = Thread_Mark();
    continuation = make_CNT(environment,
                            frame,
                            thread);
    arguments = make_PAI(continuation,
                         Main_Null);
    return Evaluate_Apply(procedure,
                          arguments,
                          Tail_call); }
```

build argument list with continuation

```
static NIL_type initialize_call_cc_native(NIL_type)
  { special_native_define(ccc_string,
                          call_cc_native); }
```

First-class Continuations (cont'd)

```
static const TXT_type ccc_string = "call-cc";
```

```
static EXP_type call_cc_native(VEC_type Vector,
                               EXP_type Tail_call)
  { CNT_type continuation;
    EXP_type procedure;
    PAI_type arguments;
    THR_type thread;
    UNS_type raw_size;
    VEC_type environment,
             frame;
    raw_size = size_VEC(Vector);
    if (raw_size != 1)
      return Main_Error_Text(EX1_error_string,
                             ccc_string);
    procedure = Vector[1];
    environment = Environment_Get_Environment();
    frame = Environment_Global_Frame();
    thread = Thread_Mark();
    continuation = make_CNT(environment,
                            frame,
                            thread);
    arguments = make_PAI(continuation,
                         Main_Null);
    return Evaluate_Apply(procedure,
                          arguments,
                          Tail_call); }
```

call procedure

```
static NIL_type initialize_call_cc_native(NIL_type)
  { special_native_define(ccc_string,
      call_cc_native); }
```

Interference with Iterations

```
static EXP_type continue_while_predicate(EXP_type Boolean,
                                         EXP_type Tail_call)
  {     ...
    while_thread = (wHI_type)Thread_Patch(Continue_while_body);
        ...
    return evaluate_inline_sequence(body,
                                    raw_body_size,
                                    Main_False); }

static EXP_type continue_while_body(EXP_type Value,
                                    EXP_type Tail_call)
  {     ...
    while_thread = (wHI_type)Thread_Patch(Continue_while_predicate);
    while_thread->res = Value;
        ...
    return evaluate_inline_with_push(predicate,
                                     raw_predicate_size,
                                     Main_False); }

static EXP_type evaluate_while(WHI_type While)
  {     ...
    while_thread = (wHI_type)Thread_Push(Continue_while_predicate,
                                         Main_False,
                                         wHI_size);
        ...
    return evaluate_inline_with_push(predicate,
                                     raw_predicate_size,
                                     Main_False); }
```

Interference with Iterations

```
static EXP_type continue_while_predicate(EXP_type Boolean,
                                         EXP_type Tail_call)
  {     ...
    while_thread = (wHI_type)Thread_Patch(Continue_while_body);
        ...
    return evaluate_inline_sequence(body,
                                    raw_body_size,
                                    Main_False); }

static EXP_type continue_while_body(EXP_type Value,
                                    EXP_type Tail_call)
  {     ...
    while_thread = (wHI_type)Thread_Patch(Continue_while_predicate);
    while thread->res = Value;
```

prepare evaluation of predicate

```
                                ith_push(predicate,
                                         raw_predicate_size,
                                         Main_False); }

static EXP_type evaluate_while(WHI_type While)
  {     ...
    while_thread = (wHI_type)Thread_Push(Continue_while_predicate,
                                         Main_False,
                                         wHI_size);
        ...
    return evaluate_inline_with_push(predicate,
                                     raw_predicate_size,
                                     Main_False); }
```

Interference with Iterations

```
static EXP_type continue_while_predicate(EXP_type Boolean,
                                         EXP_type Tail_call)
  {     ...
    while_thread = (wHI_type)Thread_Patch(Continue_while_body);
        ...
    return evaluate_inline_sequence(body,
                                    raw_body_size,
                                    Main_False); }

static EXP_type continue_while_body(EXP_type Value,
                                    EXP_type Tail_call)
  {     ...
    while_thread = (wHI_type)Thread_Patch(Continue_while_predicate);
    while_thread->res = Value;
        ...
    return evaluate_inline_with_push(

static EXP_type evaluate_while(WHI_type While)
  {     ...
    while_thread = (wHI_type)Thread_Push(Continue_while_predicate,
                                         Main_False,
                                         wHI_size);
        ...
    return evaluate_inline_with_push(predicate,
                                     raw_predicate_size,
                                     Main_False); }
```

evaluate predicate

Interference with Iterations

```
static EXP_type continue_while_predicate(EXP_type Boolean,
                                         EXP_type Tail_call)
  {     ...
    while_thread = (wHI_type)Thread_Patch(Continue_while_body);
        ...
    return evaluate_inline_sequence(body,
                                    raw_body_size,
                                    Main_False); }

static EXP_type continue_whi

  {     ...
    while_thread = (wHI_type
    while_thread->res = Valu
        ...
    return evaluate_inline_with_push(predicate,
                                     raw_predicate_size,
                                     Main_False); }

static EXP_type evaluate_while(WHI_type While)
  {     ...
    while_thread = (wHI_type)Thread_Push(Continue_while_predicate,
                                         Main_False,
                                         wHI_size);
        ...
    return evaluate_inline_with_push(predicate,
                                     raw_predicate_size,
                                     Main_False); }
```

change identity of thread without replacing the thread-frame

Interference with Iterations

```
static EXP_type continue_while_predicate(EXP_type Boolean,
                                         EXP_type Tail_call)
  {     ...
    while_thread = (wHI_type)Thread_Patch(Continue_while_body);
        ...
    return evaluate_inline_sequence(body,
                                    raw_body_size,
                                    Main_False); }

static EXP_type continue_while_body(EXP_type Value,
                                    EXP_type Tail_call)
  {     ...
    while_thread = (wHI_type)Thread_Patch(Continue_while_predicate);
    while_thread->res = Value;
        ...
    return evaluate_inline_with_push(predicate,
                                     raw_predicate_size,
                                     Main_False); }

static EXP_type evaluate_while(WHI_type While)
  {     ...
    while_thread = (wHI_type)Thread_Push(Continue_while_predicate,
                                         Main_False,
                                         wHI_size);
        ...
    return evaluate_inline_with_push(predicate,
                                     raw_predicate_size,
                                     Main_False); }
```

evaluate body

Interference with Iterations

```
static EXP_type continue_while_predicate(EXP_type Boolean,
                                         EXP_type Tail_call)
  {     ...
    while_thread = (wHI_type)Thread_Patch(Continue_while_body);
        ...
    return evaluate_inline_sequence(body,
                                    raw_body_size,
                                    Main_False); }

static EXP_type continue_while_body(EXP_type Value,
                                    EXP_type Tail_call)
  {     ...
    while_thread = (wHI_type)Thread_Patch(Continue_while_predicate);
    while_thread->res = Va
        ...
    return evaluate_inline

static EXP_type evaluate_w
  {     ...
    while_thread = (wHI_type)Thread_Push(Continue_while_predicate,
                                         Main_False,
                                         wHI_size);
        ...
    return evaluate_inline_with_push(predicate,
                                     raw_predicate_size,
                                     Main_False); }
```

change identity of thread without replacing the thread-frame

Interference with Iterations

```
static EXP_type continue_while_predicate(EXP_type Boolean,
                                         EXP_type Tail_call)
  {     ...
    while_thread = (wHI_type)Thread_Patch(Continue_while_body);
        ...
    return evaluate_inline_sequence(body,
                                    raw_body_size,
                                    Main_False); }

static EXP_type continue_while_body(
                                    
  {     ...
    while_thread = (wHI_type)Thread_Patch(Continue_while_predicate);
    while_thread->res = Value;
        ...
    return evaluate_inline_with_push(predicate,
                                     raw_predicate_size,
                                     Main_False); }

static EXP_type evaluate_while(WHI_type While)
  {     ...
    while_thread = (wHI_type)Thread_Push(Continue_while_predicate,
                                         Main_False,
                                         wHI_size);
        ...
    return evaluate_inline_with_push(predicate,
                                     raw_predicate_size,
                                     Main_False); }
```

evaluate predicate

Interference with Iterations(cont’d)

until now ...

```
Thread_Keep()
```

maintain current thread without change

```
Thread_Patch(Thread_id)
```

replace thread id in current thread

Interference with Iterations(cont'd)

as of now ...

`Thread_Keep()`

clone current thread
if necessary

`Thread_Patch(Thread_id)`

clone current thread
if necessary and replace
thread id

Interference with Iterations(cont'd)

necessary = if thread is part of 1st class continuation

as of now ...

```
Thread_Keep()
```

clone current thread if necessary

```
Thread_Patch(Thread_id)
```

clone current thread if necessary and replace thread id

Interference with Iterations(cont'd)

as of now ...

`Thread_Keep()`

necessary = if thread is part of 1st class continuation

clone current thread
if necessary

`Thread_Patch(Thread_id)`

clone = shallow copy except for top thread-frame

rent thread
ary and replace
thread id

Interference with Iterations(cont'd)

```
static EXP_type call_cc_native(VEC_type Vector,
                               EXP_type Tail_call)
  { CNT_type continuation;
    EXP_type procedure;
    PAI_type arguments;
    THR_type thread;
    UNS_type raw_size;
    VEC_type environment,
             frame;
    raw_size = size_VEC(Vector);
    if (raw_size != 1)
      return Main_Error_Text(EX1_error_string,
                             ccc_string);
    procedure = Vector[1];
    environment = Environment_Get_Environment();
    frame = Environment_Global_Frame();
    thread = Thread_Mark();
    continuation = make_CNT(environment,
                            frame,
                            thread);
    arguments = make_PAI(continuation,
                         Main_Null);
    return Evaluate_Apply(procedure,
                          arguments,
                          Tail_call); }
```

Interference with Iterations(cont'd)

```
static EXP_type call_cc_native(VEC_type Vector,
                               EXP_type Tail_call)
  { CNT_type continuation;
             procedure;
             arguments;
             thread;
             raw_size;
             environment,
             frame;
    raw_size = size_VEC(Vector);
    if (raw_size != 1)
      return Main_Error_Text(EX1_error_string,
                             ccc_string);
    procedure = Vector[1];
    environment = Environment_Get_Environment();
    frame = Environment_Global_Frame();
    thread = Thread_Mark();
    continuation = make_CNT(environment,
                            frame,
                            thread);
    arguments = make_PAI(continuation,
                         Main_Null);
    return Evaluate_Apply(procedure,
                          arguments,
                          Tail_call); }
```

solution: mark thread

Interference with Iterations(cont'd)

solution: mark thread

```
static EXP_type call_cc_native(VEC_type Vector,
                               EXP_type Tail_call)
  { CNT_type continuation;
             procedure;
             arguments;
             thread;
             raw_size;
             environment
             frame;
    raw_size = size_VEC(
    if (raw_size != 1)
      return Main_Error_Text(EXT_error_string,
                             ccc_string);
    procedure = Vector[1];
    environment = Environment_Get_Environment();
    frame = Environment_Global_Frame();
    thread = Thread_Mark();
    continuation = make_CNT(environment,
                            frame,
                            thread);
    arguments = make_PAI(continuation,
                         Main_Null);
    return Evaluate_Apply(procedure,
                          arguments,
                          Tail_call); }
```

```
THR_type Thread_Mark(NIL_type)
  { THR_type thread;
    for (thread = Threaded_continuation;
         !is_NUL(thread);
         thread = thread->thr)
      mark_THR(thread);
    return Threaded_continuation; }
```

Interference with Iterations(cont'd)

solution: mark thread

```
static EXP_type call_cc_native(VEC_type Vector,
                               EXP_type Tail_call)
  { CNT_type continuation;
             procedure;
             arguments;
             thread;
             raw_size;
             environment
             frame;
    raw_size = size_VEC(
    if (raw_size != 1)
      return Main_Error_Text(EX
                             co
    procedure = Vector[1];
    environment = Environment_Get_Environment();
    frame = Environment_Global_Frame();
    thread = Thread_Mark();
    continuation = make_CNT(environment,
                            frame,
                            thread);
    arguments = make_PAI(continuation,
                         Main_Null);
    return Evaluate_Apply(procedure,
                          arguments,
                          Tail_call); }
```

```
THR_type Thread_Mark(NIL_type)
  { THR_type thread;
    for (thread = Threaded_continuation;
         !is_NUL(thread);
         thread = thread->thr)
      mark_THR(thread);
    return Threaded_continuation; }
```

```
NIL_type mark_THR(THR_type Thread)
  { Memory_Set_Tag((PTR_type)Thread,
                   THR_tag); }
```

Interference with Iterations(cont'd)

solution: mark thread

```
static EXP_type call_cc_native(VEC_type Vector,
                               EXP_type Tail_call)
  { CNT_type continuation;
             procedure;
             arguments;
             thread;
             raw_size;
             environment
             frame;
    raw_size = size_VEC(
    if (raw_size != 1)
      return Main_Error_Text(EX
                           co
    procedure = Vector[1];
    environment = Environment_Get_Environment(
    frame = Environment_Global_Frame();
    thread = Thread_Mark();
    continuation = make_CNT(environment,
                            frame,
                            thread);
    arguments = make_PAI(continuation,
                         Main_Null);
    return Evaluate_Apply(procedure,
                          arguments,
                          Tail_call); }
```

```
THR_type Thread_Mark(NIL_type)
  { THR_type thread;
    for (thread = Threaded_continuation;
         !is_NUL(thread);
         thread = thread->thr)
      mark_THR(thread);
    return Threaded_continuation; }
```

```
NIL_type mark_THR(THR_type Thread)
  { Memory_Set_Tag((PTR_type)Thread,
                   THR_tag); }
```

```
THR_tag = 0x14<<1 | 0x0,
THr_tag = 0x15<<1 | 0x0,
```

A fully functional Slip interpreter

Slip – Debugger Console

10.6 | Debug — Overview | Breakpoints | Build and Debug | Tasks | Restart | Pause | Debugger | Clear Log

```
[Session started at 2010-05-05 11:55:26 +0200.]
GNU gdb 6.3.50-20050815 (Apple version gdb-1461.2) (Fri Mar  5 04:43:10 UTC 2010)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys000
sharedlibrary apply-load-rules all
Loading program into debugger…
Program loaded.
run
[Switching to process 19291]
Running…
cpSlip/c version 11
>>>(eval (read "Primes.scm"))
begin ParallelPrimes
start task A
A finds a prime: 2
start task B
B finds a prime: 2
start task C
C finds a prime: 2
A finds a prime: 3
B finds a prime: 3
C finds a prime: 3
A finds a prime: 5
B finds a prime: 5
C finds a prime: 5
A finds a prime: 7
B finds a prime: 7
C finds a prime: 7
A finds a prime: 11
B finds a prime: 11
C finds a prime: 11
stop task A
B finds a prime: 13
C finds a prime: 13
B finds a prime: 17
C finds a prime: 17
B finds a prime: 19
C finds a prime: 19
B finds a prime: 23
stop task C
B finds a prime: 29
stop task B
end ParallelPrimes
<unspecified>
>>>
```

GDB: Running...